IMPLEMENTING HEALTH INFORMATION EXCHANGE WITH SEARCHABLE ENCRYPTION

BY

IGORS SVECS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Carl A. Gunter

Abstract

Health Information Exchange (HIE) is an infrastructure that facilitates exchange of electronic health records between healthcare organizations. Because medical records are highly sensitive data subject to various federal and local regulation, in addition to company policies, it is imperative to provide privacy and security guarantees, as well as audit trail.

In this thesis, we consider the problem of providing the functionality of a HIE composed of a distributed collection of providers (sources) who contribute to a centralized repository represented by a cryptographic file system, and suggest an implementation that demonstrates its feasibility. While security-enhanced file systems have been extensively studied before, more recent research establishes a rigorous standard of formally provable security properties. However, since encryption imposes overhead and loss of functionality, we propose a novel cryptographic construction called BlindStorage that enables keyword search capability over encrypted indices.

The major contribution of this work is the demonstration how an advanced encryption technique can be deployed in a context close to the requirements for a standards-based HIE. We emphasize practical aspect of our design by using Web Services-based transactions that closely follow a subset of the state-of-the-art Cross-Enterprise Document Sharing-b (XDS.b) standard in the architecture. Document Repository and Document Registry gateways are being used to interface between hospital-facing Web Services and the central BlindStorage parts of the network, so that hospital applications can use the established XDS.b exchange standard without being affected by the core BlindStorage-based implementation.

Acknowledgments

This work would not have been possible without the support of many people. I would like to thank my thesis adviser, Prof. Carl Gunter, for his mentoring with the thesis and all the work that led to it. I am also grateful to Prof. Manoj Prabhakaran for my resulting understanding of foundations in cryptography and his collaboration on the important underlying part of the thesis, and Muhammad Naveed for our ongoing collaboration on the cryptographic part of the project and implementation work. This material is partially based on the work supported by the Office of the National Coordinator through the SHARPS project, and I thank Tony Michalos and Andrea Whitesell for their administrative input to it.

Table of Contents

List of Figures

List of Abbreviations

ABE	Attribute-Based Encryption
API	Application Progamming Interface
CCD	Continuity of Care Document
CCR	Continuity of Care Record
EHR	Electronic Health Record
EMR	Electronic Medical Record
HCO	Healthcare Organization
HIE	Health Information Exchange
IHE	Integrating the Healthcare Enterprise
NFS	Network File System
PCAST	President's Council of Advisors on Science and Technology
PRG	Pseudorandom generator
PRF	Pseudorandom function
PRP	Pseudorandom permutation
NHIN	Nationwide Health Information Network
SOAP	Simple Object Access Protocol
S/MIME	Secure/Multipurpose Internet Mail Extensions
SSE	Searchable Symmetric Encryption
UC	Universal Composability
WSDL	Web Service Definition Language
XDS	Cross-Enterprise Document Sharing
XML	Extensible Markup Language

Chapter 1

Introduction

Over the last two decades the government, industry, and academia have been incraesingly recognizing the importance of health information technology. Healthcare has tended to lag other industries (like communications and financial services) in transiting to electronic records. However, as proprietary electronic medical record (EMR) systems get deployed at healthcare organization, so does the need increase to exchange records via a Health Information Exchange.

At the same time, multiple federal, state, and organizational policies prohibit insecure transfer of certain type of sensitive medical data. It follows that unless we are willing to accept negative consequences of incomplete information (that could potentially lead to misdiagnoses or adverse effects of interacting prescription medication), we must use strong cryptography that could be mathematically proven to be equivalent to physical security.

There exists a variety of HIE architectures, both in production and as prototypes, that prevent unauthorized access to documents in transit. However, they are mostly focused access control mechanism, and in cases where encryption at rest is used, no formal security analysis is possible due to their complexity.

All architectures of complex systems could be classified as either distributed or centralized, and HIEs are no exception. From the security point of view, distributed approaches are often optimal, as nodes that wish to communicate with each other can establish a secure point-to-point link, and exchange information directly without leaking anything to the environment. Unfortunately, point-to-point design often leads to quadratic complexity and inefficiency, and limited functionality. For instance, it is very difficult to implement a successful search feature without a central index. Similarly, each transaction in the system should be subject to audit, which is significantly easier to accomplish with a centralized component that can take

part in all transactions. Therefore, we argue that a centralized architecture that guarantees security no weaker than in a distributed approach is optimal.

Virtually the only way to accomplish this goal is to use cryptography, which immediately introduces another challenge — severe loss of functionality. As a rule, the stronger the cryptogtaphic scheme is, the less observers can find out about the data, and therefore, the fewer operations could be performed on it. This relegates the hub in centralized architectures to a mere "dumb" storage device, losing ability to leverage potential functionality that we discussed before. This is the reason why the research community makes major efforts to enrich cryptographic schemes with additional functionality.

One of the most popular, and wildly applicable, examples of such functionality is searchable encryption. It generally refers to schemes that allow pre-precessing of the record to contruct an index, which will later be used for keyword look-ups. Moving beyond single keywords — e.g., to wildcard searches or even keyword conjunctions — is still a research question.

In this thesis, we introduce a searchable encryption scheme called BlindStorage that can be used as a foundation of a centralized HIE architecture that maintains strong security guarantees. It was designed with security proofs in mind from the early stages, and therefore provides an advantage over other HIE architectures. In addition, one of its strong points is that in only requires a Get/Put interface to a storage devices, which allows to use readily available highly scalable NoSQL products, instead of creating a storage server from scratch.

However, we must still demonstrate that our HIE will be practical and interoperable enough to provide additional motivation for its implementation. In order to show this, we wrap BlindStorage around a web services-based environment that closely follows the principles of Cross-Enterprise Document Sharing (XDS.b) profile, which is used as the standard of choice for established health information exchange. We design gateway nodes that relay messages between web services and BlindStorage parts of the architecture. We implemented each part in a programming language that best fits its requirements (C++ for BlindStorage — bit manipulation, efficiency, pointer arithmetic; Java EE for web services - readily available frameworks, simplicity of higher-level language abstractions). The system was also tested by exchanging a CCD document that was published for CDA guideline validation.

This rest of this thesis is organized as follows. In Chapter 2 we give background and discuss related work on approaches to secure storage, searchable encryption, and applications of encryption in the healthcare domain. We also discuss health information exchanges — a number of proposed architectures, and messaging standards in use. Chapter 3 presents the HIE architecture, and how it can be implemented with BlindStorage building blocks — low-level file system and searchable encryption that is built on top of that, as well the connectivity layer that relays messages between web services and BlindStorage components. Implementation is discussed in Chapter 4, and we talk about limitations, alternative approaches, and possible extensions in the final two chapters.

Chapter 2

Background and Related Work

2.1 Secure Cryptographic Storage

Over the past decade, we have observed several major trends in computing that direct the industry and research efforts. First, more and more computing devices are connected online each day, along with the humans that use them. Major new industries such as advertising and social media accommodate new customers, but require gathering vast quantities of data for analytics. Additionally, already established industries seek to better understand and optimize their processes with data mining (business intelligence). This gives rise to the phenomenon of "big data", directing a lot of effort towards efficient management of data sets. Second, this data grows at a faster rate than the capacity of individual processors as predicted by Moore's law, making it necessary to distribute storage and computation across multiple machines. Finally, financial advantages such as specializing and aligning business structure around specific types of services that benefit from economy of scale enabled growth of companies that specialize in managing other parties' data. These trends set a fertile ground for rising popularity for cloud computing, hosted systems, and storage outsourcing.

While some industries work with data that is not considered particularly sensitive and therefore can immediately take advantage of cloud storage providers, healthcare is abided by multiple regulations to ensure confidentiality and integrity of patients' data. Any solution that uses third-party services providers must provide strong security and privacy guarantees on top of their primary function (e.g., file storage). There is a variety of federal and local laws that prohibit transfer of certain classes patients' data in health information exchanges, such as genetic information, substance abuse, mental health, and HIV/AIDS. However, if security of an HIE-like system could be mathematically proven to not leak any information

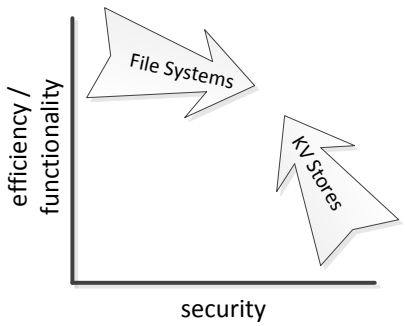

Figure 2.1: Convergence between efficient file systems and secure key-value stores

at all, it might be possible to not be subject to these laws.

As a result of increasing recognition of the importance of security, research community has been exploring various approaches towards securing storage systems with cryptography. The overall process is illustrated in Fig 2.1. One approach is to retrofit existing file systems that were primarily designed for efficiency and functionality with partial security features such as encryption of file contents and paths, while another approach is to move forward from theoretically secure but inefficient key-value stores, enriching them with functionality and improving efficiency.

2.1.1 File systems

Traditionally, a file system has been considered local to a computer; therefore, most research was directed towards making file systems consistent, error-resistant, and efficient, and little towards security. Early distributed file systems such as NFS did not focus much on security either, and when they did, they did not consider protecting data from the server, as the most common setting was that both the server and the clients were under the same authority. Hence, only access control mechanisms such as authentication and authorization were being improved to prevent unauthorized users, while still assuming that systems administrators were working under common authority. As a result of this common setting, little effort was made to secure contents of the files from the file server.

One of the first file system designs that explicitly focused on security through cryptography, the Cryptographic File System (CFS) was introduced by Matt Blaze in 1993 [1]. Users can associate a key with the directories they wish to protect, and file contents and paths in

corresponding directories will be encrypted and decrypted by a system-level daemon. Cleartext is never stored or sent to remote file server, and CFS can use any available file system including remove file servers such as NFS.

Next incremental improvement of CFS is the Transparent Cryptographic File System (TCFS) [2]. It provides more convenience and flexibility compared to CFS, by allowing more granular file-level encryption (e.g., an option to skip executables and only encrypt document), and one-time authorization that will persist across file accesses by the same user.

