FragDB – Secure Localized Storage Based on Super-Distributed RFID-Tag Infrastructures

Marc Langheinrich
Institute for Pervasive Computing, ETH Zurich
8092 Zurich, Switzerland
langhein@inf.ethz.ch

Abstract

Smart environments and wearables will make the storage and subsequent sharing of digitized multimedia diaries and meeting protocols – whom we meet, or what we say or do – cheap and easy. However, controlling access to this data will become cumbersome if traditional forms of access control are to be used: Overly restrictive rules might deny the potential of data sharing, while a lack of control could easily lead to Orwellian surveillance scenarios. This paper presents FragDB, a storage concept based on localized access control, where data storage and retrieval are bound to a specific place, *rather than a the knowledge of a particular password or certificate. FragDB uses tiny RFID tags embedded in floors, walls or desks, to compute a local key that is used to encrypt and decrypt data in a global storage system. We describe the setup of an initial prototype and analyze its complexity.*

1. Introduction

With storage media continuously dropping in price, the vision of storing *all* events of our lives, in the manner of a 24/7 multimedia diary, seems soon to become feasible. So-called "capture and access" projects such as Classroom2000/eClass [1] or Teamspace [4] have employed audio and video recording to make lectures and meetings accessible to attendees and external guests for later perusal. Other "active spaces" research, such as Microsoft's *Easy Living* [3], Stanford's *interactive workspaces* [6] or *Gaia* at the University of Illinois at Urbana Champaign [8], envision the comprehensive digitization of our lives in order to provide novel services and smart reactive environments. All of these projects have recognized that controlling access to the stored data requires novel access control schemes, as traditional forms of role-based access control (RBAC) fall short of the required flexibility.

A number of researchers have begun to explore the use of location as an access control parameter, thus allowing users to regulate access to particular data not only on *who* wants to access it, but *from where* [5, 7, 9, 10]. However, common to all approaches is the need for *explicit* access control, i.e., data owners will need to formulate and adjust security policies in order to properly regulate access to the stored data. While this might be feasible in an office setting, e.g., where employees are used (or required) to protect sensitive documents, many novel types of data acquired by active spaces might be difficult to properly assign an access category. In many situations, *implicit* access control might be sufficient, which uses *situated privacy controls* to limit data access. Situated access is not regulated explicitly through security parameters of access policies, but implicitly through time and space. With situated access control, only those close enough in time and space will be able to "witness", i.e., retrieve, stored data, while those far away, both in time and space, will not.

While this "free for all, if near enough" approach might sound counterintuitive for traditional data sets, such as contact information, health or financial data, it might be sufficient for semi-public data that does not warrant explicit protection, but which nevertheless should be prevented from being globally and eternally available. An example would be the above-mentioned meeting rooms, where the individual participants could store logs of their own wearable sensors directly in the meeting room, allowing participants who come late, or maybe even next year's students, to easily find it there. Smart vehicles could store information on road conditions or encountered hazards, say, at mile 27, on tags they remembered a few miles ahead, e.g., at mile 25 (and maybe later again at mile 29). This would allow the following traffic to be informed in time, without

giving an outside observer any information on the actual locations of individual cars or events. FragDB attempts to demonstrate the technical feasibility of such an implicit privacy control system.

2. Basic Principles

The basic idea of FragDB is analogous to how people managed their privacy in the past: While certainly under close observation by their neighbors, detailed information about individuals was not available in far away places. In order to find out about someone's past, one had to travel to a person's home town and talk to friends and neighbors. Thus, privacy was an inherent aspect of the locality of a person. Instead of having to *manage* one's privacy, which always entails the possibility of mismanaging it, the limited communication and storage capabilities of the past implicitly hampered the unwanted disclosure of personal facts across spatial and temporal boundaries.

FragDB aims at recreating some part of this inherent privacy of a place, by constructing a system that facilitates a localized storage and retrieval concept. Data is seemingly deposited at a particular location and can only be retrieved by visiting this particular place again. Since FragDB uses a remotely accessible storage system for actual data storage (e.g., a file server), all stored data is encrypted with the particular *fingerprint* of the original storage location. Only if this fingerprint is known, or by physically traveling again to the original storage location to (re-)compute this fingerprint, can data in the storage system be retrieved.

