Securing Remote Data Stores

Design and Implementation of an Encrypted Data Store

Diplomarbeit zur Erlangung des akademischen Grades eines
Magisters der Sozial- und Wirtschaftswissenschaften

Eingereicht an der Johannes Kepler Universitt Linz

Institut für Wirtschaftsinformatik

Data & Knowledge Engineering

Eingereicht bei: o.Univ.-Prof. Dr. Michael Schrefl

Betreuende Assistenten: Mag. Katharina Grün, Mag. Michael Karlinger

Walter Dorninger

December 7, 2005

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, daß ich die Diplomarbeit mit dem Titel - Securing Remote Data Stores - selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, im December 7, 2005

Meiner Familie gewidmet.

Acknowledgment

I would like to thank...

o.Univ.-Prof. Dr. Michael Schrefl for making it possible to perform this study in a modern and interesting scientific area and the interresting discussions.

Mag. Katharina Grün and Mag. Michael Karlinger for their valuable help and advices and their incredible feedback for this work.

My parents for their mental support.

My grandparents for her unconditional love.

My brothers Albert and Thomas for the fun I have with them and their help in solving problems.

My friends, especially those I have neglected during this work.

My colleges for taking over work I was not able to complete because of time constraints during the development of this thesis.

Kurzfassung

Der anhaltende Trend, Daten auszulagern steigert die Menge an Datenbeständen, welche bei externen IT-Dienstleistern gespeichert werden. Bei der Auslagerung von Daten werden oft hohe Risiken hinsichtlich des Datenschutzes eingegangen. Dies ist vor allem ein Problem für Finanzdienstleister, Institutionen im Gesundheitsbereich, sowie für die öffentliche Verwaltung. Während durch Nutzung externer IT-Dienstleister die Kosten für IT-Services drastisch verringert werden können, ist mangelndes Vertrauen in externe IT-Dienstleister oft der Grund, IT-Services nicht auszulagern.

Bei der Auslagerung von Daten müssen sich Unternehmen oft auf vertragliche Vereinbarungen verlassen, welche jedoch lediglich den Umgang mit den ausgelagerten Daten, sowie mögliche Konsequenzen bei Verletzung der Vereinbarungen definieren. Diese Vereinbarungen, auch Service Level Agreements (SLAs) genannt, können die Sicherheit der Daten nicht garantieren, da Angestellte der IT-Dienstleister meist ungehinderten Zugang zu diesen Daten haben. Die gespeicherten Daten können zwar verschlüsselt werden, um Abfragen durchzuführen werden diese Daten jedoch wieder am Server des ASPs entschlüsselt und sind damit im Speicher des Servers unverschlüsselt verfügbar.

SemCrypt ist ein Forschungsprojekt welches sich mit Techniken zur Abfrage und Änderung von XML Dokumenten auf externen unsicheren Servern beschäftigt. Die grundlegende Idee ist, Daten ausschließlich auf einem sicheren Client zu ver- und entschlüsseln. Da Abfragen auf verschlüsselte Daten ohne Zusatzinformationen nur eingeschränkt möglich sind, müssen die Abfragen am Client mit Hilfe von Indexstrukturen und Metadaten durchgeführt werden.

Diese Diplomarbeit befaßt sich mit dem Datenspeicher von SemCrypt, welcher SemCrypt Store genannt wird. SemCrypt Store soll Änderungen sowie Abfragen von verschlüsselten Daten ermöglichen. Der Datenspeicher soll sowohl die interne Struktur als auch die Daten selbst verschlüsseln. Hierfür werden XML Dokumente in Fragmenten gespeichert. Die einzelnen Fragmente werden mit einem sicheren Verschlüsselungsalgorithmus verschlüsselt. SemCrypt Store besteht aus zwei Komponenten. Die erste Komponente mit dem Namen Storage Engine läuft

auf dem Client und kümmert sich um Ver- sowie Entschlüsselung der Daten. Die zweite Komponente namens Storage Provider stellt einen einheitlichen Datenzugriff für die Storage Engine zur Verfügung.

SemCrypt Store ermöglicht das Speichern, Abfragen, Verschlüsseln sowie Entschlüsseln von Daten ohne diese am Server preiszugeben.

Abstract

The trend towards outsourcing increases the number of documents stored at external Application Service Providers (ASPs). However, this storage approach raises privacy and security concerns because ASPs cannot be trusted with respect to privacy. This is especially a problem for organizations maintaining sensitive data like financial-, health care- or government data. While the use of ASPs has proven beneficial at a business level because costs for IT services can be reduced significantly, the lack of trust is often the reason for organizations to not outsource their data stores and thus having to maintain expensive data stores in-house.

When outsourcing data, organizations have to rely on service level agreements that can only define how the data is maintained by the storage provider. These service level agreements cannot guarantee the privacy and security of data because employees of the service provider still have access to the outsourced data. Encryption is a possible approach for data protection but requires specific techniques for querying this data.

SemCrypt is a research project that aims at querying and updating encrypted XML documents stored at external untrusted servers. The principal idea is to encrypt and decrypt data only at the client in a trusted environment and use the database of an ASP to store the encrypted data. Because querying encrypted data has very limited capabilities, the client has to perform query processing by exploiting the structural semantics of XML documents as well as index structures and meta data, making it possible to efficiently process queries. SemCrypt thus enables to query and update encrypted XML documents on untrusted servers while ensuring privacy.

This thesis concentrates on the storage layer of SemCrypt, called SemCrypt Store. The main target is to provide a secure remote data store that can operate in an untrusted environment but still enables query processing. The security requirements for the SemCrypt Store are to encrypt the data and hide the associations inside XML documents. These requirements are fulfilled by storing XML documents in an identifier-value based approach using well proven encryption techniques to hide the content and structure of XML documents. SemCrypt Store consists of

two components. The first one is the Storage Engine which runs on the client and performs encryption and decryption of values that have to be stored by the SemCrypt Store. The Storage Provider is the second component that operates on the server and provides a unified transactional database access to arbitrary data sources for the SemCrypt Store.

SemCrypt Store enables to encrypt, store, retrieve and decrypt data without revealing any information about data stored at the storage provider, thus making it possible to employ untrusted data stores for storing sensible data.

Contents

Chapter 1

Introduction

Contents

This chapter describes the current situation in the Application Service Provider (ASP) business and presents an example scenario for a secure data store. The chapter also discusses today's security problems in the ASP business. Furthermore the current approaches in the field of secure data stores are described. The section *The SemCrypt approach towards secure data access* outlines the basic idea and architecture of SemCrypt. In the section *Objective of this thesis* the purpose of this master thesis is explained. The reader of this chapter should get an understanding of the motivation of this master thesis and the related project SemCrypt.

1.1 Motivation

Storing large amounts of data not only requires a significant investment in hardware, it also places a burden on the in-house information technology (IT) systems. To effectively keep up with the growing volume of data, today's enterprises take advantage of the wide range of specialized IT services provided by ASPs.

When outsourcing data services, companies can reduce their focus on information technology and instead can concentrate on enhancing their core business. This strategy reduces the total cost of ownership for IT services because an ASP typically provides the same or similar services to other companies, thus making it possible to offer the same services for a lower price.

Many organizations operate on sensitive data, e.g. in the financial industry, health industry, insurances and government organizations. Despite security and privacy risks such organizations do not want to renounce outsourcing opportunities since they store their sensitive data at ASPs. As an example Weinstein [34] mentions a case where a Pakistani subcontract worker threatened to post U.S. patients medical data on the Web if claimed back pay was not forthcoming.