Cryptfs [3] is another prominent early cryptographic file systems that is similar to CFS and TCFS. It is implemented as a kernel module with a stackable vnode interface, meaning that it serves as a wrapper around traditional vnodes. It provides a number of incremental advantages, such as not requiring auxiliary storage space on a local disk (that could lead to potential vulnerabilities), and working with later versions of NFS.

One common major disadvantage of all these approaches is the total lack of theoretical security analysis, potentially making them vulnerable for future attacks. In addition, they do not provide any extra functionality or flexibility beyond file access, with further restrictions by their kernel module implementations. While we may be convinced that the basic functionality of encrypting file contents is reasonably error-free, it is much more difficult for the authors to convince us that metadata-relation operations remain secure.

2.1.2 Key-value stores

Key-value stores have been gaining popularity in clouds and other distributed systems. Multiple large-scale cloud storage providers emerged, such as Amazon S3, Microsoft Azure, and Dropbox, providing a key-value interface to their systems. Traditional relational databases have given way to NoSQL stores that exhibit higher scalability and response time in exchange for lesser consistency. In addition, while securing relational databases is possible as demonstrated by projects such as CryptDB [4], the inherent need for server-side computation due to relational algebra severely limits cryptographic approaches. On the other hand, it is trivial to encrypt values in key-value stores, and decrypt them on the client side, while

still taking advantage of the distributed nature of cloud infrastructure. Because of this, significant research efforts are directed towards enhancing cryptographic cloud storage systems with extra functionality, such as adding search capability [5], proof of data possession protocols [6], and public auditing [7]. Seny Kamara and Kristin Lauter give an excellent overview of cryptographic cloud storage architectures in [8].

In the next three sections, we concentrate on the problem of searchable encryption. There are two general approaches to enabling search - either using a homomorphic encryption scheme that remain inefficient, or by designing a data structure that contains a precomputed index with an elaborate construction. Within the data structure approach, schemes are differentiated into oblivious RAMs, private-key cryptosystems, and public-key approaches.

Oblivious RAMs

Oblivious RAM (O-RAM) is a data structure that solves the problem of searchable encryption in full generality, first proposed by Goldreich [9] and Ostrovsky [10]. It was first investigated in context of untrusted random access memory that is used by programs being executed on a local computer. Oblivious RAMs hide access pattern, randomizing the location of an object after each access, preventing the adversary from correlating trapdoors with returned results. However, this data structure is very inefficient, requiring logarithmic complexity of the number of rounds, as well as bandwidth per request.

O-RAM construction was improved by Pinkas and Reinman [11], who applied recent advances in data structures such as Cuckoo hashing to enable linear external memory, and $O(\log^2 n)$ request complexity. Since request complexity is still non-constant, we consider this construction inefficient for practical considerations. In addition, their scheme only guarantees *amortized* logarithmic complexity, meaning that client may potentially wait $O(n)$ time in the worst case. This is unacceptable with commercial cloud storage providers who typically provide quality of service guarantees via service level agreements.

One of the latest works in O-RAMs was introduced by Stefanov et. al [12]. They decided to not pursue traditional design decisions of O-RAMs, and instead evaluate the system in a practical setting, and determine if practical results justify suboptimal theoretical properties.

Their scheme requires linear client storage, but the evaluation demonstrates that for a 1 TB file system, 858 MB of storage is needed on the client. Additionally, they reduced the constant factor of the bandwidth overhead to 26x, for 1 TB configuration. The major drawback of this scheme is that both the server and the client needs to perform complex operations and store state, which makes error recovery difficult, and prevents using established commercial cloud key-value stores such as Amazon S3 and Dropbox. Additionally, this scheme still has logarithmic bandwidth overhead that is inherent to all oblivious RAMs.

Symmetric cryptosystems

One of the most influential works on symmetric searchable encryption was done by Curtmola et al. [5], which introduced improved definitions for both non-adaptive and adaptive searchable encryption, as well as corresponding constructions. In this work, their SSE-1 construction takes constant number of rounds, and requires linear storage space on the server, proportional to the number of keywords and documents. However, the major drawback of their scheme is that it requires non-trivial operations on the server-side, which cannot be implemented with a simple GET/PUT interface that is supported by many popular key-value stores today. This construction was used as the foundation for a more comprehensive storage system CS2 [13], which supports additional features such as search authentication and integrity. Their SSE-1 scheme was extended in later work to support dynamic updates [14] and distributed architectures [15]. However, these newer papers offer only incremental advantages over the original construction.

Assymmetric cryptosystems

Public-key searchable encryption allows separation of document source and producer roles, by enabling producers to use consumer's public key to upload documents to the storage. Representative work in this area was done by Boneh et al. [16], with a later construction that allows private information retrieval [17]. The fundamental drawback that inherent to all public-key searchable encryption schemes is that the set of keywords is vulnerable to dictionary attacks, as search trapdoors can be generated by anyone who possesses document

consumer's public key. We consider keyword protection important enough in the healthcare domain (e.g., if keywords are used for diagnoses then sensitive conditions will be revealed), and therefore, do not use public-key encryption.

2.1.3 Encryption in Healthcare Context

As the importance of securing electronic health records started to be recognized in the last decade along with advances in cryptographic systems, it was imminent for researchers to propose applications of cryptography in healthcare. In *Patient Controlled Encryption* [18], the authors argue that encryption can be used as a tool to implement an access control system that allows patients to control how their records are disseminated. They explored a way to encrypt a hierarchical XML-based medical record, such that different parts of it would be accessible with different keys, effectively implementing access control. In Indivo [19] HIE architecture, the authors propose to use encrypted stores at HCOs as document repositories, and the system uses general-purpose access control policies to enable personal control of health records. This design, however, does not use any advanced encryption schemes, meaning that medical records would have to be decrypted by the server before being transmitted to clients.

On the public-key cryptography side, there have been several proposals to use attribute-based encryption (ABE), either to allow patients to share their records among multiple healthcare providers [20], or to enable healthcare organizations to securely export medical records directly to patients' mobile devices [21]. A major advantage of ABE in healthcare context is that sensitive categories of documents (e.g., diagnoses) could be mapped to attributes, and that individuals or organization that do not possess the attributes would be denied decryption. There are still challenges with ABE schemes, such as managing effective key distribution infrastructure and dealing with key and attribute revocations.

2.2 Health Information Exchanges

2.2.1 Overview

Health Information Exchange refers to infrastructure that facilities exchange of medical records between healthcare organizations, and possibly, patients. As the government started federal and local programs to encourage implementation of information technology in healthcare, hospitals increased their efficiency and automated operations by using commercial electronic medical records (EMR) systems. More often than not, EMR formats are proprietary and not easily exportable across organizations, making it difficult to exchange medical records between HCOs. Hence, the need for regional and even national HIEs became evident, and academia, government, and industry invested heavily in exploring methods to implement HIEs.

There is a variety of challenges associated with exchange of medical information. First and foremost, medical data is considered extremely sensitive, and several laws have been enacted to regulate access to it and its reporting. One of the first major laws is the Health Insurance Portability and Accountability Act (HIPAA), which introduces the Privacy Rule that is heavily concerned with managing patient's informed consent, and the Security Rule, which specifies a number of safeguard policies that must be implemented by HCOs to protect medical data. Following HIPAA, the Health Information Technology for Economic and Clinical Health (HITECH) Act was passed to encourage "meaningful use" of electronic health record, with electronic health information exchange being one of the major components of meaningful use criteria. A number of other laws such as the federal Genetic Information Nondiscrimination Act (GINA), and state-level Illinois Health Information Exchange and Technology Act define sensitive information that is not allowed to be transmitted insecurely. It follows that an effective HIE must either properly segment data (which is a major non-trivial multifaceted problem), or provide a level of electronic security that is equivalent to physical security (which can be accomplished by formally proving security properties of cryptographic systems).

An important component of a successful HIE is reliable auditing, which puts constraints

on architectural design choices.

2.2.2 Architectures

In 2010, the Presidents Council of Advisors on Science and Technology (PCAST) released a report [22] that recommended an architecture for nationwide health information exchange. The report proposes an approach based on a separation of duties between managers of repositories of health data and a key authority that does not manage health data but grants access by making access control decisions and distributing keys. It emphasizes the importance of protection of data both "on the wire" (in transit), and "at rest" (in a storage system). This report had influence on a number of HIE architectures, and we particularly incorporate its recommendation of encrypting data at rest with the possibility of using encryption as access control in future extensions.

The Nationwide Health Information Exchange Network (NHIN) initiative launched two major projects - CONNECT and DIRECT.

CONNECT[1] is a large-scale architecture that is composed of a set of profiles and Java-based web services for a secure interoperable health information exchange [23]. While it provides extensive facilities for authentication and authorization, it remains unclear whether the data is properly encrypted across the entire chain of transaction. Additionally, the architecture is complex enough to rule out any attempts of formal security analysis.

The DIRECT project[2] is a lightweight architecture for HIE that is based on S/MIME protected email messages. In this design, the DNS system is augmented to contain "pointers" to HCOs that participate in the exchange.

An example of a non-US health information exchange is the partially centralized Dutch EPD [24] system, where the central hub that is run by the government contains a reference index per patient, and the actual documents are located on hospitals' systems. We favor the principles of a partially centralized approach where we can use the third party to provide as much functionality as possible while not compromising on security (e.g., auditing or

[1] http://www.connectopensource.org/
[2] http://wiki.directproject.org/

replication), but in the EPD design, data breach of the hub can lead to major consequences as there is no at-rest encryption.

2.3 Message Standards

Since one of the major goals of a Health Information Exchange is to promote interoperability between different proprietary health information management systems, the choice of envelope standard for messaging is important. Web services have been established as a popular technology for interoperable remote method invocation, so the natural choice is to use an XML [25] based standard.

2.3.1 Electronic Health Records

Electronic health records contain both structured data (such as patient's information and diagnosis/billing codes), and unstructured information (doctor's notes), and as a result of this, most widely-used standards for EHRs use an XML-based format. The underlying standard behind many clinical systems is Health Level Seven (HL7), which is designed to support all healthcare workflows. A subsection of HL7 that is used for information exchange is collectively called Clinical Document Architecture (CDA), which is further subdivided into Continuity of Care Records (CCR) that are used for patient summaries, and Continuity of Care Documents (CCDs), which are used for more detailed information. The major benefit of using CCDs is that this format is encouraged by the HITECH Meaningful Use criteria, and therefore, the architecture proposed in this thesis must support it.