The idea of using the "fingerprint" of a particular location as an access key to a storage repository creates two immediate challenges:

1. *Fluid Boundaries:* One cannot expect to find oneself directly at the same spot for data retrieval as used for data storage. As such, our storage system must be able to tolerate a certain fluidity in positioning, while still recreating the correct access keys (i.e., fingerprint).

2. *Time Variance:* In order to prevent that a one-time readout of a place's fingerprint leads to a perpetual access to all data being stored at this place, the access keys of a place have to periodically change.

Challenge two immediately leads to another complication: once an access key of a place changes – this might happen as fast as every day or every hour – access to this information might be lost forever, unless we have saved the particular key used during storage.

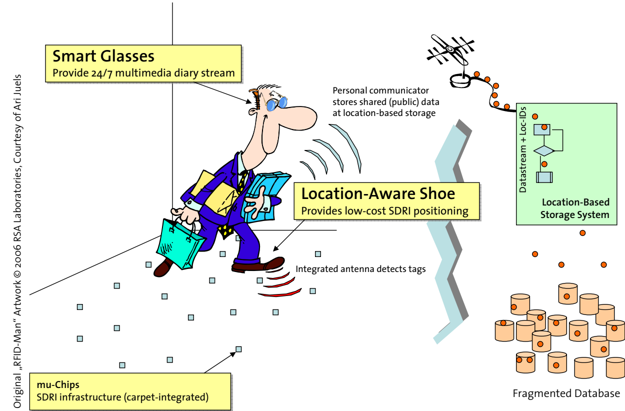


Figure 1. Overview of General Operation.

However, the idea of conveniently sharing semi-public information with people in the vicinity, both time- and space-wise, is the main reason for such an access control scheme – if all we wanted to do is protect personal information, much more effective means would be possible, e.g., local storage in a wearable system, or a biometric encryption key. Our third challenge is thus:

3. *Time Continuity:* Instead of simply exchanging an old fingerprint for a new one, a location needs to keep track of a number of old prints (say, the last five or the last fifty, depending on the resolution), so as to still support the retrieval of data stored at this place in the past. However, old keys should eventually expire, recreating some sort of "forgetfulness" principle.

Note that this only apparently contradicts our time variance principle: While our second challenge addresses the storage of *new* information, the time continuity principles concerns the access to *old* information. New data should continuously be fingerprinted differently, even at the same place, but old fingerprints should continue to "lie around" for a while.

Last but not least, by having a remotely accessible storage system where fingerprinted data is saved, we also must make sure that data access is impossible without knowing the proper key (fingerprint) of its storage location. Otherwise, a simple database scan could reveal any location-bound data stored in it. Thus, our fourth requirement is that of secure storage:

4. *Secure Storage:* Irrespective of actual storage location in cyberspace – be it a server in Boston or Cape Town, or multiple servers distributed around the world – the stored data must be properly encrypted, in order to render database attacks infeasible.

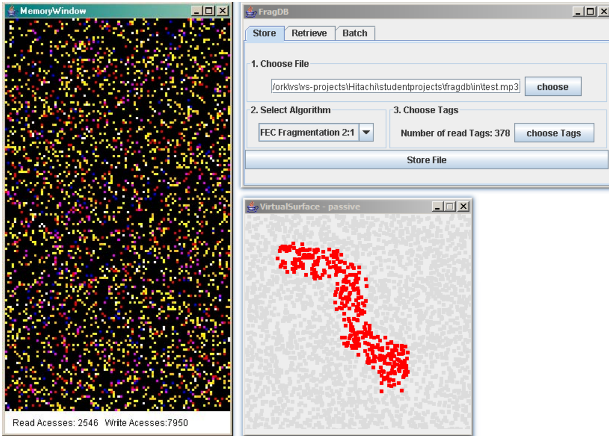


Figure 2. Prototype Interface.

