Journal of Location Based Services

Vol. 2, No. X, Summer 2008, 000–000

Secure localized storage based on super-distributed RFID-tag infrastructures

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

Abstract: Humans are social creatures that often take pleasure in sharing ideas and opinions with others. While the Internet has made this process much easier (Wikipedia, Flickr, Facebook), it has also made the assembly of such shared information into central profiles with the help of a simple online search dangerously trivial. Access control could of course be used to prevent this, yet this would also limit the sharing of such information to a much smaller set of well-known users, as well as introducing a significant administrative overhead. 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 the knowledge of a particular password or certificate. Data shared using FragDB is much harder to assemble into a profile, as a person's activities across space and time are not centrally assembled, but require actual physical presence for querying. FragDB uses the IDs of tiny RFID tags embedded in floors, walls, or doors, to compute a local key that is used to encrypt and decrypt data in a global storage system. It is particularly suited to disseminate information pertaining to a particular location or a local group. We describe the requirements and implementation of such a system, and analyze its complexity.

Keywords: Access control, privacy, secure storage, localized storage, RFID, smart environments

1 Introduction

The system presented in this work – FragDB – protects stored digital information by binding initial access to it to a specific place: If one wants to read the data for the first time, one has to be (roughly) in the same spot where it was initially stored. By virtue of being in the same location, previously stored information can be discovered, without having to exchange passwords or define access control policies. Its main goal is to simplify access control to locally relevant information, by requiring no special localization technology, nor introducing policy management overhead.

Why would such a system be useful? After all, without passwords or other user authentication mechanisms in place, *anybody* would have access to data stored in a particular location, simply by being there! On the other hand: if I have information that I do not mind sharing, why not put it up on the Web, so that everybody can find it?

The latter approach – publishing personal data, preferences, and opinions on the Web – has in fact become popular. The tremendous rise of social networks and content sharing platforms (e.g., Facebook, Flickr, or YouTube) in recent years bears witness to the fact that humans are social creatures, who take pleasure in documenting their lives for others to see and comment upon. However, many users of such services do not seem to be aware that the centralized storage of their holiday pictures or favorite music facilitates the assembly of detailed personal profiles, all with the help of a simple online search. A recent survey of over 600 British companies revealed that one in five had used online searches in social networking sites such as Facebook to verify potential employees (Foster 2007), with a significant minority of all users not even being aware of the visibility of their Facebook profile (Gross and Acquisti 2006).

While many such sites have begun to introduce better access controls, their uptake has been slow, as this directly limits the ability to “meet” and interact with new people. If passwords or user-based access controls are used, those wishing to share data with others have to either repeatedly send out passwords to interested parties (as used to limit access to, e.g., lectures slides on the class homepage, or one's holiday pictures on the personal Web site) or manually manage lists of friends and family (e.g., on Flickr).

A similar challenge awaits so-called “capture and access” deployments, as exemplified in the Classroom2000/ eClass (Abowd 1999) or Teamspace (Geyer, et al. 2001) projects, which employed audio and video recording to make lectures and meetings accessible to attendees and external guests for later perusal. Progress in image and video search capabilities might soon make image-based queries feasible, thus allowing targeted searches for participants (and their behavior) of such recorded events. Again, access control is cumbersome, especially if not only the participants themselves should get access, but also the students in next year’s class, or those from last year who retake the exam, or even students who are only *thinking* of taking the class.

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* (Hong and Landay 2004, Sastry, Shankar and Wagner 2003, Sampemane, Naldurg and Campbell 2002, Kindberg, Zhang and Shankar 2002, Brands and Chaum 1994). Using location based access control, e.g., students could publish their latest party pictures only “on campus”, thus making it more costly for potential employers to dig into their past, especially for jobs in different cities or even countries. Similarly, recorded meetings could only be accessible within the particular room or building they were recorded in, thus limiting world-wide exposure without preventing local students or employees from accessing such records.

Common to current location-based access control systems, however, is the need for *explicit* access control, i.e., data owners will need to formulate and adjust security policies, based on a verified location claim, 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 everyday situations might not warrant the creation and configuration of elaborate access control policies. In such 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 (or "physical") access control, only those who are 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 (Kriplean, et al. 2007).

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 data that one does *want* to share with others, but which nevertheless should not be globally available. An example would be the above-mentioned classroom, where the individual participants could store logs of their own wearable sensors directly in the lecture hall, allowing participants who come late, or maybe even next year's students, to easily find it there, yet preventing someone from half-way around the globe from monitoring their performance in class.[1] While traditional location-based access control methods (Gonzales-Tablas, et al. 2005) could offer similar functionality, the setup for both users and system administrators would be considerably higher, as locations and their access parameters would need to be explicitly defined.

[1] Note that a centralized system that simply deletes class data after a prescribed period would work very differently: It would still allow anybody to access the data up to that point, while FragDB requires students to actually attend the same classroom (or, e.g., visit the professor's office).

Another example might be smart vehicles, which 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, since local data would "stay" local.[2] A final example could be a coffee shop that would offer electronic bulletin boards that are only accessible to patrons visiting the store, where not only classifieds but also the music their patrons listen to or the times they spend there could be stored.

Obviously, using an explicit access control-system for such data will always allow more fine-grained control over who gets to access what data at what time (or from where). However, studies have shown that the actual disclosure behavior of users often significantly differs from their initially stated preferences (Berendt, Günther and Spiekermann 2005), which indicates that users of a capture and access application would need to repeatedly update their original disclosure specifications. Olson, Grudin, and Horvitz found that one of the key requirements for users when formulating access policies was the identity of the recipient, indicating that defining such controls for yet unknown recipients would result in overly restrictive policies (Olson, Grudin and Horvitz 2005). Others have pointed out that the process of devising access control rules is inherently difficult for end-users (Adams and Sasse 1999, Cao and Iverson 2006). The motivation for an implicit privacy control system is thus to encourage the sharing of more or less innocuous information – data that one does not

[2] While one could imagine simply using anonymous messages sent to a central system to achieve the same effect, this would require some form of location proof in order to prevent an attacker to create fake hazards, thus greatly increasing the complexity and costs of such a system. By using a local storage scheme, an attacker could only attack one stretch of road at a time, considerably weakening this type of attack.

mind sharing or actually *wants* to share with others, yet which one nevertheless might not (or should not) want to be available from anywhere in the world, nor for the data to be centrally collected and stored (which would facilitate profiling).

An implicit privacy control system attempts to limit access to such data without denying it completely, e.g., it prevents others from remotely observing our moves and actions, yet it makes our data easily available to people we share the same rooms, coffee shops, or roads with, and it does so with very little administrative overhead on behalf of the user. The system described in this article – FragDB – attempts to demonstrate the technical feasibility of such an implicit privacy control system, at much lower costs (both for use and setup) as traditional location-based access control systems.

2 Basic Principles

One can liken the basic idea of FragDB to how people manage their privacy in the real (i.e., not virtual) world: While many of our daily actions are visible to our neighbors, fellow shoppers, and co-workers, detailed information about such activities is typically not available in faraway places.[3] In order to find out about the details of someone's daily life, one would need to travel to a person's home town and talk to friends and neighbors. Thus, much of our privacy is still an inherent aspect of the locality of our person. Instead of having to *manage* one's privacy, which always entails the possibility of mismanaging it, the limited communication and storage capabilities of our fellow humans implicitly hampers the unwanted disclosure of personal facts across spatial and temporal boundaries (Marx 2001).

[3] This of course excludes online activities such as Web surfing in an Internet café or using our credit card to make purchases.

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.

Fingerprinting locations is a popular technique for self-positioning using radio signals (LaMarca, et al. 2005) or RFID tags (J. Bohn 2006). In contrast to such techniques, we do not attempt to provide explicit location information to either the user or an external observer. "Fingerprints" in our system have no particular location information associated with them, nor would such a fixed association be advantageous given our requirements outlined below (cf. the *time variance* principle).

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 exactly 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., the 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 (i.e., also in the future), the access keys of a place have to periodically change.

Challenge two immediately leads to another issue: 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 (or, of course, made a local copy of the data, e.g., on our personal system). Any *new* visitor to a place would not be able to find any previously stored information if the location's fingerprint was just changed. However, the idea of conveniently sharing 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 *past* data stored at this place (for future visitors). However, one might want old keys to *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. This allows, e.g., students to access the lecture notes for last week's class, or even for last year's class. Also note that "for a while" is an application-dependant concept: In the case of the classroom, one might want to have student questions and discussions around for several semesters, while in the example of the road hazard, a few hours or days might be sufficient.