The above example makes clear that outsourcing data raises the need for secure remote data stores providing the full functionality of conventional databases but without the risk of exposing sensitive data.

1.2 Sample scenario

As an example for the need of a secure remote data store, think of a software company called Enigma that is primary selling software licenses. Assume that Enigma wants to outsource the data store of license contracts with its customers. Instead of storing the license contracts in-house, Enigma wants to employ an ASP to manage the license contracts data store i.e. archiving old license contracts and performing frequent backups. Outsourcing these activities enables Enigma to concentrate on its customer relationships instead of concentrating on the data store of its contracts. Still Enigma needs to access this data for querying and maintaining the license contracts. Beside operational maintenance tasks such as backing up data, the ASP also offers the company's sales force access to the license contracts from all over the world 24 hours a day.

Figure 1.1 displays this scenario. Enigma sends requests for contract data and the ASP returns the requested information.

The contracts of Enigma contain the following information:

- *Contract identifier* - The contract identifier is used to uniquely identify a contract.
- *Licensee name* - This is the name of the licensing company.
- *Price* - The license fee the licensee has to pay frequently, granting him rights to use the respective software.

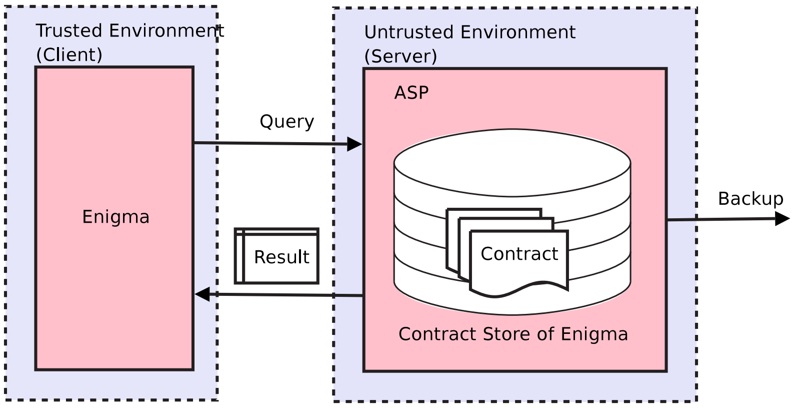

Figure 1.1: Sample outsourcing scenario

- *Payment information* - The payment information containing credit card id, card name and valid-thru date is stored along with the contract and can be used to charge the licensee on a frequent basis without having to request this information every time a payment is due[1].
- *Terms and conditions* - The terms and conditions that are part of the contract.

Figure 1.2 shows the data structure of the contract with id ”478”. The licensee in this contract is ”basf” and ”basf” has to pay a license fee of ”3000” Euros. The payment information in the contract stores the credit card type which is ”visa” the credit card id ”1234” and the expiration date of the the credit card ”12/12/05”.

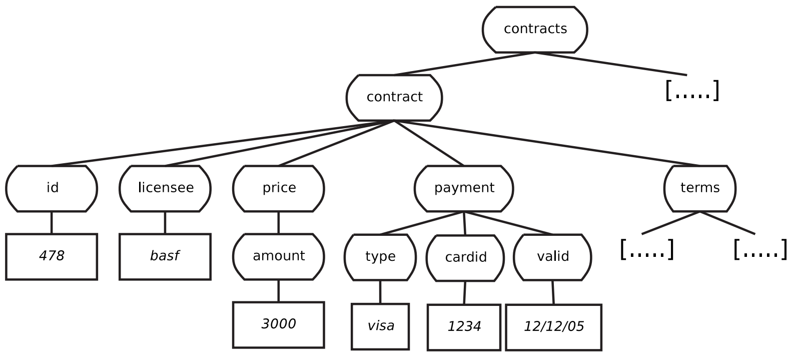

Figure 1.2: Exemplary license contract

[1]Note that putting the payment information into the contract data is not a very good style from a datamodelling perspective but is modeled this way to keep the example simple.

1.3 Problem definition

While the use of ASPs has proven beneficial at a business level because of significant cost reductions, integrating ASPs into the network and processing environment of client organizations raises obvious security concerns, especially when outsourcing sensitive data.

"According to a survey performed by Osterman Research security continues to be a primary obstacle to companies choosing to outsource business processes. 86.6 per cent of respondents felt that data security is a significant concern when considering outsourced service providers, and more than half of the respondents are not outsourcing their storage and archiving requirements due to concerns around third-party access to their data" [10].

Outsourced data is exposed to the following risks:

- *Eavesdropping* can be performed over communication channels like the Internet and in wide area network environments where ASPs route portions of their network traffic through insecure data channels. Eavesdropping can also be performed in local area network environments. Insiders with access to the physical wiring have the possibility to illegally view data and also may install network sniffers to eavesdrop on network traffic.

- *Data theft* is primarily perpetrated by employees who directly work with, or have access to valuable data. Some organizations are concerned about administrators of data stores because they typically have all privileges and are able to access and manipulate all data in the data store. These organizations feel that the employees should merely administer the data store, but should not be able to see the data in the data store. Some organizations are also concerned about the concentration of privileges in one person, and would prefer to partition the administrators function, or enforce two-person rules. While most organizations have implemented firewalls and intrusion detection systems, very few focus on one of the biggest threats to their organization, which in fact is the average employee who steals proprietary data for personal gain or use by another company. The damage caused by data theft can be considerable with today's ability to transmit very large files via email, web pages, USB devices, DVD storage and other devices. New threats are introduced by removable media devices that are getting smaller and have increased capacity.

- *Data tampering* - It is intrinsic for distributed environments that a malicious third party can compromise integrity by tampering stored data.

- *Data Loss* is another security problem which arises when storage devices containing sensitive data are lost. This typically happens during the transport of data e.g. the transport of data to its backup location. customers.
- *Privacy* means that organizations need to be sure that their data is properly protected from the ASP's other customers which may be sharing the same data store and processing environment.

Despite efforts to protect data stores, customers still often do not fully trust ASPs. This is especially true for the financial industry and government areas dealing with sensitive data. In many cases these organizations store sensitive data at ASPs. When an organization contracts with an ASP, it needs to be able to fully utilize the ASP's services while not putting its own private data at risk. Therefore the ASP has to protect data against the following forms of unauthorized access:

- *Unauthorized access from outside* - Protecting data from outside access can be achieved by firewalls and physical access control mechanisms.

  Example 1 (Data separation) *For the scenario mentioned in section 1.1 this means that the ASP has to protect Enigma's license contracts from access by other companies which also use the ASP to store data.*

- *Unauthorized internal access* - This includes protecting data from unauthorized employees working for the ASP and ensuring that only authorized employees of the ASP can get in touch with customer data.

  Example 2 (Internal data access) *The ASP has to ensure security such that, no one except authorized sales representatives have access to Enigma's license contracts. This also includes that the ASP has to ensure that even system operators and administrators handling data at the ASP or other employees don't have access to this data. The administrators could abuse the credit card number used in the license contracts or even could steal license contracts for a competitor of Enigma.*

Further, in situations where sensitive data is stored at an ASP, the contracting organization needs to know that the ASP is safeguarding the data to the same or even higher standards than the organization itself. Today arrangements defined in Service Level Agreements (SLAs) are very common to avoid unauthorized access of data. But SLAs can only make sure that certain procedures are enforced when sensitive data is handled by an ASP. Typically SLAs between an ASP and a contracting organization are used to define the level of security that has to be applied to the data stores.