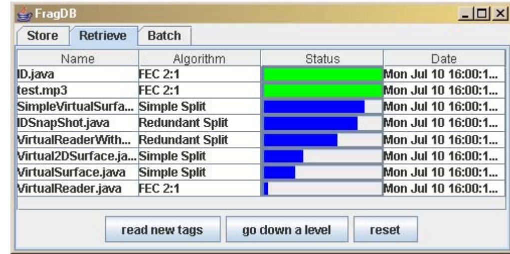


Figure 3. Key Reassembly/Data Access.

In our actual implementation of FragDB, we use the IDs of a large numbers of RFID-tags, embedded in the environment, to serve as the key to a virtual storage location. As RFID-tags can only be read locally with a reader device, we can ensure that users must be at or near the place where data was stored, in order to find the data's access parameters, which then allow data retrieval from anywhere. The idea of incorporating large populations of miniature RFID tags into the environment was first proposed by Bohn and Mattern [2], who envisioned passive RFID tags deployed in vast quantities and in a highly redundant fashion over large areas or object surfaces – so-called *Super Distributed RFID-Tag Infrastructures* (SDRIs) – in order to provide novel services such as positioning or collaboration.

Figure 1 shows the overall operation principle of FragDB. In the example, a user fingerprints a particular SDRI environment – as detected by a shoe antenna – to store publicly available media information from his smart glasses at a location-bound storage cell. The data is fragmented into pieces and can only be related to each other by supplying a considerable subset of IDs (i.e., key fragments) from the SDRI-tags present at that location. The next section describes our prototype implementation in detail.

3. Prototype System

Our FragDB prototype consists of a simulator, allowing us to virtually place RFID-tags on *virtual surfaces* and subsequently simulating the storage and retrieval of data through a set of read-in tag IDs, as well as an actual RFID-reader interface that supports the entire process with real RFID-tags, albeit at a much smaller scale (i.e., typically dozens, instead of thousands of tags). Also, data storage is handled by a generic *storage system interface* that currently stores information in main memory, but which could just as well use a file server or a distributed P2P-storage system.

Figure 2 shows the user interface of the prototype after storing a file in the virtual environment. Centered at the bottom, the *virtual surface* shows a set of tags that have been read in (shaded). Tag selection can be done using a paintbrush-like cursor that allows simulating the process of reading tags on the surface. The *controller window* at the top right can then be used to store, e.g., an audio file at the virtual location of the read tags, using a particular storage algorithm ("FEC Fragmentation 2:1" in this example, see section 4.1). The *memory window* shown at the left side gives a view of the global storage system, indicating the storage cells where data has been placed. In the example, the audio file is divided into a set of individual fragments and stored all across the storage system, in order to make reassembly by a simple storage system scan infeasible. A separate *batch controller* (not shown) allows automating these steps multiple times, i.e., tag selection, file selection, and storage of the file at the selected tag locations, in order to achieve a more realistic system usage.

The set of read tags (shaded in the virtual surface window in figure 2) represents the key for both locating and decrypting the stored data in the storage system – saving this "key" allows the data owner continuous access to the stored data. Users without this key must physically travel to the initial location where the storage was performed (i.e., where the RFID-tags representing the key are located) and reassemble this key. The interface for key reassembly, and thus data access, is shown in figure 3. As during file storage, the user first uses a paintbrush-like cursor to select a set of tags from the virtual surface that should be read in. During tag reading, the system continuously assembles the tag IDs into potential access keys and shows a list of found files under the retrieval-tab of the controller window. In the example, the keys for the two topmost files have been completely reassembled, while keys for six other files have been found but not completely reassembled,

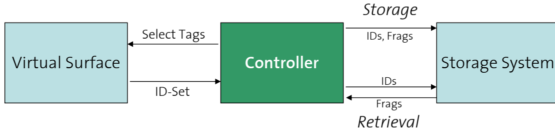


Figure 4. FragDB Controller.

as indicated by the status column. Section 4 below describes the mechanics of this process.