It is important to point out that the principles of time continuity and time variance only apply to *novel* access to stored information, i.e., from visitors that have not yet accessed this information. Obviously, once data has been accessed, one is free to copy it to local storage and access it at any time in the future, or even pass it on to others or publishing it, e.g., on the Internet. With time variance, we ensure that having the fingerprint of a particular location does not entitle one to continuously access all data that is ever stored there, while time continuity allows one to find stored data at a particular location even if the storage happened sometime in the past. Neither principle applies to the subsequent *dissemination* of the collected information, which would require some sort of digital rights management (DRM), combined with a trusted computing platform.

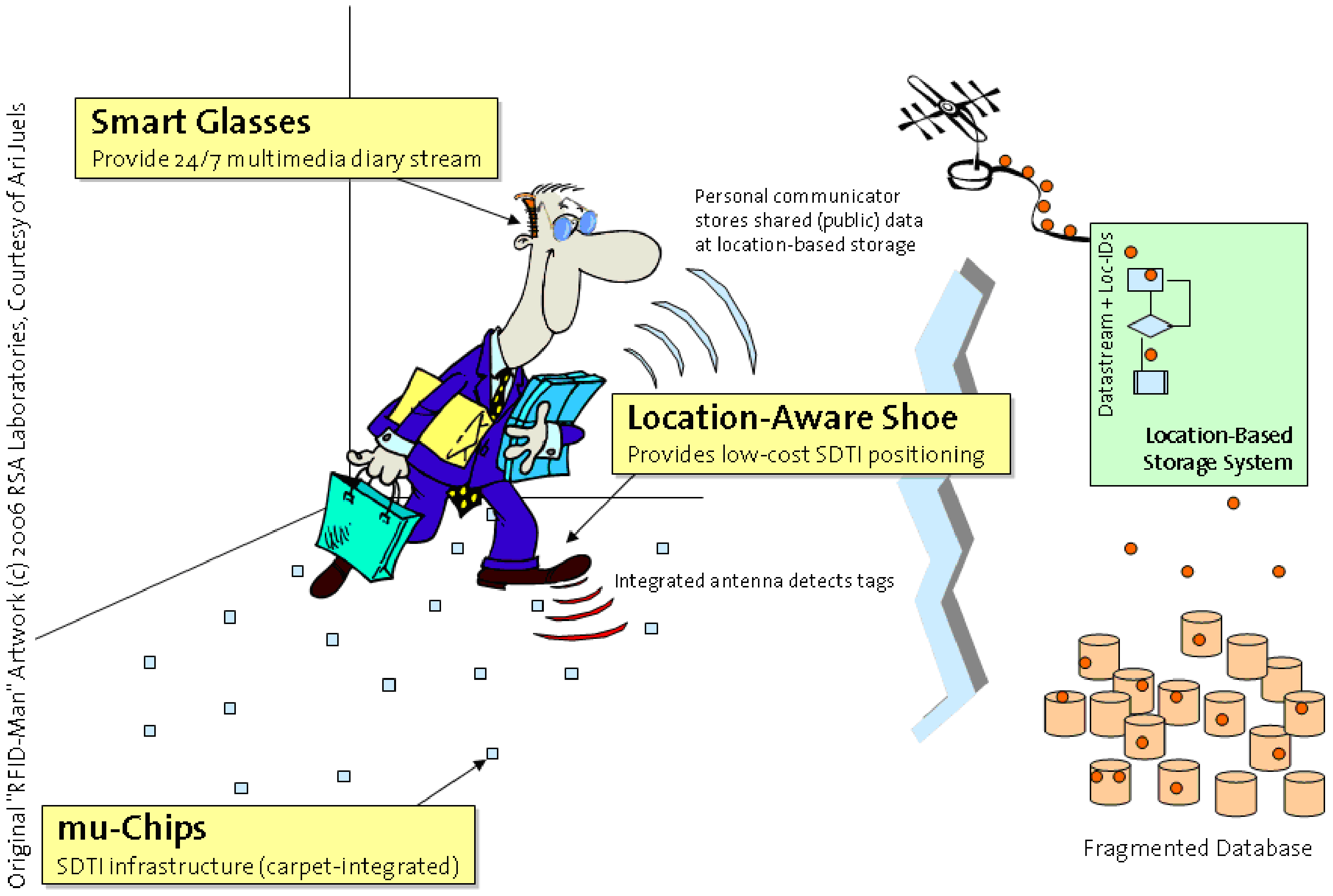
Alternatively, data owners could delete their data from the storage system, instead of relying on time variance and time continuity to eventually expire access to it. However, as pointed out in the introduction, we want to reduce setup and management as much as possible, so time continuity and time variance can provide sensible defaults. If needed, manual deletion could still be applied (e.g., if a classified ad in the coffee shop expires).

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 so that database attacks are infeasible.

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 (Bohn and Mattern 2004), 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.

Note that our system does not require any form of security on the tags – it would work equally well with barcodes or even numbers written on a wall (though these two approaches would not support the time variance principle). All we need is a random number that is only available at a particular location, and preferably the ability to update that number occasionally (see section 6 for a discussion of security aspects).


Original "RFID-Man" Artwork (c) 2006 RSA Laboratories, Courtesy of Ari Juels
Smart Glasses
Provide 24/7 multimedia diary stream
Personal communicator stores shared (public) data at location-based storage
Datastream + Loc-IDs
Location-Based Storage System
Location-Aware Shoe
Provides low-cost SDTI positioning
Integrated antenna detects tags
mu-Chips
SDTI infrastructure (carpet-integrated)
Fragmented Database

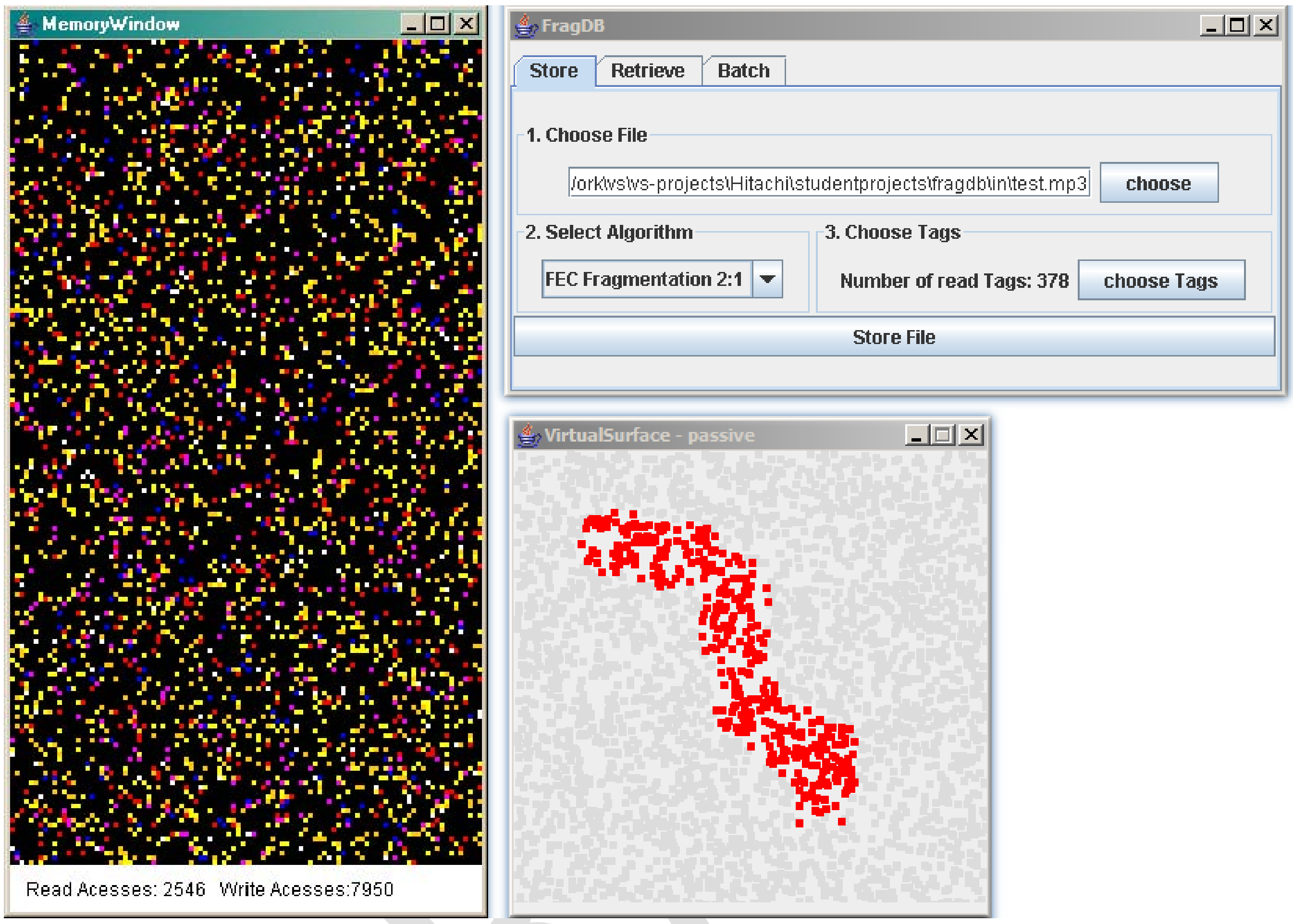
- Figure 1: Overview of the general operation. The IDs of tiny RFID-chips form a location fingerprint, which is used to securely store information in a global, distributed database. In order to retrieve and decrypt the stored data, the set of original tag-IDs must be approximated.