Since unauthorized data access by employees cannot be avoided with SLAs many ASPs have recognized the need to find ways to improve the security of their data stores. Data encryption is a crucial step in securing data stores. Encryption is the primary mechanism to ensure the security of sensitive data. It not only protects data if it was lost or stolen, it also reduces the possibility of security breaches. A technical solution is required that prevents risks like those mentioned above but still provides full functionality of common data stores e.g. databases.

All data has to be encrypted and must never be decrypted at the ASP such that neither intruders nor employees of the ASP have access to plain data. Cipher algorithms that are provided by standard databases like e.g. Oracle or DB2 are not sufficient for this approach because these data stores have to decrypt data for query processing which violates the demand of performing no decryption at the ASP. This raises the need of an encrypted remote queryable data store that is operated by the ASP and is accessed by the client.

Example 3 (Encrypting data) *If all license contracts of Enigma are stored encrypted at the ASP, Enigma does not have to fear that the data can be misused by the ASPs employees.*

1.4 State of the art and current approaches

This section describes the current approaches towards encrypting and accessing data:

- *File encryption*[1] - Many operating systems provide the ability to encrypt files and to store them in a conventional filesystem [24] [7]. However, every time a query is performed or data is manipulated this approach requires to (1) retrieve whole files from the ASP, (2) decrypt them to process the query or manipulate the data and (3) finally transfer the data back to the ASP. Transferring whole files from the ASP to the client and then back to the server leads to performance losses. Furthermore many filesystems require to perform encryption and decryption at server side thus gaining security vulnerabilities.

- *Pattern matching in encrypted text*[1] - There are approaches that suggest to store data encrypted and provide mechanisms to search directly in the encrypted data without having to decrypt the data [14]. The problem of these approaches is that the query abilities are very limited and therefore these solutions do not provide full functionality of common databases. Furthermore if the cipher algorithm becomes insecure the whole approach does not work anymore because the pattern matching relies on the specific cipher algorithm.

- *Database encryption* - There are many approaches that aim to encrypt the data in data stores. Hacigümüs et. al. [19] propose a way to store data encrypted and to query the encrypted data. The strategy of this approach is to perform as much as possible of the query processing at the ASP without having to decrypt data. All remaining operations that require data in plain text are performed at the client side. The technique deploys a "coarse index", which allows partial server side execution of an SQL query. The result of this query is sent to the client. The correct result of the query is found by decrypting the data and executing a compensation query at the client side.

  Unfortunately this approach has not proven to be secure if the schema is known and it is also proven semantically insecure due to possible frequency analysis of repeated storage patterns [17].

1.5 The Semcrypt approach towards secure data stores

SemCrypt is a research project funded by FIT-IT which is an initiative of the Bundesministerium für Verkehr, Innovation und Technologie (BMVIT). The project is driven by the Department of Business Informatics - Data and Knowledge Engineering Johannes Kepler University Linz, the E-Commerce Competence Center and the ec3Networks GmbH. This section outlines the principal ideas of the overall SemCrypt project as described in [29].

SemCrypt aims at realizing a queryable encrypted data store for XML documents. SemCrypt enables to query and update encrypted XML documents. The principal approach is similar to the current approaches but the disadvantages of the current approaches mentioned in section 1.4 are avoided.

SemCrypt operates in a trusted- and an untrusted environment. Thus the basic architecture of SemCrypt is split into the following two components shown in figure 1.3:

- SemCrypt Database Manager
- Storage Provider

[1]Note that this approach only covers some of the features provided by SemCrypt thus it can only be compared to these features and not to the full SemCrypt solution

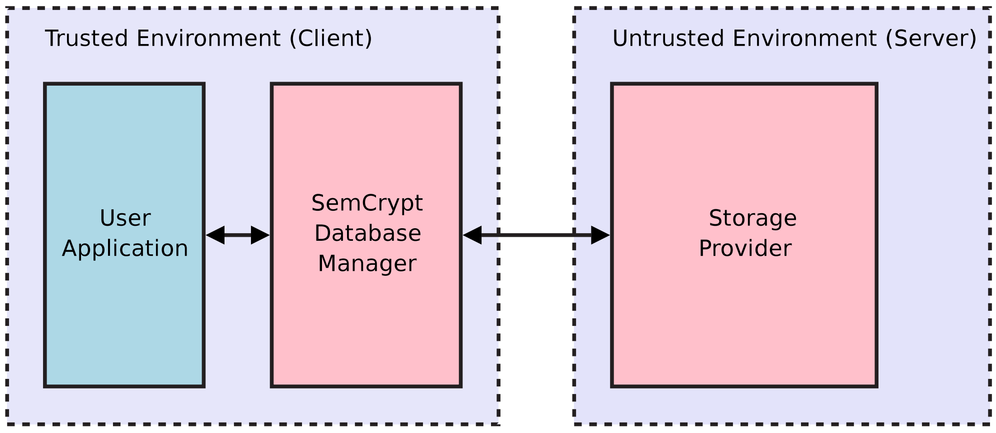
Trusted Environment (Client)
Untrusted Environment (Server)
User Application
SemCrypt Database Manager
Storage Provider

Figure 1.3: SemCrypt architecture overview

The *SemCrypt Database Manager* operates in the trusted environment and serves requests from the end user application. The SemCrypt Database Manager provides high level services like indexing, document upload/download and query processing to user applications. All operations that need to be executed in a trusted environment, in order to avoid security breaches, are performed by the SemCrypt Database Manager. To access a data store, the SemCrypt Database Manager communicates with the Storage Provider.

The *Storage Provider* operates in the untrusted environment and provides access to the data store. The Storage Provider acts as an interface between a third party data store (e.g. database) and the SemCrypt Database Manager thus enabling access to arbitrary data stores. Because the Storage Provider operates in the untrusted environment, the SemCrypt Database Manager has to encrypt data before it is sent to the Storage Provider. Therefore the Storage Provider only handles encrypted data. Furthermore everytime the Storage Provider returns data to the SemCrypt Database Manager the result has to be decrypted before it can be processed by the SemCrypt Database Manager.

To unitize the communication between the SemCrypt Database Manager and the Storage Provider a further component named Storage Engine is required. The Storage Engine encapsulates the whole communication between the high level services like query processing and indexing of the SemCrypt Database Manager and the Storage Provider. Figure 1.4 shows the Storage Engine that operates between these two components. Both components, the Storage Engine and the Storage Provider, together are subsequently referred to as the *Semcrypt Store*.

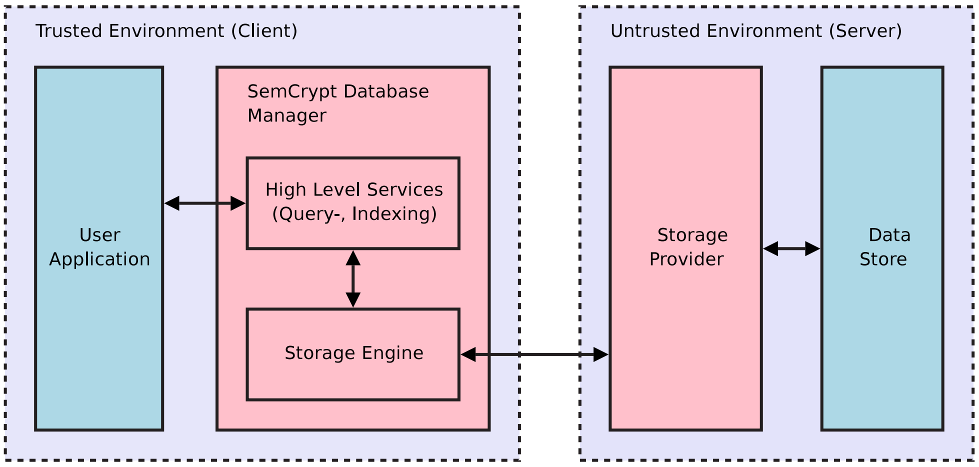
Trusted Environment (Client)
User Application
SemCrypt Database Manager
High Level Services (Query-, Indexing)
Storage Engine
Untrusted Environment (Server)
Storage Provider
Data Store

Figure 1.4: Storage Engine overview

1.6 Objective of the thesis

The focus of this thesis is the design and implementation of the Storage Engine and the Storage Provider enabling SemCrypt to store and query encrypted data.