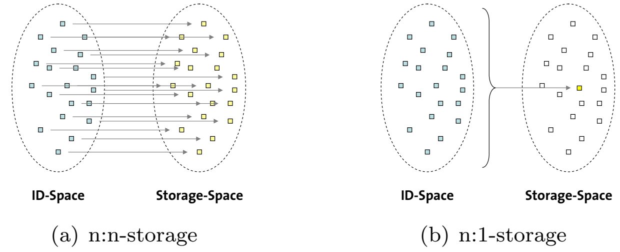
Both storage and retrieval (but not the batch operations) also work with actual RFID hardware. We have connected a Hitachi μ-chip reader to our prototype and affixed about forty μ-chips to a number of cardboards, representing a floor or desk space. μ-chips feature a size of 0.16mm^2 and a stick antenna of about 10cm. They contain a factory-written, read-only 104 bit serial number, which can be read out from up to 5-10cm distance. μ-chips and -readers do not use an anti-collision protocol, so having several μ-chips in range will most likely result in failed readouts. The FragDB prototype maps physical RFID-tags onto a simulated one, thus allowing our μ-chips to support the same features as our simulated ones, i.e., time- or usage-based ID changes, as well as storage of prior IDs.

4. Architecture

Figure 4 shows the general architectural division. A central *controller* interfaces the virtual surface (or, alternatively, a real hardware reader) to receive a set of tags read at a particular location. It then uses these tag IDs to either store data in the storage system, or attempt to retrieve data stored "at" these tag IDs from the storage system. The architecture supports the four distinct features described above: fluid boundaries, time variance, time continuity, and secure storage.

4.1. Fluid Boundaries

A straightforward way of binding a file to a specific set of tags is using the tags' IDs as pointers to individual memory locations, and storing a fragment of the file at each memory cell, as illustrated in figure 5(a). In order to tolerate variances in the tag set, a *fragmentation algorithm* is used that encodes the desired level of redundancy into each fragment, e.g., using a forward error correction code (FEC). The FragDB prototype supports three different kinds of fragmentation algorithms: A *simple split* algorithm simply cuts a file into as many pieces as memory cells available,



(a) n:n-storage (b) n:1-storage

Figure 5. ID-Storage Mapping Options.

with no redundancy. This is useful for streaming data, such as audio or video, where a certain loss of fragments can be tolerated. The *redundant split* algorithm saves each fragment twice, i.e., fragments the file in only half as many pieces as possible. While it is able to tolerate slightly more missing fragments, it is still most useful for streaming media files. The *FEC 2:1* algorithm finally uses Reed-Solomon forward error correcting codes to encode redundancy information evenly across all fragments, allowing the system to reassemble the entire file with *any* half of the fragments.

Alternatively, one could assemble the individual IDs into a single key that would point to a single storage location for the entire file, as illustrated in figure 5(b). Again, some means to tolerate incomplete tag sets would need to be incorporated, e.g., by using threshold cryptography (e.g., [11]) to allow a subset of the original tags to resolve into the used address (i.e., the "secret"). While this should work well for a single file, the presence of multiple files would quickly break any used threshold cryptography algorithm, as these typically cannot differentiate between shares from multiple secrets.

4.2. Time Variance

In order to prevent that a one-time visit to a place yields eternal access to the data stored at this place, access IDs will need to periodically change. Future RFID tags might employ miniature timer components, which could be powered by a capacitive element that would be charged when the tag is within a reader's field, and subsequently be able to power the on-chip clock for a certain period of time. Alternatively, tags could be programmed to change their ID upon each readout with a certain probability, yielding a similar behavior as a timer-based solution. The FragDB prototype supports both approaches in its simulator, while providing a probabilistic ID-translation table for the real hardware reader in order to simulate the second method also for actual read-only RFID tags.

RFID Tags
Storage System
Current ID
File 4
File 91

8AF62	File67	54017	-	03BCA	File315	7843A	File315, File942
F2539	File315	8DC12	File315, File4	948AA	File942	43B21	File67
B35F*	File942, File4	0439*	File952	8CC3*	File4	D132*	File4, File42, ...
04DA*	File12, File44	B3EC*	File44, File12	1231*	File44	E321*	File44
74A**	-	C34**	File91	BC3**	File91, File12	DE2**	File91, File12
9DB**	File91	324**	-	987**	File14, File74	9E4**	File74
89***	File14, File15, ...	AC***	File14, File74	AA***	File4856	12***	File14
C1***	File4856	9C***	File44265	84***	-	4E***	File32

Figure 6. Virtual Layered Storage.