2.3.2 Cross-Enterprise Document Sharing

Cross-Enterprise Document Sharing.b (XDS.b) integration profile was developed by the IHE and specifies a set of actors (web services) that participate in the HIE process, as well as required and optional SOAP-based message formats that are used between these services. Even though new revisions of XDS are still under active development, with the latest revision

9.0 being released on August 31, 2012, it proved to be successful enough to be selected for a number of HIE implemenetations. For example, Illinois' Office of Health Information Technology requires their ILHIE prototypes to support XDS.b [26].

Since we aim to support exchange of encrypted data between enterprises, we consider how to incorporate it within the XDS.b standard. One approach is to use XML Encryption [27], which introduces `EncryptedData` and `EncryptedKey` elements. However, recent work [28] has shown significant vulnerabilities in XML Encryption, and we decided to not use it.

Chapter 3

Design

3.1 Architecture

The general architecture of our health information exchange design is composed of two parts: web services-based end notes (Document Sources and Consumers) that represent healthcare organizations and use transactions that are heavily influenced by Cross-Enterprise Document Sharing (XDS.b), and the core storage system that is based on searchable encryption implemented on top of our novel cryptographic BlindStorage construction. The overall architecture is illustrated in Fig. 3.1 with web services components color-coded in blue.

The XDS.b profile specifies five actors: Document Source, Document Consumer as end nodes, Document Registry that maintains metadata for all documents, Document Repository that physically stores the records, and Patient Identity Source that aggregates unique identifiers for patients and provides validation services[1]. We decided that implementing Patient Identity Source was outside the scope of this thesis, and that Document Registry function is implemented by the BlindStorage server. Its index and data structures contain the mapping between the keywords and references to the documents. The documents themselves are stored in a separate storage, which is also accessible via a Get/Put interface.

In the existing design, keywords belong to a flat domain (i.e., they represent one queriable attribute such as patient's name), although a more complex keyword domain could be built by using prefixes. Therefore, only a limited subset of query operations is currently supported - specifically, requesting documents by patient's name.

The interface between web components and BlindStorage is implemented with gateways that translate XML-based messages into corresponding BlindStorage API calls. These gate-

[1] http://wiki.ihe.net/images/d/d7/XDS-Actor-Transaction-b.jpg

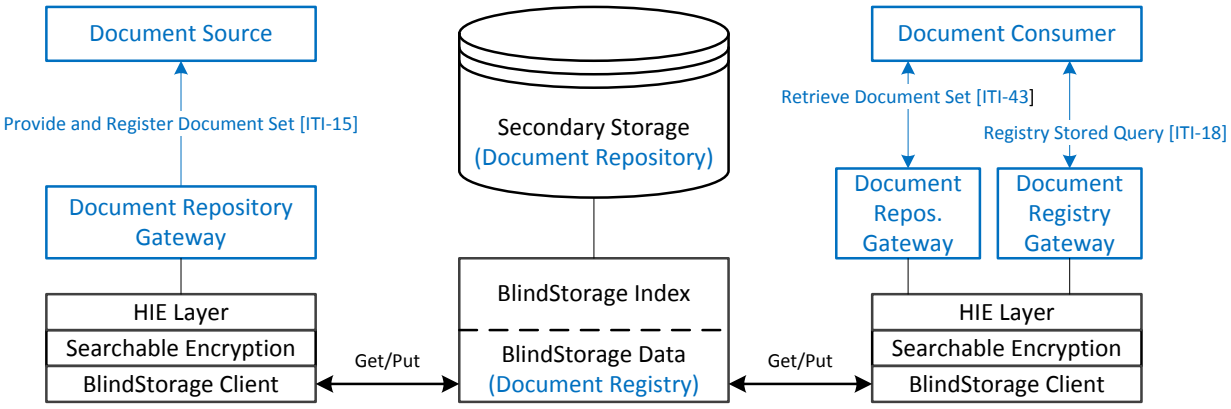

Figure 3.1: Architecture overview: Web services-compliant end nodes exchange documents via gateways that relay messages to the BlindStorage server. Messages between the services are labeled with XDS.b transactions that they correspond to.

ways are considered trusted code and define the boundaries in our security analysis, and must reside on the healthcare organizations' systems, either as separate visible nodes or as intercepting proxies. The advantage of intercepting proxy configuration is that end nodes are completely oblivious to the translation process, resulting in simplified configuration. On the other hand, it complicates network design and introduces more points of failure while making changes to the HIE; therefore, we decided to use the explicit approach. In this architecture, end notes would directly establish connections to the gateways.

An alternative design was considered when the gateways would be located at the third party that is responsible for record storage. This provides the benefit of reducing multiple instances, as only one set of adapters would need to be active per the entire HIE system, as opposed to having one adapter per HCO. This design could be viable without redoing the security analysis if we assume that the organization that hosts these servers does not collude with the core HIE provider. However, due to the increasing organizational complexity this design was not selected for our architecture.

The main cryptographic storage architecture could be represented as a stack of layers: BlindStorage that implements the key-value store, searchable encryption that uses BlindStorage to provide document indexing and keyword search capability, and the application-specific HIE layer that translates domain-specific requests into keyword searches. The following sections in this chapter will fully describe each of these building blocks, from the lowest to the highest layer.

3.2 BlindStorage: Cryptographic File System[2]

3.2.1 Definition

The Syntax. A blind storage system consists of a client and a "dumb" storage server. The server is expected to provide only two operations, download and upload. The data is represented as an array of *blocks*; the download operation is allowed to specify a list of indices of blocks to be downloaded; similarly, the upload operation is allowed to specify a list of data blocks and indices for those blocks.

A blind storage system is defined by three polynomial-time algorithms on the client-side: BSTORE.Keygen, BSTORE.Build and BSTORE.Access. Of these, BSTORE.Access is an interactive protocol.

- BSTORE.Keygen takes security parameter as an input and outputs a key $\mathsf{K}_{\text{BSTORE}}$ (typically a collection of keys for the various cryptographic primitives used). Note that $\mathsf{K}_{\text{BSTORE}}$, which the client is required to retain throughout the lifetime of the system, is required to be independent of the data to be stored.

- BSTORE.Build takes as inputs a key $\mathsf{K}_{\text{BSTORE}}$, a list of ids and data (encoded as blocks) $\{\mathsf{id}_i, \mathsf{data}_i\}_{i=1}^{t}$ for all the files the storage will be initialized with, and generates an array of blocks A to be uploaded to the server.

- BSTORE.Access takes as input file id id, an operation specifier op (BSTORE.read, BSTORE.write, BSTORE.update or BSTORE.delete), and optionally data data (if op is BSTORE.write or BSTORE.update). Then it interacts with the server (through the upload/download interface) and returns a status message and optionally file data (for the BSTORE.read and BSTORE.update operations). For the BSTORE.update operation, BSTORE.Access allows more flexibility:[3] first it requires only id as input, and outputs the current size of the file with that ID; then it accepts as input (an upper bound on)

[2] Joint work with Manoj M. Prabhakaran and Muhammad Naveed

[3] Our construction achieves this level of flexibility without efficiency overheads; one can always use a BSTORE.read followed by a BSTORE.write to get the effect of an update, but this is less efficient and potentially reveals more information.

what the size of the file will be after update; then it outputs the current file data, and only then requires the new data with which the file will be updated.

Security Requirement. The security requirement of a blind-storage system is specified following the "real/ideal" paradigm that is standard in cryptographic literature. This includes specifying an adversary model and an "ideal functionality," as detailed below. The formal security requirement we shall require is that of Universally Composable security [29] (but restricted to our adversary model).[4]

In the adversary model we consider, the adversary is allowed to corrupt only the server *passively* — i.e., as an honest-but-curious adversary. (If the client is corrupt, we need not provide any security guarantees.)

The ideal functionality is specified as a virtual trusted third party $\mathcal{F}_{\text{STORE}}$ that mediates between the client and the server (modeling the information leaked to the server). $\mathcal{F}_{\text{STORE}}$ accepts two commands from the client: $\mathcal{F}_{\text{STORE}}$.Build and $\mathcal{F}_{\text{STORE}}$.Access, along with inputs to these commands (which are identical to the inputs to BSTORE.Build and BSTORE.Access as described above). In this ideal model, it is $\mathcal{F}_{\text{STORE}}$ which maintains the collection of files, and performs all the operations specified by the $\mathcal{F}_{\text{STORE}}$.Build and $\mathcal{F}_{\text{STORE}}$.Access commands. In addition, it reveals limited information to the server as follows:

- On receiving the command $\mathcal{F}_{\text{STORE}}$.Build, $\mathcal{F}_{\text{STORE}}$ sends to the server the system parameters — namely, upperbounds on the total number of files and total amount of data to be stored.

- On command $\mathcal{F}_{\text{STORE}}$.Access, $\mathcal{F}_{\text{STORE}}$ sends a tuple $(\mathsf{op}, j, \mathsf{size})$ to the server, where op specifies what the access operation is,j is the last instance when the same file was accessed ($j = 0$ means that this file was not accessed before), and size is the size (in number of blocks) of that file. For the BSTORE.update operation, the size of the file revealed is the larger of the sizes before and after the update.

[4] We remark that for our setting of passive adversaries, UC security is a conceptually simpler notion than for the setting of active adversaries. Nevertheless, for the sake of concreteness, we use the UC security model, which automatically ensures security even when the inputs to the client are adaptively chosen, influenced by the adversary.