Error! Reference source not found. 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. In contrast to work explicitly addressing secure verifiable location claims, e.g., (Sastry, Shankar and Wagner 2003), using RFID significantly lowers deployment costs, as no specific "secure" localization hardware is required.

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 a surface 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). The simulator allows us to test data storage and retrieval algorithms, while the hardware interface serves as a demonstrator. 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, such as Oceanstore (Rhea, et al. 2003) or Freenet (Clarke, et al. 2002).



1. Choose File

/ork\ws\ws-projects\Hitachi\studentprojects\fragdb\in\test.mp3 choose

2. Select Algorithm

FEC Fragmentation 2:1

3. Choose Tags

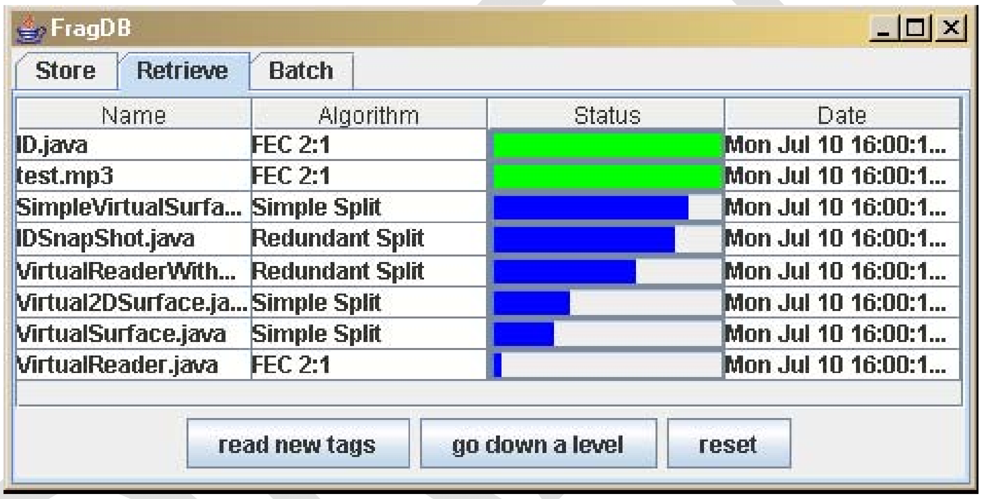
Number of read Tags: 378 choose Tags

Store File

Read Acesses: 2546 Write Acesses:7950

- Figure 2: Prototype interface. The FragDB Prototype is used for both simulating large tag populations (using a simulated surface with embedded tags as shown in the window "virtual surface"), as well as directly controlling a physical reader device. A "memory window" shows read and write access to a virtual storage system.

Error! Reference source not found. shows the user interface of the prototype after storing a file in the virtual environment. Centered at the bottom, the window labeled "virtual surface" shows a simulated surface with embedded tags, some of which 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, mimicking the limited read range of an antenna. To simulate a larger read range or a continuous reading while moving around, one can click multiple times and subsequently enlarge the reading area. 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, making 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.

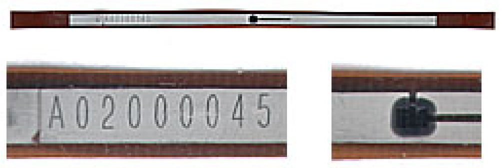


Name	Algorithm	Status	Date
ID.java	FEC 2:1	🟩	Mon Jul 10 16:00:1...
test.mp3	FEC 2:1	🟩	Mon Jul 10 16:00:1...
SimpleVirtualSurfa...	Simple Split	🟦	Mon Jul 10 16:00:1...
IDSnapShot.java	Redundant Split	🟦	Mon Jul 10 16:00:1...
VirtualReaderWith...	Redundant Split	🟦	Mon Jul 10 16:00:1...
Virtual2DSurface.ja...	Simple Split	🟦	Mon Jul 10 16:00:1...
VirtualSurface.java	Simple Split	🟦	Mon Jul 10 16:00:1...
VirtualReader.java	FEC 2:1	🟦	Mon Jul 10 16:00:1...

- Figure 3: Key reassembly/data access. The system first decrypts each fragment using the derived encryption key, then sorts them according to their meta-data. Completely reassembled files have a green, fully filled status bar, while partially restored files show a correspondingly shortened blue bar.

The set of read tags (shaded in the "virtual surface" window in **Error! Reference source not found.**) 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 **Error!**

Reference source not found.. As during file storage, the user first uses a paintbrush-like cursor to select a set of tags from the simulated 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, as indicated by the status column. Section 4 below describes the mechanics of this process.



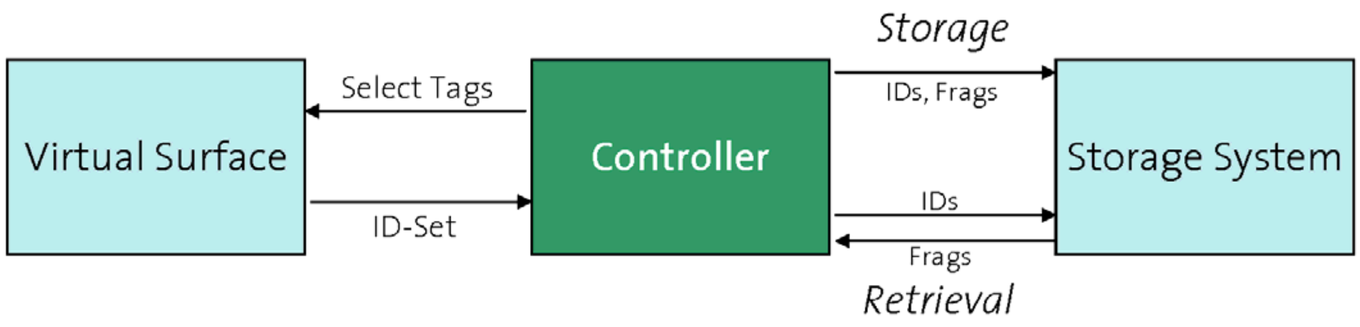
- Figure 4: Hitachi mu-chip with antenna.

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. Hitachi's first generation of µ-chips feature a size of 0.16mm^2 and a stick antenna of about 10cm (see **Error! Reference source not found.** for a close-up view).

They contain a factory-written, read-only 104 bit serial number, which can be read out from up to 5-10cm distance. The µ-chips and µ-chip-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 (this of course only works for our prototype – a real FragDB deployment requires multi-ID tags that can both change their own ID and store old IDs in order to support our principle of time variance).