The requirements for SemCrypt Store have to be described and the strategies to meet these requirements have to be implemented.

It is possible to employ the SemCrypt Store in other projects as well. Within the scope of this master thesis the development of the SemCrypt Store focuses primary on the requirements of SemCrypt.

SemCrypt needs a storage layer for the query-, index- and metadata processing components compromised within the SemCrypt Database Manager. The objective is to derive the architecture from the design document [21] and perform an in depth design for the SemCrypt Store.

1.7 Outline

Chapter 2 This chapter lists requirements that have to be met by the SemCrypt Store. Additionally strategies to identify the requirements are outlined.

Chapter 3 Based on the requirements and the proposed strategies in the previous chapter, the architecture and modularization of the SemCrypt Store is explained.

Chapter 4 This chapter describes the technologies that are chosen to implement the SemCrypt Store.

Chapter 5 This chapter describes the details of the implementation of the components defined by the architectural components. Further this chapter describes how the chosen technologies interfered the implementation.

Chapter 6 This chapter presents related work. It briefly describes other approaches of secure data stores.

Chapter 7 The work that is out of scope of this master thesis is described in this chapter. These are features that were not implemented and therefore are not mentioned in Chapter 5, but can be integrated in future releases.

Chapter 8 This chapter concludes the thesis.

Chapter 2

Requirements and Strategies

Contents

This chapter describes the requirements that need to be met by the SemCrypt Store. The requirements are derived from Schrefl et al. [29] and related requirements are grouped into categories to provide a high level overview. Each section in this chapter covers one category. For each requirement the strategies that are used to meet the requirement are described. The possible problems related to the strategies are mentioned and solutions are presented.

2.1 Protect document structure and content

Requirement 1 (Hide content) *The content of XML documents holding sensitive information has to be hidden. Only authorized persons should be able to access the content of such documents.*

Strategy 1 (Cipher algorithms) To avoid security threats like those described in section 1.3, e.g. data theft, and to meet the requirement to hide the content of the document, cipher algorithms are used to encrypt all data stored by the Storage Provider. The encrypted data contains document data, index data and metadata that describes the document.

The general principle of encryption is to use an encryption function (E) and a key (k) to encrypt a plain text (p) into a cipher text (c). For decrypting c a decryption function (D) is used together with k. Figure 2.1 depicts this basic encryption and decryption concept[1].

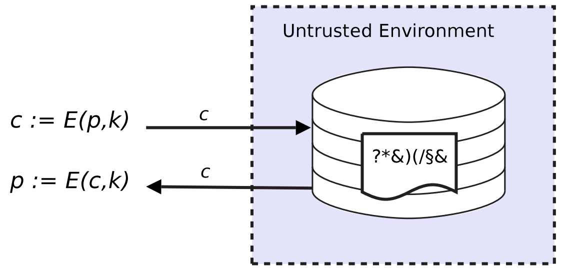

Figure 2.1: Encryption and decryption of data

Requirement 2 (Hide associations) *It is not sufficient to only encrypt the data of a document. The internal associations of documents have to be hidden, making it harder to figure out valuable information stored in the document.*

Strategy 2 (Identifier-value pairs grouped into containers) All data like document content, indexes and metadata is stored as identifier-value pairs. When applying this approach to complex data structures like XML documents these documents have to be fragmented into small data chunks. Using labeling schemes like the one described by Grün et al. [18] means that identifiers are applied to the data chunks resulting in multiple identifier-value pairs for one complex data structure. According to the labeling scheme described by Grün et al. [18], the nodes in a document are assigned with identifiers. This task is performed outside of the Storage Engine and is not covered by this thesis.

Example 4 *When the concept of identifier-value pairs is applied to the nodes in figure 1.2 each node is assigned with a different label. The contracts node will be assigned with label 1, the contract node with label 2-1, the id node with label 3-1 and so on. Figure 2.2 depicts all the assigned labels in the document tree.*

The identifier-value pairs are passed to the Storage Engine for storing values in the Storage Provider. Inside the Storage Engine the identifiers are used as unique keys for the values. Because every identifier-value pair is stored separate and has no direct association to its parent or children the structure of the document

[1]Note that this is not the final encryption approach used by SemCrypt - this approach will be enhanced later in this chapter.

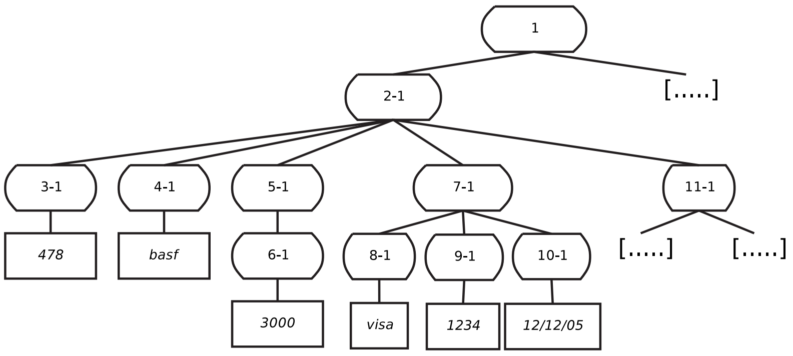

Figure 2.2: Identifiers for the contract nodes

is hidden when storing the data of the document in an identifier-value based manner.

Additionally to referencing one value, an identifier can also reference a *value list*. This avoids having to store identifier-value pairs with the same identifier and thus avoids equal storage patterns that are a potential security leaks.

An identifier is used to uniquely identify one value or value list enabling the lookup of a specific value by its identifier. The encrypted identifiers are stored together with the encrypted values in a table like structure.The encrypted value is called cipher text. One row of this table consisting of an encrypted identifier and an encrypted value is called cipher row. The table holding the cipher rows is subsequently called container.

Example 5 (Cipher rows) *Figure 2.3 depicts this relationship.*

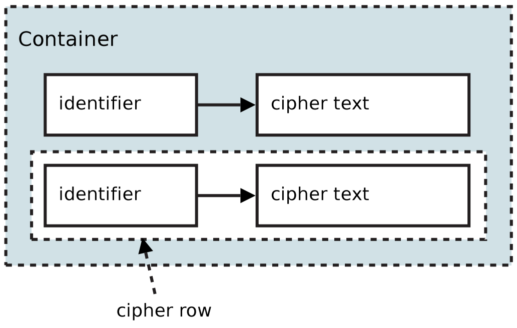

Figure 2.3: A container holding cipher rows

Requirement 3 (Hide document structure) *It is not sufficient to only encrypt the data of a document. The structure of the document also has to be hidden.*

Strategy 3 (Hash functions) Because an encrypted text has approximately the same size as the plain text[2], encrypting the identifiers separately as described in the previous strategy can lead to a security problem. Depending on the structure of the identifier ([18]), an attacker can get information about the structure of the document by looking at the length of the identifiers.