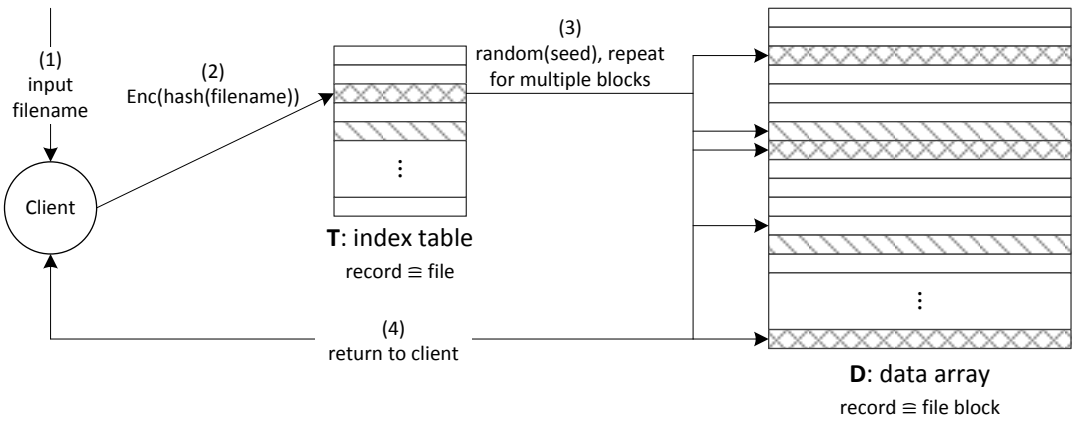

Figure 3.2: Simplified diagram of BlindStorage that shows two main data structures: index table T and data array D.

We remark that even when using the ideal $\mathcal{F}_{\text{STORE}}$ functionality, an adversary can learn some statistics about the files and accesses by analyzing the patterns in the information revealed to it. Such information could indeed be sensitive, and it is up to the higher-level application that uses a blind-storage system to ensure that this is not the case. The cryptographic construction seeks to only match the guarantees given by $\mathcal{F}_{\text{STORE}}$.

3.2.2 Construction

Before we consider formal specification of the construction, consider the *simplified* diagram in Figure 3.2. In this example, we consider the scenario when the client receives a filename as its input (1) and retrieves the contents of the file from the servers, and there are two files stored in the file system, represented by different patterns. Upon receiving the file name, the client applies hash function to it to generate file id, and then encrypts file id with format-preserving encryption to generate the index in the table T that contains the record with addressing information (2). Each record in T corresponds to a potential file in the file system, so that the size of T is the upper bound on the total number of files we allow to store. After retrieving T-record, the client generates a pseudorandom set of indices of encrypted blocks that reside in the data array D by running a pseudorandom number generator a number of times that depends on the file size (3). Therefore, the number of records in D multiplied by the block size determines the maximum capacity of the file system. This set of encrypted blocks is returned to the client (4) and contains both required file blocks and

possibly blocks that either belong to some other file, or are empty.

Next steps are not shown in the diagram. The client decrypts the set, determines which blocks belong to the file that was requested, and returns them to the upper layer. After that in a BSTORE.read operation, these blocks are re-encrypted, their version number is incremented, and they are uploaded back to the array D. In case of BSTORE.update, set size may be increased and new blocks are merged into the set, after which the set is pushed to the server, and the corresponding index entry in T is updated with new file size.

The construction uses a variety of symbols for parameters, functions, data structures, keys, and other elements, and we list them all along with their use in the reference table 3.1 for easy look-up.

A BlindStorage construction scheme.

The construction relies on the following primitives:

- a pseudorandom function (PRF), Φ,
- and a pseudorandom permutation (PRP), Ψ with short input/output strings; such a PRP is a special case of a format-preserving encryption.
- a pseudorandom generator (PRG), Γ.

(In the implementation of our prototype, as described in Chapter 4, Φ and Ψ will be implemented relying on the AES block-cipher.) The construction is given in terms of various size parameters n_{D}, m_{D}, n_{T} and m_{T}, and an expansion parameter $\alpha > 1$, and a super-logarithmic function $\beta(\cdot)$, that are all specified later. k stands for the security parameter, and it is an implicit input to all the cryptographic primitives.

- BSTORE.Keygen: A key K_{Φ} for the PRF Φ, and a key K_{ID} for the PRP Ψ are generated; $\mathsf{K}_{\mathrm{BSTORE}}$ is set to be the pair $(\mathsf{K}_{\Phi}, K_{\mathrm{ID}})$.
- BSTORE.$\mathsf{Build}(\mathbf{F}, \mathsf{K}_{\mathrm{BSTORE}})$: $\mathbf{F}$ is a list of files $\mathsf{f} = (\mathsf{id}_{\mathsf{f}}, \mathsf{data}_{\mathsf{f}})$. Below $\mathsf{size}_{\mathsf{f}}$ denotes the number of blocks in an encoding of $\mathsf{data}_{\mathsf{f}}$; each block has two short header fields

Table 3.1: Reference table of symbols in BlindStorage construction.

A	combined array of T and D for uniform indexing and simplified key-value backend, optional to implement
B	array of versions of all blocks in D; can be merged with D in the implementation
D	array that contains all encrypted file blocks, see Figure 4.2 for format
f	file under consideration, consists of a file name and its data
$\mathbf{F}$	set of all files for initial construction
id_f	numeric file identifier; limited by $\lvert\mathsf{T}\rvert$
k	security parameter; needed to ensure negligible probability of failure in the analysis
K_{ID}	key of the PRP Ψ (indexing of T)
K_Φ	key of the PRF Φ (encryption of records in T adn D)
$\mathsf{K}_{\text{BSTORE}}$	a pair of $(\mathsf{K}_\Phi, K_{\text{ID}})$
op	access operation to perform on BStore, one of read, write, update, delete
S_f	set of blocks in D that contains blocks of f, as well as other "overhead" blocks
$\widehat{S}_\mathsf{f}$	only those blocks in S_f that constitute file f, without "overhead" blocks
size_f	size of file f— number of blocks
T	index table that contains addressing information for data blocks, see 4.1 for format
α	expansion parameter that specifies the number of overhead blocks in S_f
$\beta(k)$	defines the lower bound on $\lvert S_\mathsf{f}\rvert$ for security to hold
Γ	pseudorandom generator used to generate a sequence of pseudorandom block addresses in D from a single number; seeded with σ
Φ	pseudorandom function for encrypting records in tables T and D
Ψ	pseudorandom permutation for hiding the index in T
σ_f	seed of Γ corresponding to file f
τ_f	index in T that specifies the record corresponding to file f

containing $\mathsf{id_f}$, and a *version number* initialized to 0. ($\mathsf{id_f}$ is not allowed to be all 0s, which is reserved to indicate a free block.)

- Let D be an array of n_{D} blocks of m_{D} bits each, and T be an array of n_{T} (shorter) records of m_{T} bits each.
- Initialize every block in D and every record in T with all 0s (to be encrypted later).
- For each file f in $\mathbf{F}$,
  1. Generate a pseudorandom subset $S_{\mathsf{f}} \subseteq [n_{\mathsf{D}}]$, of size $|S_{\mathsf{f}}| = \max(\lceil \alpha \cdot \mathsf{size_f} \rceil, \beta(k))$ as follows.
     (a) Generate a fresh seed σ_{f} for the PRG Γ, uniformly at random. σ_{f} is not allowed to be all 0s, which is reserved to indicate an empty entry.
     (b) Run the PRG for sufficiently many iterations, and parse the resulting string $\Gamma(\sigma_{\mathsf{f}})$ into a sequence of $|S_{\mathsf{f}}|$ integers in the range $[n_{\mathsf{D}}]$. (Duplicate numbers in the sequence can be removed now, by inserting the numbers into a binary search tree as they are generated, or more conveniently, when the subset $\widehat{S}_{\mathsf{f}}$ is generated below.)
     (c) These numbers form the subset S_{f}.
  2. At location $\tau_f = \Psi_{K_{\mathrm{ID}}}(\mathsf{id_f})$ in T, record $(\sigma_{\mathsf{f}}, \mathsf{size_f})$. Note that this can be used to reconstruct S_{f} given $\mathsf{id_f}$ (using accees to T).
  3. Check if at least $\mathsf{size_f}$ blocks in D that are indexed by the numbers in S_{f} are free (this can be done by checking the headers of those blocks, but in this phase, a sorted "free-list" could be used to do this faster). If not, *abort*. By the choice of our parameters, this will happen only with negligible probability.
  4. Pick a random subset $\widehat{S}_{\mathsf{f}} \subseteq S_{\mathsf{f}}$ of size $|\widehat{S}_{\mathsf{f}}| = \mathsf{size_f}$, such that the blocks in D that are indexed by the numbers in $\widehat{S}_{\mathsf{f}}$ are all free. (Instead of picking a truly random subset, we may rely on the fact that the numbers in the sequence used to generate S_{f} are in a pseudorandom order, and we can pick a prefix of this sequence with $\mathsf{size_f}$ distinct numbers indexing free blocks.)

5. Write the $\mathsf{size_f}$ blocks of $\mathsf{data_f}$ onto the blocks in D that are indexed by the numbers in $\widehat{S}_\mathsf{f}$ (in increasing order). These blocks get marked as not free.

– Encode T (which has shorter records) into blocks of the same size as D (each block with a version number field, but no id field) and combine it with D into a single array A. This is done in such a way that given an index τ into T, it is possible to identify a single block of A that contains the record $\mathsf{T}[\tau]$.

– Encrypt each block of A using the PRF Φ and the key K_Φ. The version number field is left unencrypted, while the rest is encrypted using the version number (initialized to 0) and the index number of the block as IV. More precisely, for the i^{th} block $\mathsf{A}[i]$, we split it as $v_i||\mathsf{B}[i]$ (v_i being the version number), and then update $\mathsf{B}[i]$ to $\mathsf{B}[i] \oplus \Phi_{\mathsf{K}_\Phi}(v_i||i)$.

(If the block-size of the PRF is less than the size of the block $\mathsf{B}[i]$, then a few lower-order bits of the IV are reserved for use as a counter, to obtain multiple blocks from the PRF for a single block in A.)

- BSTORE.$\mathsf{Access}(\mathsf{id}, \mathsf{op}, \mathsf{K}_{\text{BSTORE}})$: We describe the case when op = BSTORE.update, and mention how the other operations differ from it.