4 Architecture

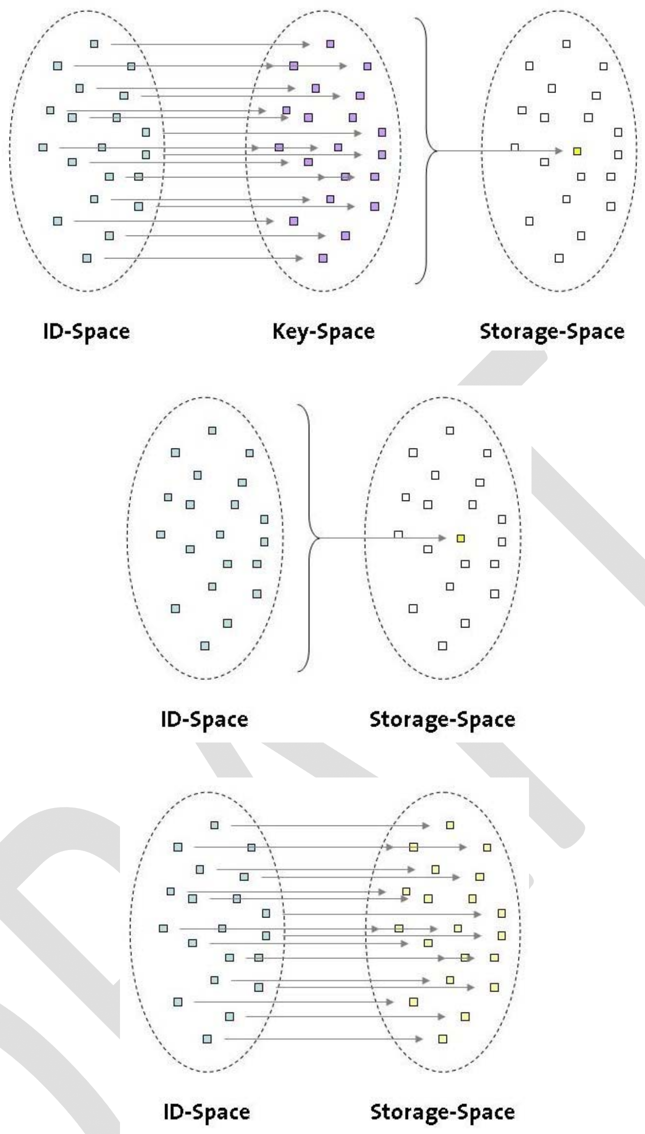
Error! Reference source not found. shows the general architectural division. A central *controller* interfaces the simulated surface (or, alternatively, a real hardware reader) to receive a set of tags read at a particular location. It then uses these tag IDs to either store data in the storage system, 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.



- Figure 5: FragDB Controller. The controller receives a set of tag-IDs from the simulated surface ("virtual surface"). It uses this to either store a fragmented file in the storage system ("storage") or to retrieve a set of corresponding fragments from it ("retrieval").

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 **Error! Reference source not found.**(bottom). 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, 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. Figure 3 shows the use of the three different fragmentation algorithms for different files.



- Figure 6: ID-storage mapping options. In the FragDB prototype, each ID points to a data share, fragmented within the database and only accessible through the individual IDs (bottom). One could, however, also envision a set of tags pointing to a single memory location, e.g., identified by a single URL (middle), or the fragmented data itself allowing the reassembly of such a URL (top).

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 **Error! Reference source not found.**(middle). Again, some means to tolerate incomplete tag sets would need to be incorporated, e.g., by using threshold cryptography (Shamir

1979) 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. Although this could be mediated by an intermediate layer, as shown in **Error! Reference source not found.**(top), it would have no advantage over the simple scheme described in **Error! Reference source not found.**(bottom). FragDB currently only supports the simple fragmentation scheme, albeit with the three different fragmentation algorithms described above.

A much simpler approach might be to use the same tag-IDs on multiple RFID-tags, thus increasing the change that the reader finds a needed ID. While certainly the simplest approach, this adds considerable complexity during deployment, as tags with identical IDs must be brought out close to another, in order to provide the needed redundancy. One could reduce the total number of different IDs available to increase the chances that this happens automatically, but this would significantly lower security (as one could try to guess the IDs available in a place). Using a forward error correcting approach allows us to use any set of tags in a particular place, as all redundancy is in the FragDB layer, rather than in the tag hardware. See also section 5.3 for a discussion on finding the right set of tags during retrieval.

4.2 Time Variance

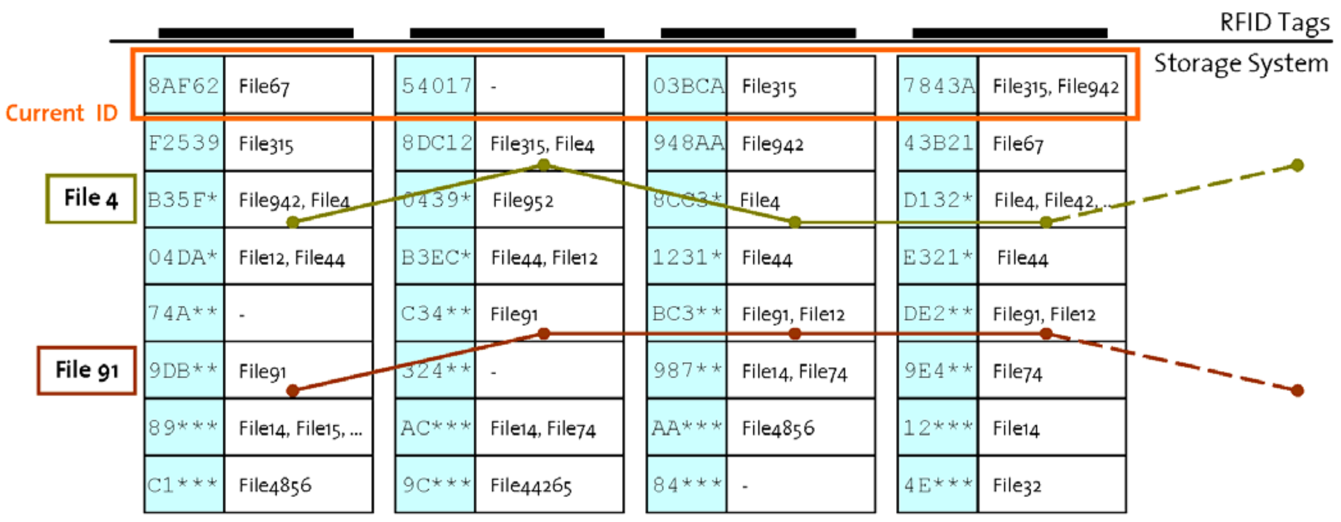
In order to prevent that a one-time visit to a place yields eternal access to the data stored at this place, access IDs will need to periodically change. Future RFID tags might employ miniature timer components, which could be powered by a capacitive element that would be charged when the tag is within 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 that simulates the second method also for actual read-only RFID tags.

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 future access to previously stored data for “legitimate” users, i.e., those who actually visit the prior storage location. Tag IDs are not simply exchanged with a new one upon an ID change, but queued, which allows local access to old data. 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 possible IDs that a reader needs to explore in order to find the correct ID that was used *S* time-steps 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.

Error! Reference source not found. 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 (see the two examples for files 91 and 4 in **Error! Reference source not found.**).

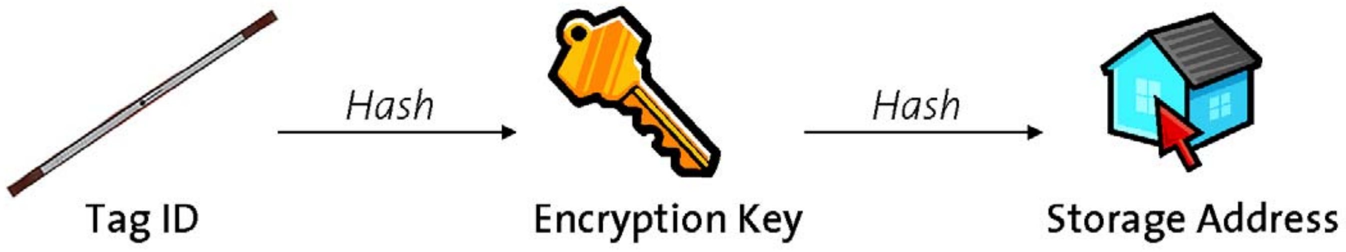


8AF62	File67	54017	-	03BCA	File315	7843A	File315, File942
F2539	File315	8DC12	File315, File4	948AA	File942	43B21	File67
B35F*	File942, File4	0439*	File952	8CC3*	File4	D132*	File4, File42,...
04DA*	File12, File44	B3EC*	File44, File12	1231*	File44	E321*	File44
74A**	-	C34**	File91	BC3**	File91, File12	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 7: Virtual layered storage. As tags create new IDs, thus pointing to new fragment storage locations, old IDs are kept in a queue, allowing for the retrieval of older files. Old IDs are shortened as they “grow old", thus making it increasingly hard to correctly determine the address of a fragment.