Because of this problem it is suggested to store hash values instead of the encrypted identifiers for querying values. The hash values are generated using a hash function on the identifier and are then stored and used to identify the value instead of the identifier itself. The advantage of a hash function is that it takes a message of any length as input and produces a fixed length hash value as output, sometimes this output is termed as message digest or digital fingerprint. A hash value identifying a cipher text is subsequently called cipher id. Figure 2.4 displays a container with two cipher rows identified by a cipher id.

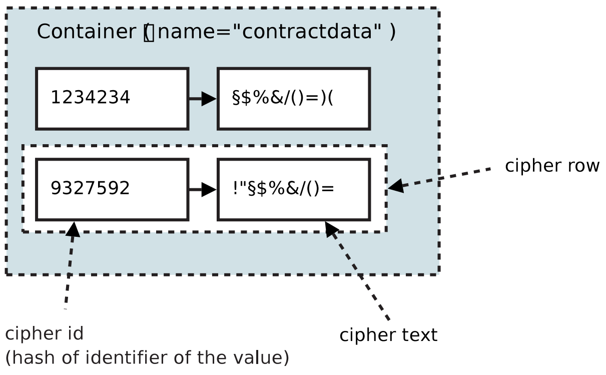
Container (name="contractdata")
1234234
§$%&/()=)(
9327592
!"§$%&/()=
cipher row
cipher id
(hash of identifier of the value)
cipher text

Figure 2.4: Cipher id's that identify the cipher text

The most common hash functions are:

- *Message-Digest algorithm 5 (MD5)* - This hash function has been used in a wide variety of security applications but after flaws have been found in 1996 and 2004 cryptographers recommend to use SHA-1 instead of MD5 [35].
- *Secure Hash Algorithm (SHA-1)* - This hash function is considered as being the successor to MD5 and was implemented by the National Institute of Standards and Technology. This function is not used as widely as MD5, but popular copyright protection systems use this hash function.

[2]When using a block ciper algorithm the size of the cipher text depends on the block length and the length of the cipher text.

Example 6 (Hash values) *This example demonstrates a character input and the resulting cipher id (MD5 hash 128 bit). Figure 2.5 shows the indentifiers and the respective cipher ids. Note that both cipher ids have the same size regardless of the length of the original identifier*[3] *length:*

$$\mathit{MD5(identifier)} = \mathit{cipher\ id}$$

MD5(/contracts/contract[1]/payment/@cardid) = 9e107d9d372bb6826bd81d3542a419d6

MD5(/contracts) = d41d8cd98f00b204e9800998ecf8427e

Figure 2.5: MD5 hash value example

The fixed output length of the hash functions has the advantage that regardless of the length of the identifiers used, the cipher ids stored in the Storage Provider will always be of the same size which improves security.

But the use of hash functions for identifying values has a drawback. A fundamental property of all hash functions is that if two hashes (according to the same function) are different, then the two inputs were different. This property is a consequence of hash functions being deterministic, mathematical functions, but they are generally only surjective functions. Consequently, the equality of two hash values does not guarantee that the two inputs were the same. The situations of two different inputs resulting in the same hash value is called hash collision. This is a problem for using a hash value as a cipher id to look up a value, because a single value cannot be uniquely identified anymore.

To solve this problem the identifier is stored together with the value in the cipher text and the query to lookup an identifier is split into two steps. In the first step the cipher id is used to find the correct cipher row at the Storage Provider that holds the identifier. The second step is to find the correct identifier-value pair inside the cipher row, which is done after the cipher row was decrypted by the Storage Engine. Figure 2.6 shows a container holding the cipher ids and the identifiers that are stored together with the respective values.

[3]In this example different identifiers are used to illustrate the behavior of hashfunctions

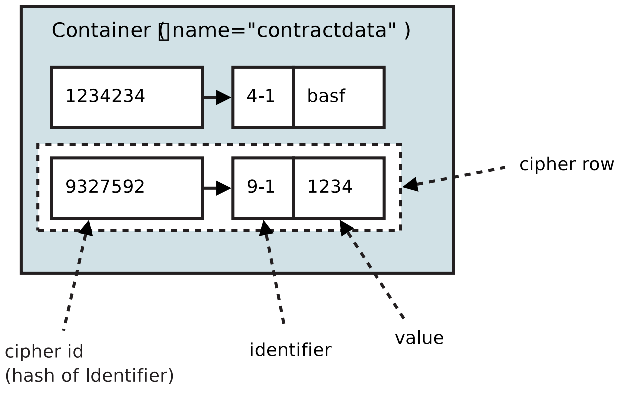

Figure 2.6: A container holding cipher ids and values (decrypted)

Figure 2.7 shows the same container as decribed in 2.6 as it is seen by the Storage Provider. The identifiers and values are encrypted together in one cipher text.

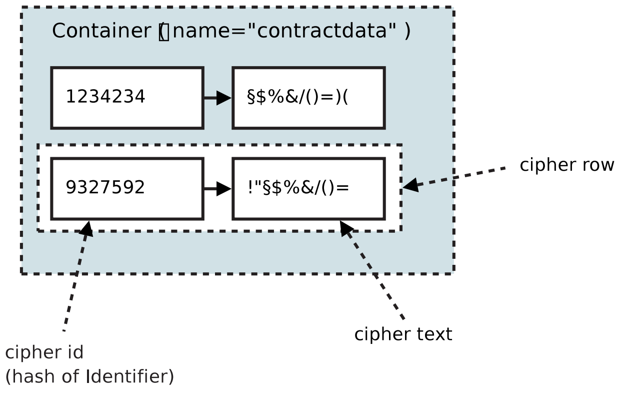

Figure 2.7: A container with cipher text

Hash functions also have an impact on the physical storage layout of the data. Hash functions which are surjective functions can be qualified as *good* or *bad hash functions*. A good hash function is one that yields only few hash collisions whereas a bad hash function results in many hash collisions. If *bad hash functions* are chosen for hashing the identifiers, as a consequence, the Storage Provider will store many identifier-value pairs together in few cipher rows. These cipher rows have to be loaded every time when any of the identifier-value pairs are accessed. This means that it is a general rule that cipher rows holding less values are accessed less often than cipher rows containing many identifier-value pairs.[4]

[4]When using MD5 or SHA-1 it is very rare that one gets into contention troubles because both are *good* hash functions

2.2 Avoid statistical analysis

Requirement 4 (Avoid statistical analysis because of cipher characteristics) *If the same plain text is encrypted with the same cipher algorithm and the same cryptographic key, the resulting cipher text will be equal. In real life, documents often start with similar or identical data, and an attacker should not be able to detect this using statistical analysis. Therefore each time the same plain text occurs it has to be encrypted differently.*

Strategy 4 (Nonce based encryption) To be able to generate a different cipher text result each time the same plain text is encrypted, the cipher algorithm has to be parameterized with a different value each time the encryption is performed.

There are different approaches of parameterizing the cipher algorithm with a variable input.

- *Message numbers* - The first approach is to just use a counter that starts with zero and is incremented each time a text has to be encrypted. The counter is used to parameterize the cipher algorithm in order to produce a different encryption result for same plain texts. This is a problem for many plain texts because it is very likely that documents start with similar characters. If the beginning blocks of the plain text have small differences, then the simple counter potentially cancels the differences in xor operations, and identical cipher text blocks are again generated.