1. First, compute $\tau = \Psi_{K_{\text{ID}}}(\mathsf{id})$, and recover (the block containing) the record $\mathsf{T}[\tau]$ from the server.
2. Decrypt the block (by unmasking it with $\Phi_{\mathsf{K}_\Phi}(v_i||i)$ where i is the index of the block containing the record $\mathsf{T}[\tau]$, and v_i is the version number encoded in the block), to obtain $(\sigma_\mathsf{f}, \mathsf{size_f})$.
3. If σ_f is not all 0s, then output $\mathsf{size_f}$ as the current size of the file. Else, output a message indicating that the file is not present; also, for the next step, set σ_f to be a freshly generated seed, and set $\mathsf{size_f}$ to be 0.
4. Accept as input $\mathsf{size'_f}$, the size of the file after update.
5. Use σ_f to reconstruct a set S_f as described above, but with size

$$|S_\mathsf{f}| = \max(\lceil \alpha \cdot \max(\mathsf{size_f}, \mathsf{size'_f}) \rceil, \beta(k)).$$

Then retrieve the set of blocks $\mathsf{D}[S_\mathsf{f}]$ from the server.

6. Decrypt each of these blocks (using Φ_{K_Φ} and the version number and index of the respective blocks) and identify $\widehat{S}_\mathsf{f}$ as the set of indices of blocks belonging to this file (by checking if their headers match id). If $\widehat{S}_\mathsf{f}$ is not empty, combine these blocks together (in increasing order of their indices) to recover the entire contents of the file, and output it.
7. Accept as input new contents data' encoded as $\mathsf{size}'_\mathsf{f}$ blocks.
8. Identify a subset $\widehat{S}'_\mathsf{f} \subseteq S_\mathsf{f}$ of size $\mathsf{size}'_\mathsf{f}$ as follows. If $\mathsf{size}'_\mathsf{f} < \mathsf{size}_\mathsf{f}$, then $\widehat{S}'_\mathsf{f} \subseteq \widehat{S}_\mathsf{f}$, is a random subset of $\widehat{S}_\mathsf{f}$. Else, $\widehat{S}_\mathsf{f} \subseteq \widehat{S}'_\mathsf{f} \subseteq S_\mathsf{f}$ such that $\widehat{S}'_\mathsf{f} \setminus \widehat{S}_\mathsf{f}$ is a random set of free blocks in S_f. If no such subset exists, *abort*. Again, by the choice of our parameters, this will happen only with negligible probability.
9. Then update $\mathsf{D}[\widehat{S}'_\mathsf{f}]$ with the blocks of data'. If $\mathsf{size}'_\mathsf{f} < \mathsf{size}_\mathsf{f}$ mark as free the blocks indexed by $\widehat{S}_\mathsf{f} \setminus \widehat{S}'_\mathsf{f}$.
10. Update $\mathsf{T}[\tau]$ with $(\sigma_\mathsf{f}, \mathsf{size}'_\mathsf{f})$.
11. Encrypt all the blocks corresponding to $\mathsf{D}[\widehat{S}_\mathsf{f} \cup \widehat{S}'_\mathsf{f}]$ and the block corresponding to $\mathsf{T}[\tau]$ using the IV $v_i||i$ as described in the BSTORE.Build step, but after incrementing v_i for each block.
12. Upload the newly reencrypted blocks back to the server. Note that it is a subset of the blocks that were downloaded that are uploaded back, with their version numbers incremented by 1, and reencrypted.

- When op = BSTORE.read, the steps 1 through 6 from above are carried out, but setting $\mathsf{size}'_\mathsf{f} = 0$.
- When op = BSTORE.write, the behavior is the same as when op = BSTORE.update, except that the new file data is taken as input upfront, and no data is returned.
- When op = BSTORE.delete, the behavior is the same as when op = BSTORE.write, except that it takes $\mathsf{size}'_\mathsf{f} = 0$, and also, the record in $\mathsf{T}[\tau]$ is updated with $\sigma_\mathsf{f} = 0$ (indicating that there is no file).

3.2.3 Security Analysis

We sketch a proof of security that our construction is a secure realization of the deal blind storage functionality $\mathcal{F}_{\text{STORE}}$, for the adversary model in which the server is corrupted only passively. The proof follows the standard real/ideal paradigm in cryptography (see [29], for instance), and uses some of the standard conventions and terminology.

Roughly, the proof involves demonstrating a simulator $\mathcal{S}$ which interacts with a client only via the ideal functionality $\mathcal{F}_{\text{STORE}}$ (the ideal experiment), yet can simulate the view of the server in an actual interaction with the client in an instance of our scheme (the real experiment). The simulated view would be indistinguishable from the real view, even when combined with the inputs to the client (from an "environment"). Further — and this is the adaptive nature of our security guarantee — the inputs to the client at any point in either experiment can be arbitrarily influenced by the view of the server till then.

Before describing our simulator, we describe the main reason for security. Suppose the client makes a read access to a file for the first time. In the ideal experiment, the server learns this file's size and nothing about the other files. In the real experiment, the server sees a couple of downloads — a record in T and a set of blocks S_{f} in D. Thanks to the encryption, it is easy to enforce that the contents of these downloaded blocks give virtually no information to the server. But we need to ensure that the location of these blocks also do not reveal anything more than the size of this file. For instance, it should not be revealed how many files were added to T and D before this file was added. Intuitively, this is ensured by the fact the record in T and the pseudorandom subset S_{f} were determined by a process that was independent of the other files in the system. The only way the other files in the system influenced this process was in determining the subset $\widehat{S}_{\mathsf{f}} \subseteq S_{\mathsf{f}}$ of blocks that actually carried the data. However, recall that this subset is chosen randomly (or pseudorandomly) from the set of free blocks in S_{f}, provided such a subset of adequate size exists. Though the probability of $\widehat{S}_{\mathsf{f}}$ not existing depends on the number of occupied blocks (this probability is zero initially, and grows as more and more files get added to the system), every subset of S_{f} of that size is (virtually) equally likely to be $\widehat{S}_{\mathsf{f}}$. Hence the simulator can sample $\widehat{S}_{\mathsf{f}}$ from virtually the same distribution as in the real experiment, conditioned on an abort

not occurring. Thus the crucial argument in proving security boils down to showing that the probability of the client aborting in our protocol is negligible. We will give a standard probabilistic argument to prove this assumption.

Now we describe our simulator $\mathcal{S}$, which is in fact quite simple, and then discuss the main combinatorial argument used to show that the simulation is indistinguishable from the real execution. For the sake of clarity, we leave out some of the routine details of this proof, and focus on aspects specific to our construction.

The simulator interacts with the functionality $\mathcal{F}_{\text{STORE}}$ on the one hand, and interacts with the server on the other, translating each message it receives from $\mathcal{F}_{\text{STORE}}$ into a set of simulated messages in the interaction between the client and the server in our scheme.

1. When it receives the initial message from $\mathcal{F}_{\text{STORE}}$ with the system parameters, $\mathcal{S}$ can calculate the size of A; it simulates the blocks in A by picking uniformly random bit strings, with the version number in each block set to 0.

2. $\mathcal{S}$ initializes a table to map integers j to triples of the form $(\tau_j, S_j, \widehat{S}_j)$. Initially this table is empty. After $\mathcal{F}_{\text{STORE}}$ reports j^* accesses to $\mathcal{S}$, this table will have j^* entries. $\mathcal{S}$ also maintains the set $X = \cup_j \widehat{S}_j$, initialized to the empty set.

3. Subsequently, in access number j^*, when $\mathcal{S}$ receives a tuple $(\mathsf{op}, j, \mathsf{size})$ from $\mathcal{F}_{\text{STORE}}$, it proceeds as follows:

   (a) $\mathcal{S}$ first checks if $j > 0$. If so it sets $\tau_{j^*} = \tau_j$. Else it generates a random value for τ_{j^*}.

   (b) If $\mathsf{op} = \text{BSTORE.}\mathsf{delete}$, $\mathcal{S}$ sets $S_{j^*} = \widehat{S}_{j^*} = \emptyset$ and (if $j > 0$) sets $X = X \setminus \widehat{S}_j$.

   (c) If $j = 0$, and $\mathsf{op} \neq \text{BSTORE.}\mathsf{delete}$, $\mathcal{S}$ samples S_{j^*} with $|S_{j^*}| = \max(\lceil \alpha \cdot \mathsf{size} \rceil, \beta(k))$ uniformly, and a $\widehat{S}_{j^*}) \subseteq S_{j^*}$ conditioned on $|\widehat{S}_{j^*}| = \mathsf{size}$, and $\widehat{S}_{j^*} \cap X = \emptyset$; also, X is updated to $X \cup \widehat{S}_{j^*}$.

   If no such set $\widehat{S}_{j^*}$ exists, the simulation aborts.

   (d) If $j > 0$ and $\mathsf{op} = \text{BSTORE.}\mathsf{read}$, $\mathcal{S}$ simply sets $(S_{j^*}, \widehat{S}_{j^*}) = (S_j, \widehat{S}_j)$.

(e) If $j > 0$ and $\mathsf{op} = \text{BSTORE.}\mathsf{write}$ or $\mathsf{op} = \text{BSTORE.}\mathsf{update}$, $\mathcal{S}$ checks if $|\widehat{S}_j| > \mathsf{size}$ or not. If it is, then S_{j^*} and $\widehat{S}_{j^*}$ are set to random subsets of S_j and $\widehat{S}_j$ respectively, of appropriate sizes. Further, X is set to $X \setminus (\widehat{S}_j \setminus \widehat{S}_{j^*})$. Else, if $|\widehat{S}_j| \leq \mathsf{size}$, S_{j^*} and $\widehat{S}_{j^*}$ are set to random supersets of S_j and $\widehat{S}_j$ respectivly, subject to the conditions from item (3) above. Also, X is updated to $X \cup \widehat{S}_{j^*}$.

Again, if no such set $\widehat{S}_{j^*}$ exists, the simulation aborts.

(f) Finally, it creates the simulated view in which it downloads the record $\mathsf{T}[\tau]$, followed by a request to download the blocks $\mathsf{D}[S_{j^*}]$. In the case of operations other than BSTORE.read, it will also request to upload new versions of blocks indexed by $\widehat{S}_{j^*}$, with the block number incremented, and with fresh random bits in lieu of encrypted strings.

The simulation essentially maintains the indices of the two sets seen by the server, S_j which is downloaded, and the set $\widehat{S}_j$ which is uploaded back. It maintains consistency in terms of the pattern (same subsets are used if the same file is accessed) and the size of the files.