4.4 Secure Storage

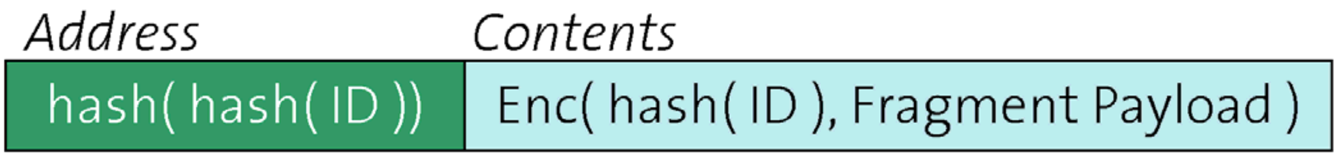
As pointed out above, FragDB does not actually store data in a particular real-world location. It only requires knowledge about a certain key (i.e., a fingerprint) that is “created” by this location to retrieve the information that was stored there (using this key). The actual file data can reside in 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 **Error! Reference source not found.**(bottom) 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.



- Figure 8: Derivation of Storage Address and Encryption Key. The tag ID is not used directly to store the data, but repeatedly hashed to derive the encryption key of the fragment data, which in turn leads to the storage address of the encrypted payload.

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 for deriving the storage address of a fragment (see **Error! Reference source not found.**).[4]

[4] While our prototype uses standard Java hash functions, we suggest the use of cryptographically secure one-way functions such as SHA-1 or SHA-256 (National Institute of Standards and Technology 2004) for deployment.



- Figure 9: Storage cell contents. The storage memory only contains encrypted information at each address, which does not allow an attacker to infer it contents, nor the location where this information was stored at. The current prototype uses SHA-1 and AES128 for hashing and encryption, respectively.

Error! Reference source not found. 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 be immediately able to access ten different storage locations containing zero, one, or more fragments each. By using the corresponding key for each cell, these fragments can be decrypted and sorted into different files (as seen in **Error! Reference source not found.** 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 performed an initial complexity analysis of the envisioned time variance and continuity algorithms. While being far from a rigorous evaluation, these figures should give an initial idea on how hard it would 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 tags 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), 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 the equation below:

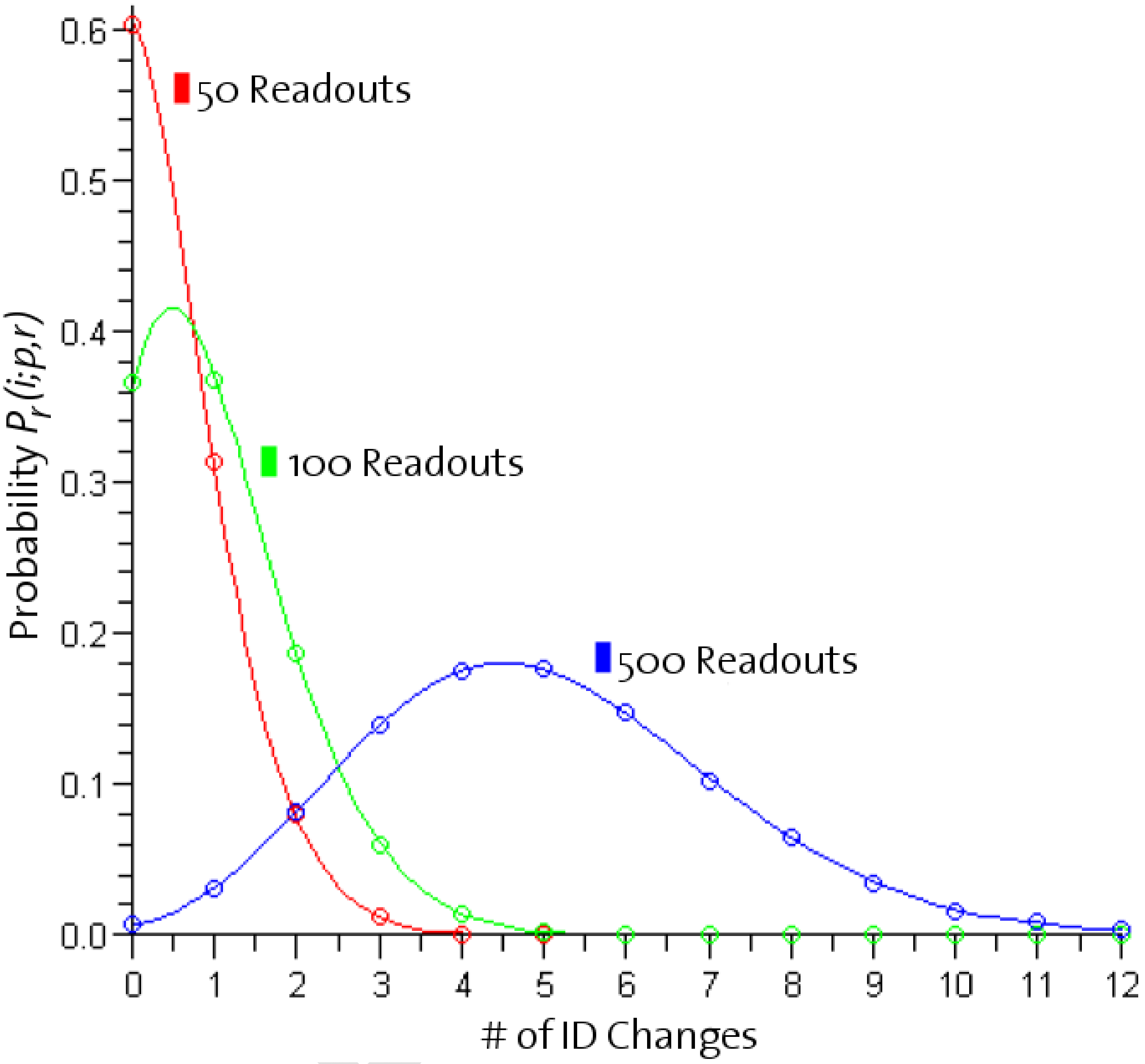
$$R \le s + 3 + 2(n-2) \approx O(s+2n)$$

- Equation 1: Maximum number of steps required to read n fragments with time-based IDs. This assumes that the first fragment is found at level s and that all other fragments are either on the same level, or one level below or above. For the first fragment, *s* steps are needed. The second fragment would need at most three reads (on the same level, one level below, and one level above). From then on, all remaining tags (*n*-2) will be located either on the first tag's level, or the second tag's level, as we assume that writing is fast enough not to incur two layer changes.