  Example 7 *As an example consider the values* zero *and* one *as the first and the second initialization values for the cipher algorithm. If the first and the second message also only differ in the first bit of the leading plain text block two equally encrypted cipher texts are generated [28].*

- *Random numbers* - Another approach is to use random numbers to parameterize the cipher algorithm each time a value is encrypted. According to [28] there are two disadvantages when using random numbers. First it is hard to find or implement a reliable random number generator. The second disadvantage is that if each random number is stored in the first cipher block of a message it consumes all the space in the first cipher block (usually 128 bit) which is a huge overhead when the cipher text is very short.

- *Nonce* - This approach is the best solution to initialize cipher algorithms. The idea is to use a number that is unique for initializing the cipher every time a message is decrypted. For every encryption a new nonce is generated.

This approach requires an algorithm that produces a unique number. The size of this number is usually much smaller than one cipher block and therefore produces less overhead compared to the random number solution.

Although the random number approach and the nonce based approach are very similar the nonce based solution is chosen to be used by the SemCrypt Store since the nonce based approach produces less storage overhead.

Consequently the cipher aproach described in strategy "Cipher algorithms" (strategy 1) has to be enhanced by a nonce (n) and this nonce has to be stored in conjunction with the cipher row. Figure 2.8 shows the enhanced nonce based cipher formulas that will be used in SemCrypt Store.

$$c := E(p,k,n)$$
$$p := D(c,k,n)$$

Figure 2.8: Nonce based Encryption and decryption

Figure 2.9 illustrates how the container looks like from the view of the Storage Provider. The figure shows the nonce and the cipher text stored together.

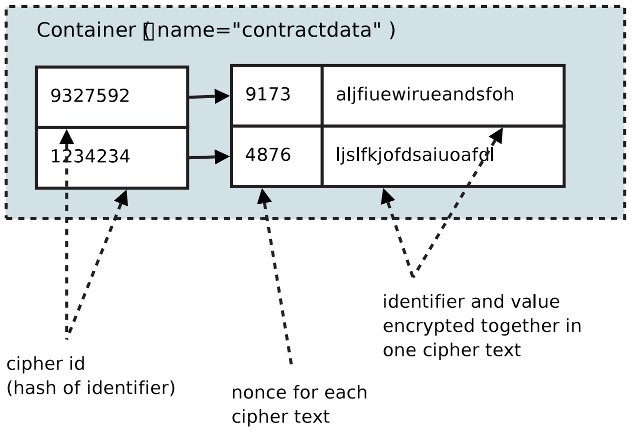

Figure 2.9: An encrypted container

Requirement 5 (Avoid statistical analysis because of hotspots) *Hotspots are pieces of data that are accessed very frequently. The requirement of avoiding repetitive access to the same data has the following reasons:*

- Performance - *Accessing the same data on an external data store like a hard disk causes performance losses. Especially when operating in a multiuser environment contention is likely to occur.*

- Security - *Hotspots constitute a potential security leak because an attacker can figure out that certain data is accessed very frequently by monitoring the disk access and can therefore derive information.*

Strategy 5 (Caching) The solution to avoid hotspots is to implement a cache inside the Storage Engine. The cache reduces the number of requests sent from the Storage Engine to the Storage Provider and thus decreases the frequency of accesses to the same data. Fewer requests to the Storage Provider imply that decrypting data with statistical analysis gets harder. Also the performance increases when using caching:

- If a request can be handled by the cache, the roundtrip to the Storage Provider is avoided.
- The overall performance of the Storage Provider increases because the overall amount of requests that need to be handled by the Storage Provider decreases.

Additionally preserving the cache during the startup and shutdown fulfills the requirement of avoiding hotspots even better because the cache needs not to be built after the startup of the client but is initially populated with the values that were saved before shutdown of the client.

Note that since containers separate the storage of identifiers-value pairs, they can also be used to reduce the size of identifier-value pairs that are associated with one hash value and thus decrease the number of access operations on one cipher row. This is only relevant when the hash function used to generate cipher ids is *bad*.

2.3 Extensibility

Requirement 6 (Use arbitrary cipher algorithms) *It is required that SemCrypt Store is independent of any existing cipher algorithm making it possible to switch to arbitrary cipher algorithms when it turns out that the cipher is insecure. The cipher can get insecure because of too small keys or security leaks that are discovered in cipher algorithms. This is already true for the DES cipher algorithm [27].*

It can also be necessary to use a certain cipher algorithm because of performance considerations. Users of SemCrypt with a primary goal of processing high volumes of data may decide to use faster but more insecure cipher algorithms.

Example 8 *Table 2.1 holds some performance examples of block cipher algorithms. It illustrates that there are large performance differences between the various algorithms. Note that the hardware that was used to execute this benchmarks was a PC with a 486 CPU with 33MHz but in this case only the relations are demonstrated.*

Algorithm	Encryption speed in kb/s
DES	35
TRIPPLE DES	12
IDEA	70
Blowfish (12 rounds)	182
Blowfish (16 rounds)	135
Blowfish (20 rounds)	110

Table 2.1: Encryption performance of block cipher algorithms

Strategy 6 (Flexible design) As per the requirement to be able to exchange the cipher algorithms the cipher component of SemCrypt is separated making it possible to plug in any new developed or already available cipher algorithm.

Requirement 7 (Use arbitrary data stores) *ASPs offer many different services for storing data including file systems and databases. It is required for SemCrypt to be able to operate on arbitrary data stores. This makes SemCrypt very flexible allowing to adopt existing data store environments for the use with SemCrypt Store.*

Strategy 7 (Storage Provider) Fulfilling the requirement to use any data store requires the Storage Provider to be split into two parts. One part that communicates with the Storage Engine and another part that operates like an adapter between the first part of the Storage Engine and the employed low level data store like a database or a file system. New adapters can be implemented for new databases as needed without having to change the core processing of SemCrypt Store.

2.4 Consistency

Requirement 8 (Ensure ACID principles) *Data consistency is of most importance for SemCrypt. The SemCrypt Store has to guarantee the following characteristics for handling data, abbreviated by the acronym ACID [22] [26]:*

- Atomicity - *relates to the operations of a transaction. Because a transaction often consists of more than a single operation, atomicity requires that all the operations of a transaction perform successfully for the transaction to be considered complete. If even a single operation cannot be performed, none of the transaction's operations are performed.*
- Data Consistency - *A transaction must transition data from one consistent state to another. In addition, the transaction must preserve the data's semantic and physical integrity*
- Isolation - *It has to be possible for many Storage Engine operations to run concurrently but the single operation should not see immediately the changes of the other operations. Isolation prevents an operation from obtaining an inconsistent view of the data. Data inconsistency can occur if one operation sees just a subset of another operations updates due to the inter-dependencies among these updates. Isolation is related to transaction concurrency.*
- Durability of Data - *means that changes made by successful data manipulation operations persist in the data store regardless of failure conditions. It guarantees that completed changes remain in the data store even if failures occurred after the completed operation.*

Strategy 8 (Transactional support) The Storage Engine provides transaction handling for high level services of the SemCrypt Database Manager. The transaction handling is required to be able to modify the stored data and commit all changes at once and to guarantee that all or no changes are applied - avoiding inconsistent data.

Requirement 9 (Ensure common data access) *The stored data must be maintained in a platform and programming language independent way making it possible for SemCrypt Store to operate on a large variety of operating systems and platforms.*

Strategy 9 (No language specific features) No language and platform specific features are used during the implementation of the storage structures at the Storage Provider.