We note that there are two differences between this simulation and the real execution. Firstly, the simulated execution uses truly random strings instead of the outputs from Φ and Ψ. To handle this we can consider a "hybrid experiment" in which the real execution is modified so that instead of Φ, Ψ and Γ, truly random functions are used. By the security guarantees of the PRF, the PRP and the PRG (the last one applied after the others, so that the PRG's seed is completely hidden from the server), this causes only an indistinguishable difference.

The second difference is in aborting: the simulation aborts when it cannot find enough blocks which are not occupied by blocks of files that have been accessed (i.e., blocks listed in X), whereas the real protocol aborts when it cannot find blocks which are actually unoccupied (even considering files that have not yet been accessed). However, it can be seen that at any point in the hybrid execution above, as well as in the simulated execution, the sets S_{f} and $\widehat{S}_{\mathsf{f}}$ are identically distributed, *conditioned on a valid set $\widehat{S}_{\mathsf{f}}$ existing.* This is because, S_{f} is identically distributed in both experiments (being chosen independent of the files in

the system), and every subset of $\widehat{S}_{\mathsf{f}}$ of the right size is equally likely to be $\widehat{S}_{\mathsf{f}}$. However, the probability of a valid subset $\widehat{S}_{\mathsf{f}}$ existing is not necessarily the same in the two experiments.

To complete our proof, therefore it remains to show that the probability of the client or the simulator aborting is negligible. Before proceeding, we remark that our goal here is to give an asymptotic proof of security (showing that the error in security goes down as a negligible function of the security parameter). The concrete parameters from this analysis are overly pessimistic and an actual implementation can use less conservative parameters.

First, recall that in the simulation as well as in the modified real execution we are considering, the output of the PRG Γ on random seeds (used to define the pseudorandom subsets) have been replaced with truly random strings. Then we upperbound the abort probability as follows. Let d_0 be an upperbound on the total number of data blocks that will be occupied. Suppose there has been no abort so far, and a new file f of $\mathsf{size}_{\mathsf{f}}$ blocks is to be inserted into the system (either during the BSTORE.Build stage of during an BSTORE.update or BSTORE.write operation). Some $d \leq d_0$ out of the n_{D} blocks in D are filled. These blocks were filled by picking random subsets, and then within these subsets, choosing random subsets with free blocks. The net effect is of choosing a random subset of d blocks out of the n_{D} blocks. Now, when f is being inserted, we pick a random subset S_{f} of size $|S_{\mathsf{f}}| = \max(\lceil \alpha \cdot \mathsf{size}_{\mathsf{f}} \rceil, \beta(k))$. The *expected number* of occupied blocks within this set is $\frac{d}{n_{\mathsf{D}}} \cdot |S_{\mathsf{f}}|$. By a standard application of Chernoff bound,[5] the probability that more than $2\frac{d}{n_{\mathsf{D}}}|S_{\mathsf{f}}|$ blocks are occupied is $2^{-\Omega(|S_{\mathsf{f}}|)}$, provided $\frac{d}{n_{\mathsf{D}}}$ is upperbounded by a constant less than 1. Since $|S_{\mathsf{f}}| \geq \beta(k)$, this probability is $2^{-\Omega(\beta(k))}$, and since $\beta(k)$ is super-logarithmic in k (for e.g., $\log^2 k$), this probability is $2^{-\omega(\log k)}$ which is negligible in k. Thus except with negligible probability, of the $|S_{\mathsf{f}}|$ blocks chosen, at least $|S_{\mathsf{f}}|(1 - 2\frac{d}{n_{\mathsf{D}}}) \geq \alpha \mathsf{size}_{\mathsf{f}}(1 - 2\frac{d}{n_{\mathsf{D}}})$ are free. We shall pick $\alpha \geq \frac{1}{1-2\gamma}$ where γ is an upperbound on d/n_{D} so that the number of free blocks is at least $\mathsf{size}_{\mathsf{f}}$. Thus except with negligible probability, the client will not abort, when adding this file. By a union bound, the probability that it aborts remains negligible as long as it adds only polynomially many

[5] In choosing a random subset of blocks, the blocks are not chosen independent of each other. So in order to apply Chernoff bound, we first consider the experiment in which the blocks are selected independent of each other with the same fixed probability, so that the expected number of blocks chosen is, say $3/2d$. Then, by an application of Chernoff bound, except with $2^{-\Omega(n_{\mathsf{D}})}$ probability, at least d blocks are occupied. Now, in this experiment, we bound the probability that more than $2\frac{d}{n_{\mathsf{D}}}|S_{\mathsf{f}}|$ blocks in S_{f}, again using Chernoff bound. This probability is an upperbound on the corresponding probability in the original experiment.

files.

3.3 Searchable Encryption[6]

3.3.1 Definitions

Our syntax for a searchable encryption scheme is simpler than in [15], since all non-trivial operations are carried out by the client, and hence, there are no server side algorithms to be specified.

A searchable encryption scheme (a.k.a. SSE or searchable symmetric encryption scheme) consists of five probabilistic polynomial time procedures (run by the client), SSE.keygen, SSE.$\mathsf{indexgen}$, SSE.search, SSE.add and SSE.remove. These procedures interact with a "dumb" server which provides $\mathsf{download}$ and upload facilities to access blocks in an array, and also a simple file-system to lookup documents by identifiers. Looking ahead, in our implementation, the upload and $\mathsf{download}$ facilities are used to implement a blind-storage scheme which is used to store the keyword indices, and the file lookup facility is used to store the actual (encrypted) documents.

- SSE.keygen: It takes the security parameter as input, and outputs a key K_{SSE}. All of the following procedures take K_{SSE} as an input.

- SSE.$\mathsf{indexgen}$: It takes as input the collection of all the documents (labeled using document IDs), a dictionary of all the keywords, and for each keyword, an *index file* listing the document IDs in which that keyword is present.[7] It interacts with the server to create a representation of this data on the server side.[8]

- SSE.search: This procedure takes as input a keyword w, interacts with the server, and returns all the documents containing w.

[6] Joint work with Manoj M. Prabhakaran and Muhammad Naveed

[7] The index files can be generated by this algorithm if it is not provided as input.

[8] Typically, this would consist of a collection of (encrypted) documents, labeled by document indices (different from document IDs), and a representation of the index, which in our constructions will be stored using a blind-storage system.

- SSE.add: This procedure takes as input a new document (labeled by a document ID that is currently not in the document collection), interacts with the server, and incorporates it into the document collection.

- SSE.remove: This procedure takes as input a document ID, interacts with the server, and if a document with that ID is present in the server, removes it from the document collection.

Security Requirement. As in the case of blind-storage, we shall specify an ideal functionality, $\mathcal{F}_{\text{SSE}}$ to capture the security requirements of a dynamic SSE scheme. We note that the standard UC security in this case automatically ensures what has been called security against adaptive chosen keyword attacks (CKA2-security) for searchable encryption.

$\mathcal{F}_{\text{SSE}}$ accepts one of the following commands from the client (along with corresponding inputs, as described above), and behaves as follows. It initializes and maintains the document collection as specified by the commands, and answers the search queries correctly based on the current document collection. In addition it informs the server which command was received and reveals additional information as follows:

- On receiving the command $\mathcal{F}_{\text{SSE}}$.indexgen, it reveals to the server the multi-set consisting of the sizes of all the documents, and a system parameter specifying a (typically liberal) upperbound on the total number of keywords and (keyword, document) pairs supported by the system. The documents in the set are assigned serial numbers for future reference (sorted randomly). We shall refer to these documents as the *original documents* (as opposed to newly added documents).

- On receiving the command $\mathcal{F}_{\text{SSE}}$.add, it reveals to the server the size of the document. A serial number is assigned to this document (counting from the original documents and all the newly added documents) for future reference. Note that many documents with the same document ID can be deleted or added back, but the serial number is unique for each *version* of the document. For each keyword in the document, the set of serial numbers for all *newly added documents* (i.e., not the original documents) (possibly deleted) that have that keyword is also revealed to the adversary.

If a previous version of the document (i.e., a document with the same document ID) existed in the collection, its serial number is also revealed to the adversary.

- On receiving the command $\mathcal{F}_{\text{SSE}}$.remove, it reveals to the server the serial number of the document being removed from the collection. *It reveals no additional information.*

- On receiving the command $\mathcal{F}_{\text{SSE}}$.search, it reveals to the server the last instance the same keyword was searched on (or that it is being searched for the first time) and the set of serial numbers of all the documents that matched the search query. This includes deleted versions of the documents.

We highlight a few aspects of our security definition, compared to that in [15] and prior work. In [15], when a document is deleted, the scheme reveals the number of keywords in the document and further, for each keyword, upto two other documents that share the same keyword. This is the case even if that keyword is never searched on. In contrast, by our security requirement, if an original document is deleted, only the number of keywords in it that are searched can be revealed. Further, it is not revealed that a deleted document shared a keyword with another document, unless such a keyword is explicitly searched for.

We remark that our functionality explicitly reveals deleted versions of the documents in search results, but this information was revealed (implicitly) by the leakage functions in [15] as well, as the identifiers for each keyword in a deleted document is revealed and this information links the deleted documents to future searches on the same keyword (when the same identifier for the keyword is revealed).

3.3.2 Searchable Encryption from Blind Storage

In this section, we describe how an efficient dynamic searchable encryption scheme can be easily built on top of a blind-storage scheme.

The construction uses a blind-storage system, and a pseudorandom permutation Ψ' for mapping document IDs (with versioning) to pseudorandom document indices.[9]

[9]Long document names can be handled using a hash function, in much the same way long file names are handled in the blind-storage system. Here the hash collisions can be stored in an encrypted document of fixed size, indexed by the pseudorandom document index, kept outside the blind-storage system.

- SSE.keygen: It generates a key $\mathsf{K}_{\text{SSE}} = (\mathsf{K}_{\text{BSTORE}}, \mathsf{K}_{\mathsf{dID}})$ where $\mathsf{K}_{\text{BSTORE}}$ is generated by BSTORE.Keygen and $\mathsf{K}_{\mathsf{dID}}$ is a key for the PRP Ψ'.