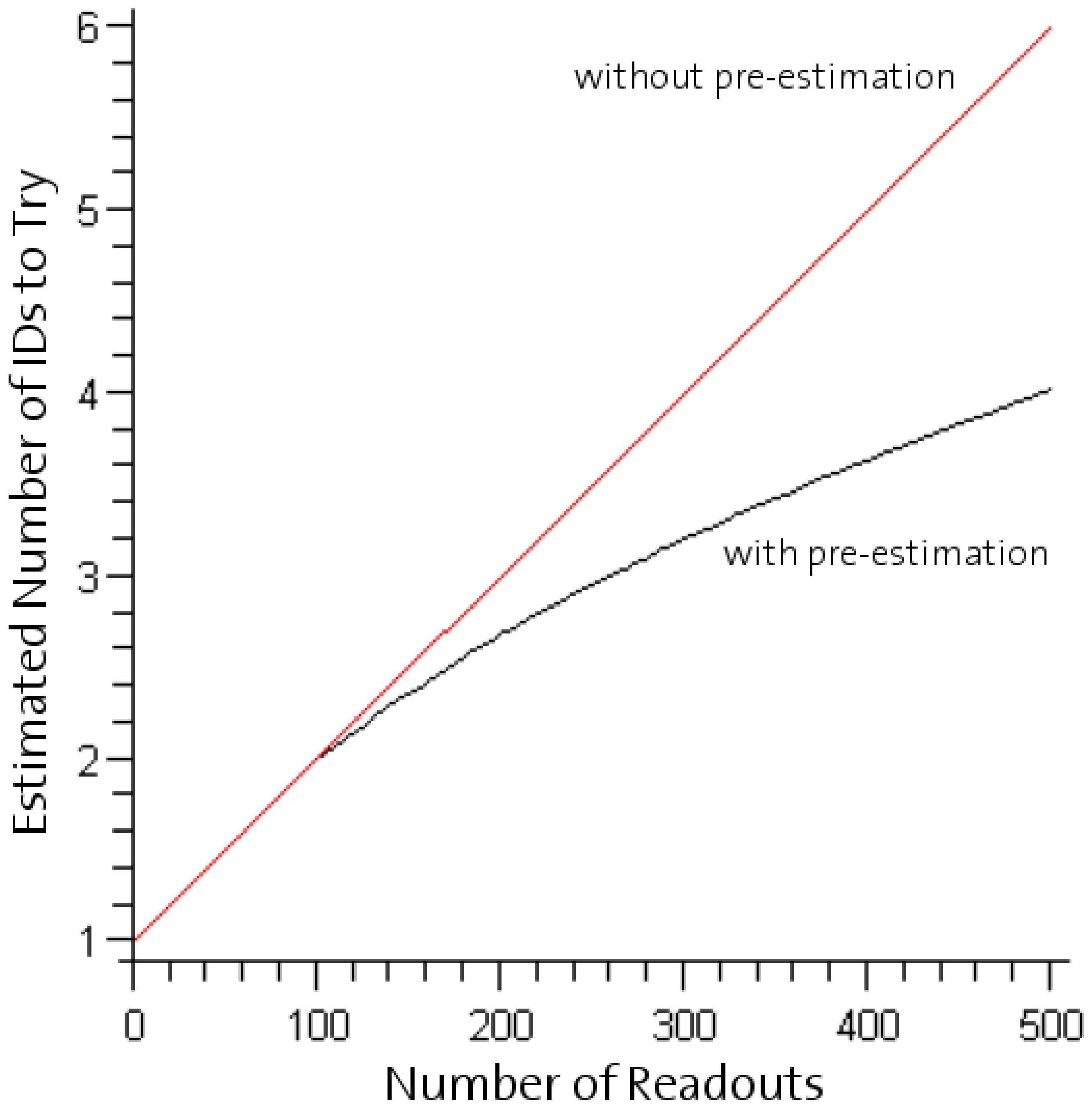
If ID-fading is in effect, i.e., if older IDs are gradually shortened, e.g., one bit per level, the reading on lower levels becomes significantly more complex. To find the first file fragment at level *s*, a user would need to try $1+2^1+2^2+2^3+\ldots+2^{s-1}=2^s-1$ different possible IDs in the worst case. Afterwards, finding the second fragment would also require costly ID expansion: after trying on the same level (2^{s-1} different possibilities), the worst case would require 2^{s-2} readouts for the layer above *s*, then 2^s for the layer below (cf. figure 7). 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 $O(2^{s+1})$, as indicated by the following equation:

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

- Equation 2: Maximum number of steps required to read n fragments with time-based ID-updates and ID-fading. This assumes that the first fragment is found at level *s* and that all other fragments are either on the same level, or one level below or above. For the first fragment, 2^s*-1* steps are needed. The second fragment would need at most three reads (on the same level, on one level below, and then on the level above), yet due to ID-fading, this would require trying out 2^{s-1}, 2^{s-2} and 2^s IDs in the worst case. From then on, all remaining tags (n-2) will be located either on the first tag's level, or the second tag's level. Note that the computations above are worst-case scenarios, and that they assume that all fragments are distributed in at most two layers. The latter does not hold if we assume usage-based ID-updates.



- Figure 10: Binomial distribution of ID-changes for an ID-change probability of p=0.01 and a number of different readouts (50, 100, 500). The more readouts are made, the flatter the curve becomes, thus making it harder to correctly guess the number of ID changes.



- Figure 11: Payoff from estimating the correct number of ID changes. Knowing how often a tag changed its ID can lower the number of IDs that need to be tried before the correct fragment can be found. The above graph uses p=1% and no ID-fading.

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 4.2). 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 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 $O(n2^s)$ with ID-fading:

$$r \leq (2^s - 1)n \approx O(n2^s)$$

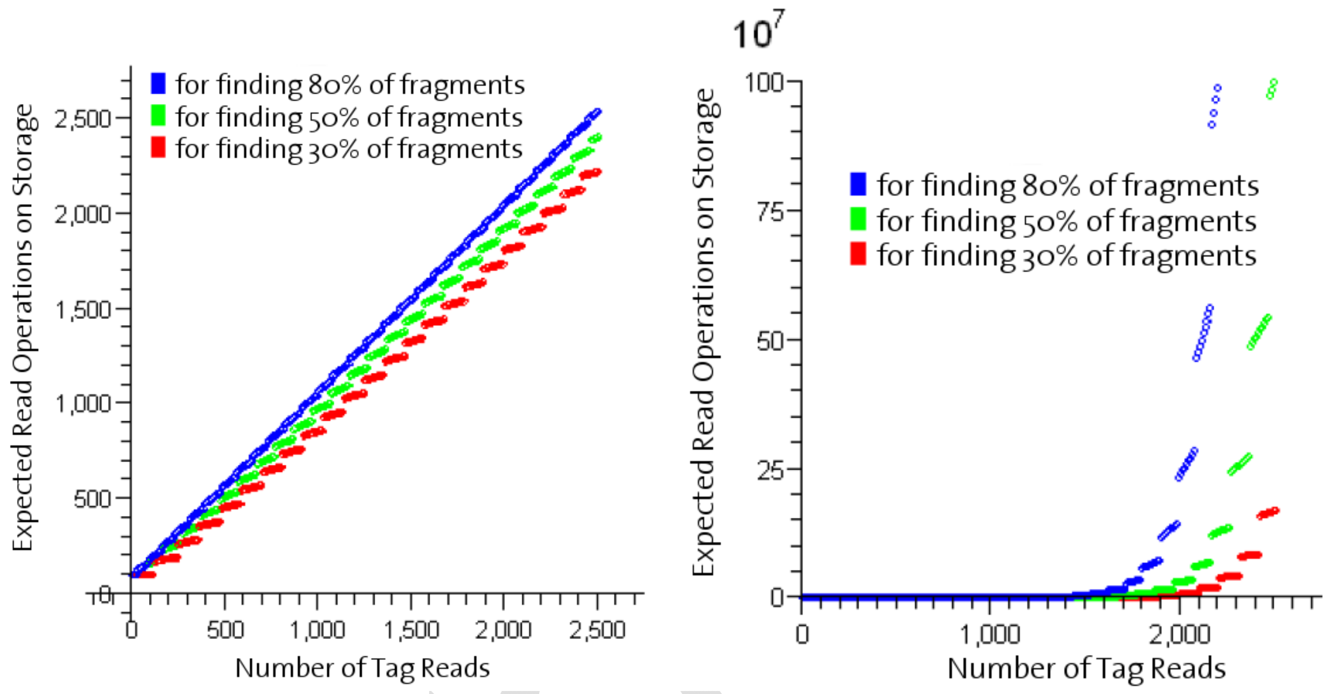
- Equation 3: Maximum number of steps required to read *n* fragments with usage-based ID-updates and ID-fading. Without any clever algorithms, finding each fragment might require us to try out all possible ID combinations up to (and including) those on level *s*

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:

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

- Equation 4: Binomial distribution of ID-changes, given a probability *p* of an ID-change per readout and *r* readouts

Plotting the above equation with a value of *p*=1% for 50, 100, and 500 reads, yields figure 10. 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 11 shows the expected payoff of this approach, using *p*=1% and no ID-fading.



(a) (b)

- Figure 12: Maximum reads requests for 100 tags. Subfigure (a) shows the number of expected read operations without ID-fading. Access to the storage system grows linearly over time – sudden jumps are due to ID-updates on the tags. Subfigure (b) shows the effect of enabling ID-fading. Retrieval quickly involves millions of storage accesses after a few thousand read operations. Note the different scale of the y-axis.