4.3. Time Continuity

While time variance ensures that a once acquired fingerprint will not guarantee perpetual access to stored data, it also cuts off access to previously stored data for "legitimate" users, i.e., those who actually visit the prior storage location. In order to still allow local access to old data, tag IDs are not simply exchanged with a new one upon an ID change, but queued. Thus, even if a new ID is in place (which will subsequently be used to store new data to support our *time variance* principle), old IDs will still be available in a tag's "lower levels," providing *time continuity* for readout.

As old IDs must be stored directly on the RFID, they will need to expire eventually, mimicking the real-world "expiration" of memories. We implemented a *gradual expiration* mechanism by shortening old IDs in the queue bit by bit as they get older. Thus, an ID at level S has 2^{S-1} bits missing, yielding $2^S - 1$ possible IDs that a reader needs to explore in order to find the correct ID that was used S timesteps before. By adjusting the "shrinkage factor," i.e., the amount of bit shortening per level, and the frequency of ID changes, e.g., each 100 readouts, the difficulty of retrieving old information at a place can be regulated, thus providing both *time continuity* and, eventually, forgetfulness.

Figure 6 gives a virtual view of a particular location, comprised of four RFID-tags shown on top. Below, each tag's storage cells are given, together with the respective contents of each cell. The IDs are stored in the ID-queue of each tag, gradually shortening the IDs as they grow older, as indicated by the starred-out numbers. To read a file, a FragDB client will need to search through such old memory cells, trying a large number of potential cell locations until a complete set of file fragments can be found.

4.4. Secure Storage

As pointed out above, FragDB does not actually store data in a particular real-world location, it only

Figure 7. Storage Keys Derivation.

requires knowledge about a certain key that is made up by this location to retrieve the information that was stored there (using this key). The actual file data can reside in any type of storage system – either a remotely accessible file server or even a global peer-to-peer storage repository. Each tag ID that is used during file storage provides a single storage address in this space, allowing our system to store one fragment of the file there, as shown in figure 5(a) previously. However, in order to facilitate file reassembly later, we need to store metadata in each such fragment, e.g., the creator of the file, the date it was stored, or the filename, but most importantly the order of the fragments and information on any employed error correction mechanism. Storing such information in plain text could make it trivial to access such data without the need to read out any tag IDs, as the storage system could be systematically scanned for matching fragments.

A straightforward solution is thus the encryption of each fragment. As we do not want to require any additional passwords or keys in the system, we simply use the tag ID (more specifically: the hashed tag ID) as an encryption key for each storage cell payload (i.e., the file fragment including its metadata). However, as we also used the tag ID to determine the storage cell where we store each fragment, we would allow an attacker to compute this encryption key trivially from the storage cell ID. Thus, we cannot use the tag ID directly, but instead hash the hashed ID again in order to derive the storage address for a fragment (see figure 7).

Figure 8 shows the contents of a single storage cell, corresponding to an RFID-tag with the current address "ID". Finding this fragment in memory does not allow an attacker to decrypt it, as this requires finding the inverse of a hash operation. If the ID is known, however, computing the memory cell location and its encryption key becomes trivial. Obviously, an attacker could simply guess an ID and retrieve the data found at this particular storage cell. By using sufficiently large IDs – 104 bits in the case of the μ-chips – such an exhaustive search of all 2^{104} memory cells is rendered impractical.

Note that each storage cell can contain multiple file fragments – these all share the same key and can be differentiated by their (decrypted) metadata. A FragDB client reading, say, ten different RFID-tags, would immediately be able to access ten different storage locations containing zero, one, or more fragments each. By

Address	Contents
hash(hash(ID))	Enc(hash(ID), Fragment Payload)

Figure 8. Storage Cell Contents.

using the corresponding key for each cell, these fragments can be decrypted and sorted into different files (as seen in figure 3 above). The next section analyzes the complexity of this process.