- SSE.$\mathsf{indexgen}$:

  1. Firstly, for each document d, assign a pseudorandom ID $\eta_d^0 = \Psi'_{\mathsf{K}_{\mathsf{dID}}}(0||\mathsf{id}_d)$, where id_d is the document ID, to which a single bit is prepended (to indicate that this is a document already present while creating the initial index).
  2. For each keyword w, construct an *index file* with file-ID index_w^0 that contains η_d^0 for each document d that contains the keyword w. No specific format is required for the data in this file; in particular, it could contain a "thumbnail" about each document in the list.
  3. Next, initialize a blind-storage system with the collection of all these index files (using BSTORE.Build).
  4. Also, (outside of the blind-storage system) upload encryptions of all the documents labeled with their pseudorandom document index η_d^0.

- SSE.remove: To minimize the amount of information leaked, we rely on a lazy delete for documents that are already present during SSE.$\mathsf{indexgen}$ (but not for the documents that were added later).

  Given a document ID id_d, proceed as follows:

  1. First check if a document with index $\eta_d^0 = \Psi'_{\mathsf{K}_{\mathsf{dID}}}(0||\mathsf{id}_d)$ exists, and if so delete it (using the file system interface of the server). The index files are not updated for the keywords in this document right away, but only during a subsequent search operation (see below).
  2. Else (if there was no file with index η_d^0), check if a document with index $\eta_d^1 = \Psi'_{\mathsf{K}_{\mathsf{dID}}}(1||\mathsf{id}_d)$ exists. If it does not, return a status message to indicate this. If it does exist, proceed as follows:

     (a) retrieve document indexed by η_d^1, and then delete it.

(b) for each keyword w in the document, use the blind-storage BSTORE.update facility to update the index file with file-ID index_w^1 to remove η_d^1 from it.

- SSE.add: To add a document d to the document collection, first call SSE.remove to remove any earlier copy of a document with the same document ID. Then proceed as follows:

  1. Firstly assign it a pseudorandom document index as $\eta_d^1 = \Psi'_{\mathsf{K}_{\mathsf{dID}}}(1||\mathsf{id}_d)$. Note that the bit preprended to id_d in this case is 1, to indicate that this is a document that was added after creating the initial index.

  2. Then, for each keyword w that appears in this document, use the BSTORE.update facility of the blind-storage scheme to update the file with file-ID index_w^1 to include η_d^1. Note that the file-IDs used in this phase are different from the ones used during SSE.indexgen. Also, note that the BSTORE.update operation will create a file if it does not already exist.

- SSE.search: Given a keyword w, retrieve the two index files with file-IDs index_w^0 and index_w^1 from the blind-storage system: for the file index_w^1 the BSTORE.read operation is used, and for the file index_w^0, the first stage of BSTORE.update operation is used. All the documents containing the keyword w have their document indices listed in these two files. Attempt to retrieve all these documents from the server; but some of the documents listed in index_w^0 could have been deleted. Complete the BSTORE.update operation on the file index_w^0 to remove the deleted files from its list.

3.4 HIE Interface

The HIE interface is performed with two web services that use messaging format corresponding to a subset of XDS.b transactions: Document Registry and Document Repository. The definition of these two web service gateways is given in Figure 3.3, and explained in next two sections.

Document Registry:

- `RegistryStoredQuery(Keywords) : OIDs`

Document Repository:

- `ProvideAndRegisterDocumentSet(Keywords, Documents) : Success/Failure`
- `RetrieveDocumentSet(OIDs) : Documents`

Figure 3.3: Inteface definition for web service gateways.

3.4.1 Document Registry

The role of document registry gateway is to map a set of keywords to a set of document references that contain these keywords. In the scenario that we implemented as described in 4, this service uses patient's first name and family name as the keywords. The XDS.b profile uses globally unique object identifiers (OIDs) for documents, which can also be used as keys in the secondary storage. Within the XDS.b profile, this transaction is known as Registry Stored Query (ITI-18).

Document Registry web services invokes the HIE layer of BlindStorage with keywords as parameters, which are then passed down the stack to the searchable encryption layer. Upon the completion of BlindStorage access operation, object identifiers are returned to the HIE layer, translated to SOAP response message, and passed back to the Document Consumer.

3.4.2 Document Repository

The Document Repository gateway serves both the Source and the Consumer; hence, it must implement two separate methods. The first method corresponds to Retrieve Document Set (ITI-43) transaction and is rather unsophisticated. Since at this stage keyword look-up has already been performed and Document Consumer knows the set of document OIDs to retrieve, the gateway only needs to pass these identifiers down to the BlindStorage module as a simple Get request. This request is then relayed to the secondary storage that contains bodies of the documents in encrypted form, and is then passed back to the Consumer.

The second service implemented by Document Repository corresponds to XDS.b Provide and Register Document Set (ITI-15) transactions. In this case, the gateway is responsible for accepting a set of Continuity of Care Documents of a particular patient, extracting patient's name, translating them into keywords, and invoking the HIE layer of BlindStorage with a set of (keyword, document) pairs. This request is then translated into the BSTORE.write operation, BlindStorage index is updated with a new mapping between the keyword and document reference, and the CCD document is encrypted and stored in the secondary storage.

Chapter 4

Implementation

Since our HIE architecture consists of two major components - the interoperable, web-services based side that directly interfaces with hospital systems, and the secure, efficient BlindStorage part, it was also prudent to consider different technology choices for their implementation. The cryptographic storage and searchable encryption components need to perform a significant amount of bit-level manipulation, efficient encryption and decryption, and pointer arithmetic, and we decided to implement it in C++. On the other hand, the HIE side had to be based on web services per our general design and goal to follow XDS.b profile as closely as possible, and would benefit from a higher-level language and readily available frameworks to generate boilerplate code. The Java EE platform provides a convenient JAX API along with a set of tools to generate web service definition language (WSDL) descriptor files directly from the source code, and therefore, we chose to implement the HIE side in Java.

Besides these advantages, intentionally selecting different technologies for implementing an advanced encryption technique and a standards-based HIE serves as a basis of measuring the transparency of our solution to the HIE context for the purpose of evaluation. We will consider the project successful when a document producer can upload a valid CCD document to the HIE using an XDS.b-like interface, and a document consumer can retrieve it using only registry and repository services.

4.1 BlindStorage

The core BlindStorage code is implemented as a set of C++ object, built around T and D tables. The format of their records are shown in Figures 4.1 and 4.2.

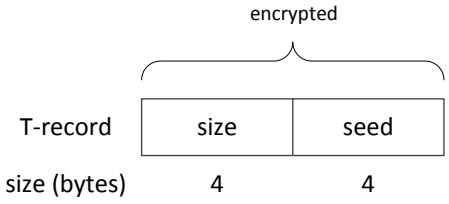

Figure 4.1: Record format in the T table.

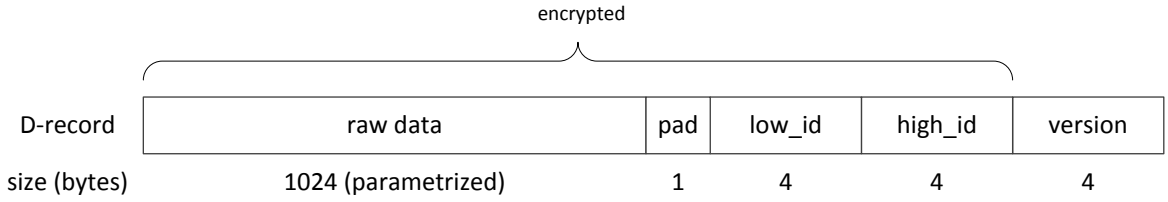

Figure 4.2: Record format in the D table.

Each T-record that corresponds to a file contains file size (in the number of blocks), and a 32-bit PRG seed, encrypted with $\mathsf{K}_{\text{BSTORE}}$. Whenever file blocks need to be accessed from D table, the seed is used as a parameter to the `srand()` function, and the `rand()` PRG is called $4 * size$ times. We set the expansion parameter α to 4 as it provides nearly optimal balance between the probability of failure (not being able to find free blocks), and overhead.

There are two representations of T and D data structures in memory . First, during the initial construction phase of BlindStorage `TDisk` and `DDisk` are located in local memory and can be accessed directly. After the initial Build operation is completed, these structures are saved to the disk (or uploaded to a remote key-value server) and are no longer accessed directly. Instead, we follow the proxy design pattern[1], and use two other objects - `TFile` and `DFile`. These do not store data in memory, but instead access records on demand. In a local implementation of BlindStorage, this is done by seeking to a correct position in the file; in the distributed version, it sends a `GET(index)` request to the key-value store.

Each T and D record has a pair of method to encrypt and decrypt its data with AES-128 (`Crypto++` library is used).

The BlindStorage code is compiled as a local executable, and accepts queries through a command line interface. Java web services in return run these processes through an `execute()` method, and communicate via standard input and standard output streams. Although this way of inter-process communication is suboptimal in terms of both performance

[1] http://www.oodesign.com/proxy-pattern.html

and general software engineering practices (Java Native Interface (JNI) being the alternative), it allows effortless testing of individual layers, as BlindStorage executables can be launched and tested directly from the command line.

4.2 Web Services

Our goal of implementing web services was to remain as close to standards-based HIE as possible while keeping the scope of this project manageable. One of the most comprehensive open HIE frameworks with source code readily available is NHIN CONNECT, and we studied what features is supports and what technologies it is built on, to replicate a scaled-down version of a production system. The Enterprise Service Components in the CONNECT solution provide implementation of XDS.b Document Registry and Repository actors which is claimed to be one of "critical enterprise components required to support electronic health information exchange" [2]. Examining the underlying technology behind CONNECT shows that it is built using Java Enterprise Edition (JEE), and in particular, Java API for XML Web Services (JAX-WS).

This investigation gives evidence that we will accomplish our goal of staying close to the requirements for a standards-based HIE system if we use the same technologies for our implementation.

We implemented the four actors that correspond to XDS.b actors using Java EE and JAX-WS on Apache Tomcat 7 with Apache Axis. Document Repository and Document Registry actors are represented as web services. On the other hand, Document Source and Document Consumer actors are connection initiators, and they are implemented as standalone Java application that use web services client library.