The resulting read-access numbers for a 100-tags-sized file are plotted without and with ID-fading in Figure 12(a) and Figure 12(b), respectively. Note that Figure 12(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* fragmentation algorithm only requires 50% of all fragments). The quantiles (Serfling 1980) are computed based on Equation

5 below (with P_r defined as shown in the equation above), the estimated access numbers follow Equation 6 and Equation 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.

$$c - quantile = \left\{ x \in \mathrm{N} \mid \sum_{i=0}^{x} P_r(i; p, r) \geq c \wedge \sum_{i=x}^{r} P_r(i; p, r) \geq r - c \right\}$$

- Equation 5: Computing different quantiles for reassembly. Depending on the actual fragmentation algorithm used, not all fragments are needed for reassembling a file. By computing different *quantiles* (Serfling 1980) (Serfling 1980), we can see how higher redundancy lowers the number of read accesses that are needed to fetch enough fragments from storage

$$E(n, p, r) = \sum_{i=0}^{c} n P_r(i; p, r) A(i) \qquad with\ A(i) = i + 1$$

- Equation 6: Estimated read effort without ID-fading. For reading *n* tags that have been read *r* times and have changed their ID with probability *p* after each read. The term *A(i)* denotes the number of storage memory accesses needed without ID-fading

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

- Equation 7: Estimated read effort with ID-fading. The term *A(i)* denotes the number of reads at ID-level *i* when ID-fading is active (i.e., all missing ID-bits need to be tried)

5.3 Tag Selection

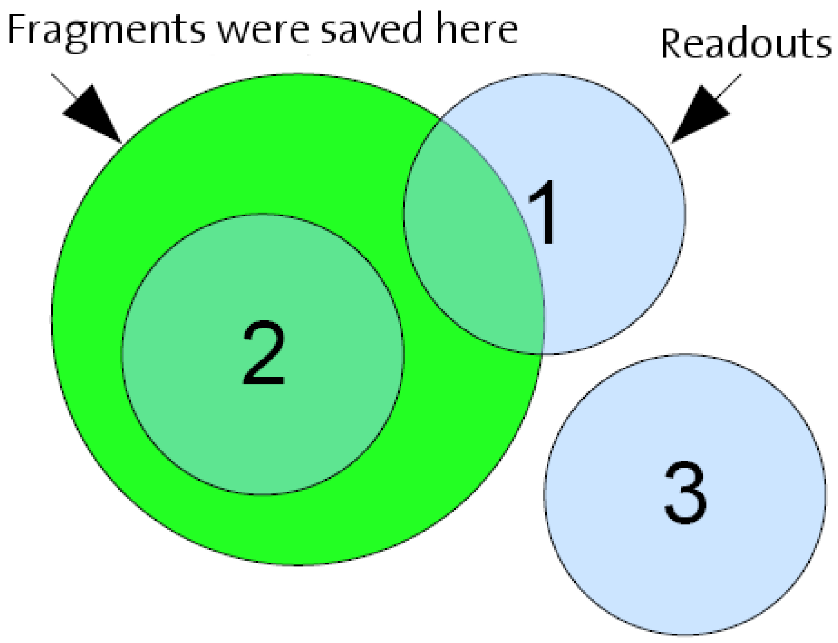
The above computations assume that the set of tags for storage and later retrieval is identical. However, if the reader is not directly positioned in the right spot during retrieval, it will not only be necessary to find the right level of where to search for the fragments, but also the right place (i.e.,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

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.



- Figure 13: Choosing the right tag set. When reading a location at a later time, the reader might not correctly catch the exact set of tags that were used during saving. In the above example, case 1 only partially overlaps, case 2 catches only correct tags, but not enough, and case 3 is totally off.

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. For time-controlled tags, this would be simple, as all fragments are stored within a few ID-levels. For usage-based ID-updates, the system could still assume a Gaussian distribution of tag-readings, as shown in **Error! Reference source not found.**, thus limiting the initial number of levels to explore.

Note that in practice, one might wish to limit the need for such exploration. A small, marked “drop zone” for file storage would make later retrieval trivial, e.g., the doorway of a meeting room, a whiteboard on the wall, or a sign next to a door. A larger deployment (e.g., the entire floor of a room) might use the same few 100 tag IDs repeatedly throughout the entire surface, allowing users to exchange data as long as they read, say, a square meter of floor space, no matter where in the room. In both cases, there would be no need for any exploration, even with limited tag reading distances.

6 Security Analysis

As pointed out in our introduction, the recent rise in social networking sites vividly illustrates that people are social creatures. The sharing of information, be it favorite TV-shows or restaurant recommendations, is an integral part of our lives. Privacy is thus not simply secrecy, but a constant boundary negotiation that allows us to disclose parts of our lives to a selected audience, without becoming completely transparent (Marx 2001). Implicit privacy control systems like FragDB support this boundary negotiation by allowing us to easily share information *locally* without making it *globally* available.

Following Schneier’s five-step security analysis process (Schneier 2003), we can describe FragDB’s security model more formally as follows:

1. *What assets are we trying to protect?* FragDB tries to prevent the *central monitoring* of shared information. It protects innocuous data, such as discussions during class, the road a particular car is driving on, or the times I spend in a local coffee shop, from central and remote access. Users of FragDB explicitly publish this data for the purpose of sharing it with others, yet want to limit the range of the recipients to those who are close in time and space. FragDB thus does *not* completely prohibit access to such data – it simply requires those interested in this data to be physically present in order to get access to it. FragDB addresses only the *acquisition* of stored data, not long-term storage or subsequent processing of already acquired data (though it could be combined with traditional digital rights management approaches to do so). FragDB does *not* try to prevent direct surveillance or monitoring of a place, e.g., through the use of a hidden camera, microphone, or RFID- or WiFi-antennas.

2. *What are the risks to these assets?* Once information is made remotely available, profiling and surveillance becomes cheap. Centralized data repositories can be easily searched, and automated processes can tirelessly assemble comprehensive profiles. FragDB limits such profiling without completely stopping one from sharing this data with others “nearby", i.e., close in both time and space.

3. *How well does the security solution mitigate those risks?* No central controller can easily query or monitor data in the system, as all stored data is protected by cryptographically secure one-way hashes (e.g., SHA-1 or SHA-256). To achieve the surveillance performance of a centralized system, the monitoring party would need to deploy people or equipment to monitor the (changing) signature of a

particular location, causing significant costs that might not be worth the effort.[5] On the other hand, there is almost no effort needed on behalf of the user to share data with local users, while still ensuring that central monitoring is rendered impractical.

4. *What other risks does the security solution cause?* In contrast to an explicit access control system, data spills might occur, where data that seemed private is actually made available to anybody that visits a particular place. However, explicit access control might make this even worse, as frustrated users tend to disable or ignore the security features of a system completely (Whitten and Tygar 1999).

5. *What costs and trade-offs does the security solution impose?* While deployment costs are lower than alternative location-based access-control mechanisms, there is an inherent trade-off between flexibility and manageability. FragDB tries to make data sharing *easy*, without leading to total surveillance. Given the complexity of real-world security and privacy, FragDB-users trade-in control for ease-of-use. While one could use *explicit* access control and a remotely accessible repository (e.g., a Web server), the added management effort of maintaining access control lists might prompt people to either formulate overly restrictive policies (i.e., very limited sharing) or overly permissive policies (e.g., disable security completely).

Note that the security of FragDB does not depend on any security features on the RFID tags. They simply serve as random keys that allow one to store and retrieve data in a distributed and encrypted fashion. Obviously, an attacker could replace “official” RFID-tags with those under his own control, thus allowing him to predict the particular fingerprint of a place at any given time. The same effect could be achieved by placing

[5] Instead of attacking the FragDB system, a monitoring party might be better

a remote reading device in a place to constantly monitor the fingerprint of a particular place. However, in both cases, it might be much easier for a determined attacker to simply install traditional surveillance equipment, rather than attack the local storage concept of FragDB. If needed, tags could be signed using a public key, yet this would entail all drawbacks of any key deployment scenario (e.g., key updates).

While the concept of time variance might also be achieved by using traditional access control methods combined with automated password expiration, this would require a non-negligible overhead for password management, in addition to the added costs of devising and maintaining an access control policy. Obviously, time variance only applies to "new" visitors to a place – it does not govern storage or processing of this data. Once one successfully retrieved some information from FragDB, e.g., the lecture notes from last year's class, one is free to store this on a local hard disk as long as desired. Note that this is a common feature with all access control mechanisms, unless additional watermarking or digital rights management (DRM) mechanisms are embedded into the data. Limiting *access* is still a powerful privacy enabler, even without controlling subsequent use, as the following analogy might illustrate: While we cannot prevent others from taking a picture of us in a public place (exception for celebrities notwithstanding) and uploading this to a public photo sharing website, this is still *very* different from a future where one could take a live picture of any (public) place in the world from the comfort of his armchair! This distinction is also present in many social and legal frameworks, e.g., regarding video surveillance. While it is perfectly fine for your neighbor to watch your doorstep from his window across the street, setting up a video camera to record it is an entirely different matter, let alone stream this video over the Internet.[6]

[6] See, e.g., www.datenschutzzentrum.de/video/vidsec_e.htm for an overview of legal issues in video surveillance.