5. Complexity Analysis

In order to better assess the feasibility of our proposed architecture, we have analyzed the complexity of the envisioned time variance and continuity principles. These figures should give a better idea on how hard it will be to retrieve old data through expired keys. The following calculations make a number of assumptions:

- The time to read out a tag is constant for all tags
- When shortening a (past) ID, all possible full IDs are equally probable
- The time to read out a fragment is constant for all storage addresses
- File retrieval operates on the correct tag set, i.e., no tags are missing, but the stored file may require the use of old IDs

As the actual reading of tag IDs can be assumed to be linear, i.e., $O(n)$, any complexity stems from the potentially large number of storage addresses that need to be checked if the file contents cannot be reassembled from the set of initial (i.e., current) IDs. In the best case, each tag ID leads directly to the corresponding data fragment, resulting in a total complexity of O(2n). However, the fact that tags periodically change their ID (either time-based or probabilistic after a number of readouts), and that these past IDs are potentially shortened over time (ID-Fading), typically results in a much larger complexity. In the following, we compute the worst case complexity for both time-based and usage-based ID-updates.

5.1. Time-Based ID-Updates

If tags change their ID in a time-based fashion, one would simply take one arbitrary tag from the tag-pool, locate the desired fragment in its list of past IDs, and once this is found, query the same ID-position in all other tags. Assuming that the initial tag IDs for storage were read together (i.e., in one sweep of a reader),

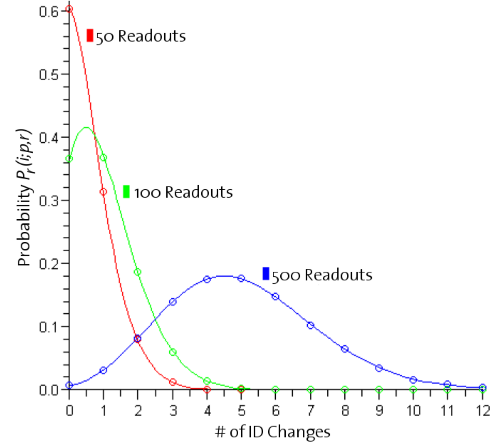


Figure 9. Distribution P_r of ID-Changes.

and that the period for switching IDs is long compared to the time it takes to read a tag, all other file fragments should be stored either at this initial ID-level, or in a neighboring level. In the worst case, the total number of reads R required to find all n fragments where the first fragment is located on the s-th level would be $O(s+2n)$, as indicated by equation 1:

$$R \leq s + 3 + 2(n-2) \approx O(s+2n) \tag{1}$$

If ID-fading is in effect, i.e., if older IDs are gradually shortened, e.g., one bit per level, the reading on lower levels becomes significantly more complex. In order to find the first file fragment at level s, a user would need to try $1 + 2^1 + 2^2 + 2^3 + \ldots + 2^{s-1} = 2^s - 1$ different possible IDs in the worst case. Afterwards, finding the second fragment would also require costly ID expansion: in the worst case, first 2^{s-2} for the layer above s, then 2^s for the layer below (cf. figure 6). Only then would the two possible layers be identified and the remaining fragments could again be found in $2(n-2)$ steps. The overall complexity thus reaches $O(2^{s+1} + 2n)$, as indicated by equation 2.

$$R \leq (2^s - 1) + (1 + 2^{s-2} + 2s) + 2(n-2) \approx O(2^{s+1} + 2n) \tag{2}$$

5.2. Usage-Based ID-Updates

With usage-based ID-updates, tags IDs are updated with a small probability p upon every read (cf. time variance in section 4). This means that often-used tags will experience a large number of ID changes, while seldom readout tags will only have few IDs changed in the same period of time. This makes our above computations more complex, as we cannot assume that

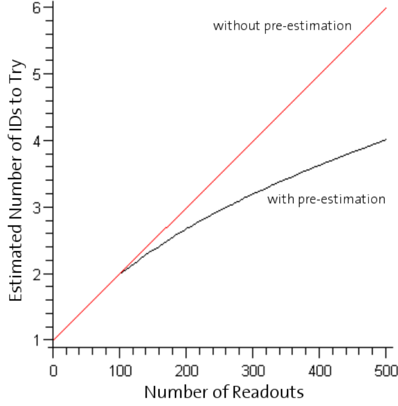


Figure 10. Payoff from Guessing.