The Document Source application accepts a set of CCD documents as its input, processes them to locate XML `<patient>` node, and extracts patient's first and last name from this node. These two names are then merged into a single keyword, and transmitted to Document Registry along with CCD contents. Document Consumer first invokes Document Registry to look-up document object identifiers that belong to a particular patient, and then contacts

[2] http://www.connectopensource.org/about/what-is-connect

Document Repository to retrieve these documents.

4.3 Evaluation

The National Institute of Standards and Technology (NIST) published a series of testing tools "for promoting the adoption of standards-based interoperability by vendors and users of healthcare information systems", which includes validation tools and sample CCD documents. For the evaluation of our project, we downloaded a representative Healthcare Information Technology Standards Panel (HITSP) summary document using HL7 CCD from the `MeaningfulUse_Examples_Jan2011`[3] data set. Since these documents were explicitly published for testing conformance of HIEs to CDA standards, we consider this data set to be ideal for our purpose of evaluation of correctness and transparency.

We successfully transmitted the test document through our HIE implementation, and verified correctness. Since neither Document Source nor Document Consumer actors were aware of BlindStorage and only communicated with Registry and Repository services, we also successfully verified correctness.

[3] http://xreg2.nist.gov/cda-validation/downloads.html

Chapter 5

Discussion

In this chapter, we will explore a number of topics that are relevant to HIEs in general and our implementation in particular.

5.1 Push and Pull Methods

Health information exchanges use one of two exchange methods — "push" technology, when the document source automatically delivers clinical data without a prompt, and a "pull" method (also known as portals) that requires physicians to search for the data they need [30]. We observe that our implementation acts as a hybrid of these two: on one hand, consumers do not directly communicate with sources and therefore sources must proactively upload records to the repository as in the "push" model. On the other hand, searchable encryption provides an option for consumers to perform basic queries that is more typical of a "pull" method.

We first discuss the advantages of both approaches. One problem with "push" methods is that typically there must be a pre-existing patient-physician relationship to enable this kind of delivery. With data being encrypted in the repository, document sources could push the record to the HIE in advance, and instead only control access through key distribution. The document consumer benefits in a similar way, as it only needs to contact the HIE storage for the document instead of locating and connecting to the source.

Unfortunately, in additional to getting some benefits of both methods, we also receive their limitations. Before a document is pushed out to the HIE, the document source must know which keywords will be considered during subsequent queries, and update the index with these keywords. To provide functionality of flexible searches the source must either

precompute many keyword combinations or subdivide the keyword namespace into a non-flat structure, or our architecture would need to be extended with multiple BlindStorage systems that would represent different indices. Document consumers are similarly limited by both the type of queries they are allowed to make (keyword searches), and flat keyword domains.

5.2 Efficiency

Another topic of discussion is whether efficiency (or performance) is important for HIEs. Since no summary statistics on HIE document rate were readily available, we will not speculate on specific numbers. However, it is not unreasonable to assume that virtually every nontrivial cryptographic scheme that supports advanced functionality is bound by some performance bottleneck — whether it is the number of keyword combinations supported, worst case access time, or storage overhead. Any index-based searchable encryption scheme, including ours, supports a trivial extension of functionality by building another layer on top of keywords. For example, document repositories could natively support basic keyword conjunctions by building the index from keyword pairs in the first place. This increases the number of keywords quadratically.

A simple example like this demonstrates that in any case, better efficiency is preferred as it can be traded in for other features — be it functionality, fewer number of servers, or simpler maintenance. Therefore we consider that performance advantages of BlindStorage as compared to other cryptographic schemes remain meaningful and useful even if we cannot directly measure the impact of efficiency.

5.3 Key Infrastructure

A major component of any distributed system with different levels of access is the key infrastructure. In-depth exploration of key infrastructure for a HIE is a topic of a project of no lesser size, but we can make several remarks.

In our design, we assume that there exists only one set of symmetric keys for all sources and consumers that participate in the exchange. While we did reject public-key searchable encryption on the basis of insufficient keyword protection, this does not mean that a public key infrastructure could not be used for dissemination of symmetric keys to the index. Additionally, searchable encryption that is built on top of BlindStorage uses different keys for keyword protection and encryption of medical records, which implies that there could be a variety of combinations of keying schemes.

5.4 Who Runs The HIE?

The questions of making the business case for an entity and justifying its existence can be easily overlooked if we only look at the architecture from the technical point of view. It has been claimed that arguably superior technologies — such as a distributed social network Diaspora — never made an impact due to lack of either a business model or some other support.

In case of health information exchanges, there exists enough evidence that states and even federal government are willing to run such an infrastructure as a public service. Nevertheless, it is worthy to consider other alternatives.

In fact, we claim that the cryptographic nature of our HIE design opens new opportunities for private businesses to provide the services of Document Registries and Repositories. BlindStorage has already been explicitly designed to support a minimal key-value storage back-end (which we call secondary storage), and since the index is never seen in plain-text by the server, it is entirely plausible that a private enterprise that is unaffiliated with HCOs can operate a limited HIE — an option that is not possible with traditional HIEs.

Chapter 6

Conclusion

In this thesis we considered how a health information exchange can be implemented with a cryptographic storage scheme that supports keyword search. We discussed common architectures of HIEs, why they are important, some of the driving factors behind the adoption of information technology, and how cryptography can be of assistance in the healthcare domain. While cryptography can be a power tool, it imposes severe limitations on functionality, which is the reason for many approaches to data preprocessing and indexing prior to encryption.

We introduce BlindStorage, a cryptographic file system that is easily used as a building block to implement searchable encryption. In its turn, searchable encryption provides an interface a domain-specific application, which in our case is a gateway that connect a greater distributed HIE to the secure core. One of the main contributions of this paper was the demonstration how an advanced encryption technique can be deployed in a context close to the requirements for a standards-based HIE, which was evaluated for correctness and end-to-end transparency with a test CCD dataset. Finally, other secondary aspects were discussed that are relevant to health information exchanges and our implementation in particular.

References

[1] M. Blaze, "A cryptographic file system for unix," in *Proceedings of the 1st ACM conference on Computer and communications security*. ACM, 1993, pp. 9–16.

[2] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano, "The design and implementation of a transparent cryptographic file system for unix," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Citeseer, 2001, pp. 199–212.

[3] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A stackable vnode level encryption file system," Citeseer, Tech. Rep., 1998.

[4] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043566 pp. 85–100.

[5] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 79–88.

[6] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315318 pp. 598–609.

[7] C. Wang, S. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *Computers, IEEE Transactions on*, vol. 62, no. 2, pp. 362–375, 2013.

[8] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptography Workshops*, 2010, pp. 136–149.

[9] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ser. STOC '87. New York, NY, USA: ACM, 1987. [Online]. Available: http://doi.acm.org/10.1145/28395.28416 pp. 182–194.

[10] R. Ostrovsky, "Efficient computation on oblivious rams," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, ser. STOC '90. New York, NY, USA: ACM, 1990. [Online]. Available: http://doi.acm.org/10.1145/100216.100289 pp. 514–523.

[11] B. Pinkas and T. Reinman, "Oblivious ram revisited," *Advances in Cryptology–CRYPTO 2010*, pp. 502–519, 2010.

[12] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *Network and Distributed System Security Symposium–NDSS 2012*.

[13] S. Kamara, C. Papamanthou, and T. Roeder, "CS2: A Searchable Cryptographic Cloud Storage System," MSR Tech Report no. MSR-TR-2011-58, 2011.

[14] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382298 pp. 965–976.

[15] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography Workshops*, 2013.

[16] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004, pp. 506–522.

[17] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. Skeith, "Public key encryption that allows pir queries," *Advances in Cryptology-CRYPTO 2007*, pp. 50–67, 2007.

[18] J. Benaloh, M. Chase, E. Horvitz, and K. Lauter, "Patient controlled encryption: ensuring privacy of electronic medical records," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, ser. CCSW '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1655008.1655024 pp. 103–114.

[19] K. D. Mandl, W. W. Simons, W. C. Crawford, and J. M. Abbett, "Indivo: a personally controlled health record for health information exchange and communication," *BMC medical informatics and decision making*, vol. 7, no. 1, p. 25, 2007.

[20] S. Narayan, M. Gagné, and R. Safavi-Naini, "Privacy preserving ehr system using attribute-based infrastructure," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, ser. CCSW '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1866835.1866845 pp. 47–52.

[21] J. A. Akinyele, M. W. Pagano, M. D. Green, C. U. Lehmann, Z. N. Peterson, and A. D. Rubin, "Securing electronic medical records using attribute-based encryption on mobile devices," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046628 pp. 75–86.

[22] Presidents Council of Advisors on Science and Technology, "Realizing the full potential of health information technology to improve healthcare for americans: The path forward," Executive Office of the President, 2010.

[23] R. Cothren and L. Westberg, "Connect: Architecture overview," CONNECT Seminar, 2009. [Online]. Available: http://www.connectopensource.org/sites/connectopensource.org/files/CONNECT_ArchitectureOverview.pdf

[24] G. vant Noordende, "Controlled dissemination of electronic medical records," in *Proceedings of the 2nd USENIX conference on Health security and privacy*. USENIX Association, 2011, pp. 13–13.

[25] World Wide Web Consortium, "Extensible markup language (xml) 1.0 (fifth edition)," W3C Recommendation, Nov. 2008. [Online]. Available: http://www.w3.org/TR/REC-xml/

[26] State of Illinois Office of Health Information Technology, "Request for grant applications for health information exchange white space program," Sep. 2012. [Online]. Available: http://www2.illinois.gov/gov/HIE/Documents/White%20Space%20RGA.pdf

[27] World Wide Web Consortium, "Xml encryption syntax and processing," W3C Recommendation, Dec. 2010. [Online]. Available: http://www.w3.org/TR/xmlenc-core/

[28] T. Jager and J. Somorovsky, "How to break xml encryption," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046756 pp. 413–422.

[29] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," Electronic Colloquium on Computational Complexity (ECCC) TR01-016, 2001, previous version "A unified framework for analyzing security of protocols" available at the ECCC archive TR01-016. Extended abstract in FOCS 2001.

[30] S. Massengill, "Can portals deliver?" ADVANCE for Healthcare Information Executives, Mar. 2009.