2.5 Performance

Requirement 10 (Performance) *Because SemCrypt potentially has to handle a large amount of data, it needs to be scalable. The design of SemCrypt Store has* *to ensure that the communication between Storage Engine and Storage Provider is fast enough for an acceptable response time for queries of the higher level services.*

Strategy 10 (Communication) Basically the strategies used to gain performance are the same that are implemented to avoid hotspots. Additionally the following strategy is used to increase the performance for the operation of the SemCrypt Store.

The Storage Engine and the Storage Provider have to implement an efficient communication mechanism to reduce response time. The amount of transmitted data between these two components is reduced to a minimum to ensure a fast communication mechanism.

Chapter 3

Architecture

Contents

Based on the requirements and strategies defined in chapter 2 and the basic architecture outlined in the SemCrypt design specification [21], this chapter describes the detailed architecture of the SemCrypt Store. After introducing the overall architecture in the overview the two main components Storage Engine and Storage Provider are described.

For developing the architecture of the SemCrypt Store, the software design principles adaptivity, extensibility and reusability are applied. Adaptivity is important because according to the requirements "Use arbitrary cipher algorithms" (requirement 6) and "Use arbitrary data stores" (requirement 7) the SemCrypt Store has to be designed in a flexible way. It has to be possible to integrate the SemCrypt Store using various cipher algorithms in existing ASP environments. Focusing on extensibility is necessary because this enables developers to add additional features in the future. Applying the concept of reusability enables parts of SemCrypt Store to be used in other projects.

The architecture is designed in a component based approach. This means that the components of the SemCrypt Store are identified and equipped with operations.

3.1 Overview

SemCrypt Store operates in a trusted- and an untrusted environment. Thus the basic SemCrypt Store architecture is split into the following two components:

- Storage Engine
- Storage Provider

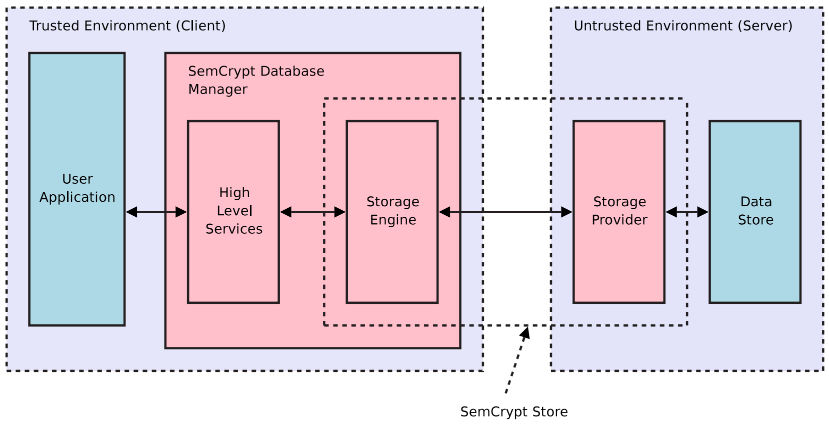

Figure 3.1: SemCrypt Store overview

Figure 3.1 depicts the components Storage Engine and Storage Provider. The Storage Engine provides an interface to store and retrieve values for high level services. Furthermore it performs encryption and decryption of values that are provided by these services. To store and retrieve encrypted values the Storage Engine communicates with the Storage Provider that provides unified access to arbitrary data stores. The Storage Provider operates with encrypted data only and enables the Storage Engine to access the transactional support of data stores.

3.2 Storage Engine

The Storage Engine operates inside the SemCrypt Database Manager in a trusted environment. This is necessary since the Storage Engine handles plain text data and thus does not qualify to operate in an untrusted environment. The Storage Engine itself is separated into the building blocks shown in figure 3.2:

- *Storage Gateway* - This building block exposes the value storage and retrieval capabilities of the Storage Engine to the SemCrypt Database Manager.
- *Transaction Controller* - This building block exposes the transaction handling capabilities of the SemCrypt Store to the SemCrypt Database Manager.

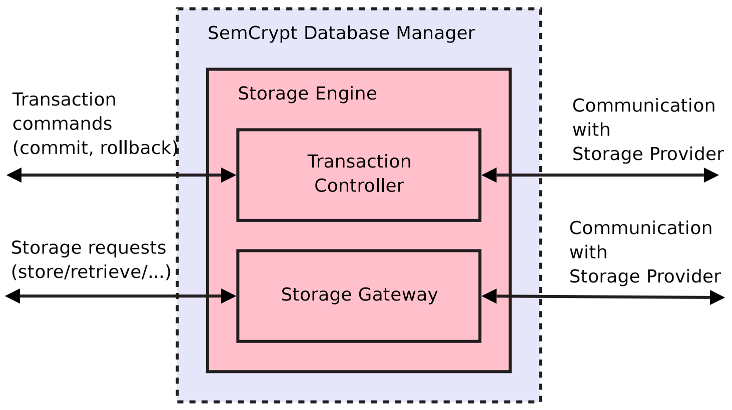
SemCrypt Database Manager
Storage Engine
Transaction commands (commit, rollback)
Transaction Controller
Communication with Storage Provider
Storage requests (store/retrieve/...)
Storage Gateway
Communication with Storage Provider

Figure 3.2: Storage Engine building blocks

3.2.1 Storage Gateway

The Storage Gateway receives requests to store or retrieve values for high level services of the SemCrypt Database Manager. When storing a value the Storage

Gateway encrypts the value and builds a cipher row which is then submitted to the Storage Provider. When a value is requested by a high level service, the Storage Engine requests the value from the Storage Provider using the respective identifier, decrypts the cipher text and identifies the associated value. The value is then passed back to the high level service.

The Storage Gateway offers the following operations to store and retrieve values for high level services:

- Store value
- Retrieve value
- Remove value

Since these operations share common functionality the Storage Gateway is internally organized into several components. The components are described in the subsequent sections. Their operations are only used inside the Storage Gateway and are not visible to external components. Figure 3.3 shows the internal components and the operations of the Storage Gateway.

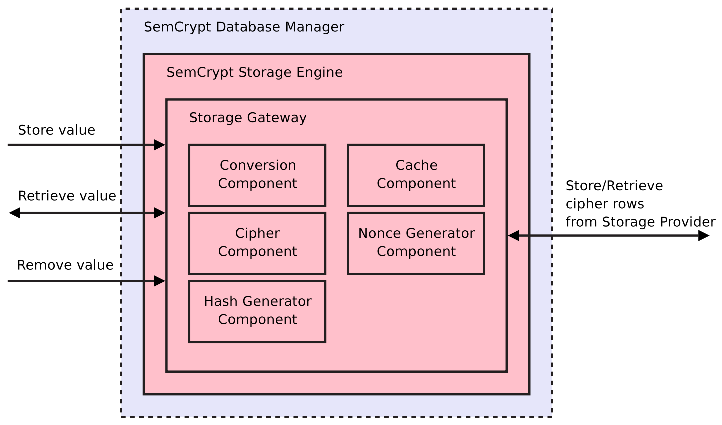

Figure 3.3: Storage Gateway components and operations

3.2.1.1 Operations

The following sections explain the operations provided by the Storage Gateway component. Each section starts with an overview table that shows the input and output parameters of the respective operation. After each overview table the operation is described.

3.2.1.1.1 Store value

Parameter	Type	Description
identifier	INPUT	Identifier for the value.
value	INPUT	The value to be stored.
container	INPUT	A reference to the container used to store the value.
status	OUTPUT	A status information whether the operation was successful or not.