all of our fragments are stored within a few consecutive layers. In theory, we might need to try all possible layers, with s being the "deepest" layer, thus reaching a worst case complexity of $O(sn)$ with constant IDs, and $O(n^s)$ with ID-fading:

$$R \leq (2^s - 1)n \approx O(n^s) \quad (3)$$

We can significantly lower this number if we can come up with reasonable guesses as to the correct level s a fragment is to be found. Knowing the probability p of an ID-change, we can compute the probability that a particular tag changed its ID i times, given a number of r readouts. This can be determined by using the binomial distribution:

$$P_r(i; p, r) = \binom{r}{i} p^i (1-p)^{r-i} \quad (4)$$

Plotting the above equation with a value of p=1% for 50, 100, and 500 reads, yields figure 9. As the number of readouts increases, the probability for the most likely number of ID changes diminishes, resulting in a flatter curve that makes it more difficult to correctly "guess" the right number of changes. Being able to estimate the number of ID-changes helps us to reduce the number of past tag IDs that we need to try out before finding a matching fragment for the file we are looking for. Figure 10 shows the expected payoff of this approach, using p=1% and no ID-fading.

The resulting read-access numbers for a 100-tags-sized file are plotted without and with ID-fading in Figure 11(a) and Figure 11(b), respectively. Note that Figure 11(b) has a y-scale of 10^7, so the two plots should not be visually compared. Plots are given for the c = 30%, 50%, and 80% quantile, indicating the number of reads required if only a subset of the fragments is needed for reassembly (e.g., the employed *FEC 2:1*

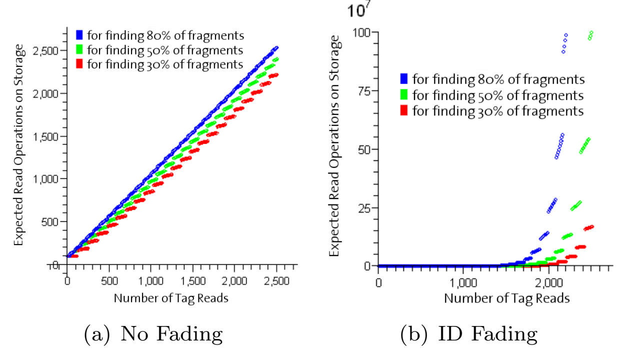


Figure 11. Max. Reads Req. for 100 Tags.

fragmentation algorithm only requires 50% of all fragments). The quantiles are computed based on equation 5 below, the estimated access numbers follow equations 6 and 7, without and with ID-fading, respectively. Notice how usage-based ID changes and fading make retrieving old data almost impossible, requiring millions of readouts to assemble a 100-tags file.

$$c\text{-}quant = \left\{ x \in \mathsf{N} \mid \sum_{i=0}^{x} P_r \geq c \wedge \sum_{i=x}^{r} P_r \geq r - c \right\} \quad (5)$$

$$E_{nofade} = \sum_{i=0}^{c} n P_r(i; p, r) A(i) \quad \text{with } A(i) = i + 1 \quad (6)$$

$$E_{fad} = \sum_{i=0}^{c} n P_r(i; p, r) A(i) \quad \text{with } A(i) = 2^{i-1} + \frac{1}{2} \quad (7)$$

5.3. Tag Selection

The above computations assume that the set of tags for storage and later retrieval is identical. However, if the reader is not directly positioned in the right spot during retrieval, it will not only be necessary to find the right level of where to search for the fragments, but also the right place (i.e., the right subset of tags). Three possible cases of suboptimal tag selection during readout are possible:

1. The found tag-set partially overlaps the original set of tags for storage
2. The found tag-set is (only) a subset of the original set of tags
3. The found tag-set does not contain any tag used originally for storage

Obviously, the system cannot control where the user points the tag reader. However, it can visually (and potentially using audio feedback) inform the user of the progress of file reassembly, thus providing guidance to the right direction in which to continue searching (cf. figure 3 on page 3 above).