If needed, however, data stored in FragDB could of course be watermarked in order to prove its origin, or use DRM technology to control the further use of this information.

7 Conclusions

We have designed and built a system for localized, secure storage, based on Super Distributed RFID-Tag Infrastructures (SDRI). The initial prototype demonstrates the potential of this application, and allows us to explore the uses and limits of this principle. Location-based storage offers *autonomous* protection for shared information that would otherwise be either left totally unsecured, or overly restricted by access policies that require significant user effort to create and maintain. Doorways, doormats, and floors can act as local storage repositories that provide local capture and access data to meeting participants, patrons, and visitors. Road markings can allow for localized placement of recommendations or alerts, without requiring a centralized surveillance infrastructure. Our FragDB prototype thus extends popular context-aware access control mechanisms such as verifiable location-claims (Gonzales-Tablas, et al. 2005) or Geo-RBAC (Damiani, et al. 2007) with an *implicit* protection scheme that offers little flexibility yet requires no configuration or setup. In contrast to other context-aware access control schemes, its hardware requirements are significantly lower, requiring only low-cost RFID tags and readers, and no central infrastructure (other than a database).

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.

8 Acknowledgements

Lukas Stucki was instrumental in implementing and analyzing the FragDB prototype as part of his Master's thesis. We thank the anonymous reviewers for their detailed comments, which prompted us to significantly revise and improve our initial submission. This work has been partially funded by Hitachi Systems Development Laboratories (SDL), Japan, who also provided the µ-chip RFID equipment.

Bibliography

Abowd, Gregory D. "Classroom 2000: An experiment with the instumentation of a living educational envrionment." *IBM Systems Journal* 38, no. 4 (October 1999): 508-530.

Adams, Anne, and Martina A. Sasse. "Users Are Not the Enemy." *Communications of the ACM* (ACM Press) 42, no. 12 (December 1999): 41-46.

Berendt, Bettina, Oliver Günther, and Sarah Spiekermann. "Privacy in E-Commerce: Stated Preferences vs. Actual Behavior." *Communications of the ACM* (ACM Press) 48, no. 3 (2005).

Bohn, Juergen. "Prototypical Implementation of Location-Aware Services based on a Middleware Architecture for Super-Distributed RFID Tag Infrastructures." *Personal and Ubiquitous Computing Journal* (Springer), November 2006.

Bohn, Jürgen, and Friedemann Mattern. "Super-distributed RFID tag infrastructures." *Ambient Intelligence - Second Europan Symposium (EUSAI 2004).* Berlin Heidelberg New York: Springer, 2004. 1-12.

Brands, Stefan, and David Chaum. "Distance-bounding protocols." *Workshop on the theory and application of cryptographic techniques on Advances in cryptology.* Berlin Heidelberg New York: Springer, 1994. 344-359.

Brummit, Barry, B. Mayers, John Krumm, A. Kern, and Steven A. Shafer. "Easyliving: Technologies for intelligent environments." Edited by Peter J. Thomas and Hans-Werner Gellersen. *Proceedings of the 2nd International Conference on Handheld and Ubiquitous Computing (HUC 2000).* Berlin Heidelberg New York: Springer, 2000. 12-29.

Cao, Xiang, and Lee Iverson. "Intentional Access Management: Making Access Control Usable for End-Users." *Proceedings of the 2nd Symposium On Usable Privacy and Security (SOUPS2006).* 2006.

Clarke, Ian, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. "Protecting free expression online with Freenet." *IEEE Internet Computing* 6, no. 1 (Jan-Feb 2002): 40-49.

Damiani, Maria Luisa, Elisa Bertino, Barbara Catania, and Paolo Perlasca. "GEO-RBAC: A spatially aware RBAC." *ACM Transcations on Information Systems Security* 10, no. 1 (2007): 1-42.

Foster, Patrick. "Caught on camera – and found on Facebook." *The Times Online*, July 17, 2007: http://technology.timesonline.co.uk/tol/news/tech_and_web/the_web/article2087306.ece.

Geyer, Werner, Heather Richter, Ludwin Fuchs, Tom Frauenhofer, Shahrokh Daijavad, and Steven Poltrock. "A team collaboration space supporting capture and access of virtual meetings." *Proceedings of the 2001 international ACM SIGGROUP Conference on Supporting Group Work (Boulder, Colorado, USA, September 30 - October 03, 2001) GROUP 01.* New York: ACM Press, 2001. 188-196.

Gonzales-Tablas, Ana Isabel, Klaus Kursawe, Benjamin Ramos, and Arturo Ribagorda. "Survey on location authentication protocols and spatial-temporal attestation services." *Embedded and ubiquitous computing. Proceedings of the EUC 2005 Workshops: UISW, NCUS, SecUbiq, USN, and TAUES, Nagasaki, Japan, December 6-9, 2005.* Berlin Heidelberg New York: Springer, 2005. 797-806.

Gross, Ralph, and Alessandro Acquisti. "Imagined Communities: Awareness, Information Sharing, and Privacy on the Facebook." *Workshop on Privacy Enhancing Technologies (PET 2006).* Cambridge, UK, 2006.

Hong, Jason I., and James A. Landay. "An architecture for privacy-sensitive ubiquitous computing." *Proceedings of the 2nd international conference on Mobile systems, applications, and services (MobiSys 2004).* New York: ACM Press, 2004. 177 - 189.

Johanson, Brad, Armando Fox, and Terry Winograd. "The interactive workspaces project: Experiences with ubiquitous computing rooms." *IEEE Pervasive Computing* (IEEE Press) 1, no. 2 (2002): 67-64.

Kindberg, Tim, Kan Zhang, and Narendar Shankar. "Context Authentication Using Constrained Channels." *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications.* IEEE Press, 2002. 14-21.

Kriplean, Travis, et al. "Physical Access Control for Captured RFID Data." *IEEE Pervasive Computing* (IEEE Press) 6, no. 4 (2007): 48-55.

LaMarca, Anthony, et al. "Place Lab: Device Positioning Using Radio Beacons in the Wild." Edited by Hans Gellersen, Roy Want and Albrecht Schmidt. *Pervasive Computing - Third International Conference, PERVASIVE 2005, Munich, Germany, May 8-13, 2005.* Berlin Heidelberg New York: Springer, 2005. 116-133.

Marx, Gary T. "Murky conceptual waters: The public and the private." *Ethics and information* 3, no. 3 (2001): 157-169.

National Institute of Standards and Technology. *Secure Hash Standard.* Federal Information Processing Standards (FIPS) Publication 180-2, with change notice to include SHA-224, 2004.

Olson, Judith S., Jonathan Grudin, and Eric Horvitz. "A study of preferences for sharing and privacy." *CHI '05 extended abstracts on Human factors in computing systems.* New York: ACM Press, 2005. 1985 - 1988.

Rhea, Sean, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. "Pond: the OceanStore Prototype." *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03).* Berkeley, CA, USA: USENIX Association, 2003. 1-14.

Román, Manuel, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. "Gaia: A Middleware Infrastructure to Enable Active Spaces." *IEEE Pervasive Computing* (IEEE Press) 1, no. 4 (Oct-Dec 2002): 74-83.

Sampemane, Geetanjali, Prasad Naldurg, and Roy H. Campbell. "Access control for active spaces." *Proceedings of the 18th Annual Computer Security Applications Conference.* IEEE Press, 2002. 343-352.

Sastry, Naveen, Umesh Shankar, and David Wagner. "Secure verification of location claims." *WiSe '03: Proceedings of the 2003 ACM workshop on Wireless security.* New York: ACM Press, 2003. 1--10.

Schneier, Bruce. *Beyond Fear.* New York, NY: Copernicus Books, 2003.

Serfling, Robert J. *Approximation Theorems of Mathematical Statistics.* Wiley, 1980.

Shamir, Adi. "How to share a secret." *Communications of the ACM* 22, no. 1 (1979): 612-613.

Waldman, Marc, Aviel D. Rubin, and Lorrie Faith Cranor. "Publius: A robust, tamper-evident, censorship-resistant web publishing system." *Proceedings of the 9th USENIX Security Symposium.* USENIX Association, 2000. 59-72.

Whitten, Alma, and Doug Tygar. "Why Johnny can't encrypt." *Proceedings of the 8th USENIX Security Symposium.* 1999. 169-184.