More importantly, the system can infer neighborhood relationships based on the tag-stream, i.e., the order in which tag IDs are being fed into the system from the reader. This might allow the system to employ smart exploration algorithms that start from a known tag and corresponding ID-level and search both neighboring tags and neighboring ID-levels for more file fragments.

6. Conclusions

We have designed and built a system for localized, secure storage, based on SDRI. The initial prototype vividly demonstrates the potential of this application, and allows us to explore the uses and limits of this principle.

Our initial analysis confirms both the feasibility of the approach, as well as its resistance to trivial exploration attacks. Information that has been stored at a particular place can only be retrieved by either saving the tag IDs that have been used, or by revisiting the storage place. In the latter case, ID-updates and ID-fading makes retrieval incrementally harder, eventually rendering very old information inaccessible by all but those who retained the access IDs.

A number of improvements are possible from this initial prototype. The storage system is currently only an in-memory hash table. Using some freely available P2P-frameworks, a corresponding distributed version of the memory could be devised. This would also need to address the problem of eventually deleting memory locations whose IDs have faded away for good, e.g., using common caching strategies such as LRU (last recently used) or NRU (not recently used). Also, fragment reassembly could further be improved by incorporating clever ID-space exploration strategies and adding corresponding user interface mechanisms to provide position guidance. Last not least, some mobile reader device could be devised in order to create sample applications, e.g., for memory prostheses or distributed vehicular information systems.

7. Acknowledgements

Lukas Stucki was instrumental in implementing and analyzing the FragDB prototype as part of his Master's thesis. Ruedi Arnold provided helpful comments on earlier drafts of this paper. This work has been partially funded by Hitachi Systems Development Laboratories (SDL), Japan, who also provided the μ-chip RFID equipment.

References

[1] G. D. Abowd. Classroom 2000: An experiment with the instumentation of a living educational envrionment. *IBM Systems Journal*, 38(4):508–530, Oct. 1999.

[2] J. Bohn and F. Mattern. Super-distributed RFID tag infrastructures. In *Ambient Intelligence – Second European Symposium, EUSAI 2004, Eindhoven, The Netherlands, Nov. 8–11, 2004, Proceedings*, volume 3295 of *Lecture Notes in Computer Science*, pages 1–12, Berlin Heidelberg New York, Nov. 2004. Springer.

[3] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. A. Shafer. Easyliving: Technologies for intelligent environments. In P. J. Thomas and H.-W. Gellersen, editors, *HUC*, volume 1927 of *Lecture Notes in Computer Science*, pages 12–29. Springer, 2000.

[4] W. Geyer, H. Richter, L. Fuchs, T. Frauenhofer, S. Daijavad, and S. Poltrock. A team collaboration space supporting capture and access of virtual meetings. In C. S. Ellis and I. Zigurs, editors, *GROUP '01: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pages 188–196, New York, NY, USA, 2001. ACM Press.

[5] J. I. Hong and J. A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *MobiSYS '04: Proceedings of the 2nd international conference on mobile systems, applications, and services*, pages 177–189. ACM Press, 2004.

[6] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *Pervasive Computing*, 1(2):67–74, Apr. 2002.

[7] T. Kindberg, K. Zhang, and N. Shankar. Context authentication using constrained channels. In *Mobile Computing Systems and Applications, 2002. Proceedings of the Fourth IEEE Workshop*, pages 14–21. IEEE Press, 2002.

[8] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure to enable active spaces. *Pervasive Computing*, 1(4):74–83, Oct. 2002.

[9] G. Sampemane, P. Naldurg, and R. H. Campbell. Access control for active spaces. In *Proc. of the 18th Annual Computer Security Applications Conference (ACSAC'02)*, pages 343–352. IEEE Press, Dec. 2002.

[10] N. Sastry, U. Shankar, and D. Wagner. Secure verification of location claims. In *WiSe '03: Proceedings of the 2003 ACM workshop on Wireless security*, pages 1–10, New York, NY, USA, 2003. ACM Press.

[11] A. Shamir. How to share a secret. *Comm. of the ACM*, 22(11):612–613, 1979.