Table 3.1: Parameters of operation "Store value"

This operation creates a new cipher row according to the given identifier and value. If a cipher row with the same cipher id already exists the identifier and value are stored in the existing cipher row. The cipher row is then encrypted and passed to the Storage Provider. To encrypt the cipher text a cipher key that is specified in the configuration of the Storage Engine and the nonce of the cipher row are used. The steps that have to be performed by the store operation are subsequently described and figure 3.4 depicts the steps in a flow diagram.

1. Create a cipher id from the given identifier. This cipher id is used to identify the cipher row in the container of the Storage Provider.

2. Check if a cipher row was found.

   If a cipher row was found, the cipher key and the nonce are used to decrypt the cipher text. The identifier is then used to find the correct value in the decrypted cipher text. The old identifier-value pair in the cipher text is then replaced by the new identifier-value pair.

   If no cipher row is found, a new cipher row is created. The cipher id created in step 1 is stored in the cipher id of the cipher row. The identifier-value pair is set in the cipher row.

3. A nonce is generated and applied to the cipher row.

4. The cipher row is encrypted using the specified cipher key and the generated nonce.

5. The cipher row is submitted to the Storage Provider which stores the cipher row in the specified container.

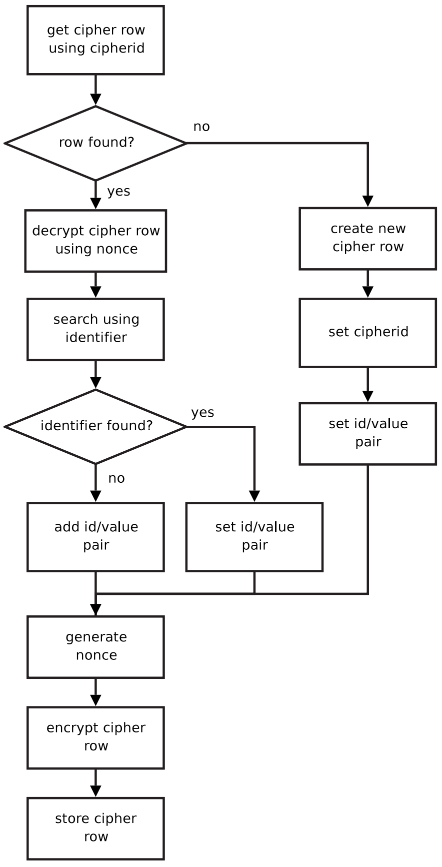

Figure 3.4: Flow of storing a value

Note that a new nonce value is generated every time the cipher row is changed by the store operation. It is also possible to only generate a nonce when the cipher row is created and use the same nonce for subsequent store operations involving this cipher row. This approach is slightly faster because the nonce does not have to be generated when cipher rows are updated, but because of the improved security a new nonce is generated every time the cipher row changes.

3.2.1.1.2 Retrieve value

Parameter	Type	Description
identifier	INPUT	Identifier for the value.
container	INPUT	A reference to the container that holds the value.
value	OUTPUT	The value that is stored together with the specified identifier. In case no value is found NULL is returned.

Table 3.2: Parameters of operation ”Retrieve value”

This operation receives a reference to a container plus an identifier as input and returns the value stored in conjunction with the identifier.

Figure 3.5 shows how a value is retrieved by the Storage Engine. Subsequently the steps of retrieving a value are explained:

1. Create a cipher id from the specified identifier. The cipher id is used to find the cipher row in the container of the Storage Provider.

2. Check if a cipher row was found.

   If no cipher row is found, return NULL.

   Otherwise continue.

3. Use the cipher key that is specified in the Storage Engine configuration and the nonce stored in the cipher row to decrypt the cipher text.

4. Find the value in the decrypted cipher text using the identifier.

5. Check if the value was found.

   If no value was found in the cipher text matching the specified identifier, NULL is returned.

   Otherwise return the value associated with the specified identifier.

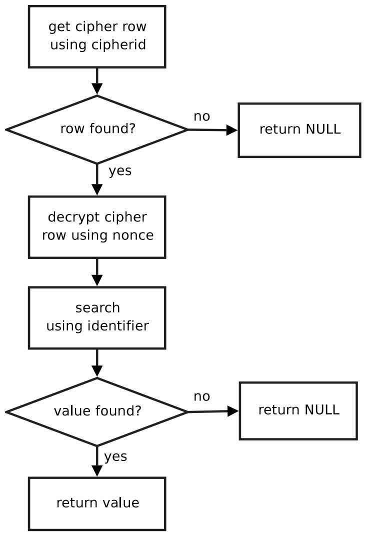

Figure 3.5: Flow of retrieving a value

3.2.1.1.3 Remove value

Parameter	Type	Description
identifier	INPUT	Identifier for the value that has to be removed.
container	INPUT	A reference to the container that holds the identifier-value pair.
status	OUTPUT	A status information whether the operation was successful or not.

Table 3.3: Parameters of operation "Remove value"

This operation removes a value with the given identifier from the specified container. Figure 3.6 shows the steps that are required to remove a value:

1. Create a cipher id from the specified identifier. This cipher id is used to find the cipher row in the Storage Provider.

2. Check if a cipher row was found.

   If no cipher row was found an error status is returned.

   Otherwise continue.

3. Decrypt the cipher text using the cipher key that is stored in the configuration of the Storage Engine and the nonce stored in the cipher row.

4. Search in the decrypted cipher text for the correct identifier

   If the identifier is not found in the cipher row return an error.

   Otherwise continue.

5. Remove the value from the cipher row.

6. Check if cipher row is empty

   If cipher row is empty remove the cipher from from the container

   Otherwise continue.

7. Generate a new nonce for the cipher row.

8. Encrypt the cipher row using the new nonce and the cipher key that is stored in the configuration of the Storage Engine.

9. Store the changed cipher row using the Storage Provider.

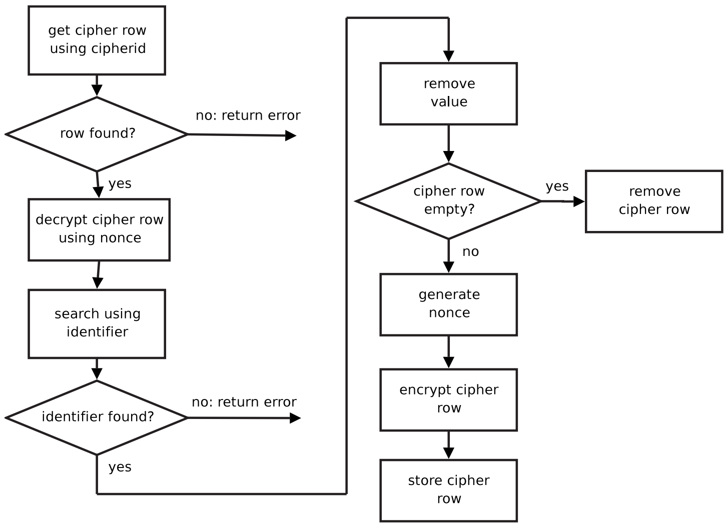

Figure 3.6: Flow of removing a value

3.2.1.2 Components

The subsequent paragraphs describe the components and their operations that provide common functionality for the Storage Gateway.

3.2.1.2.1 Cipher Component

All cipher algorithms are encapsulated in this component. The component receives plain text data and returns the associated cipher data and vice versa. The cipher component does not implement cipher algorithms by itself. It is an adapter that acts as a bridge between the Storage Gateway and an existing cipher algorithm. The reason for this adapter is that different providers for cipher algorithms have different interfaces for their algorithms. According to the requirement "Use arbitrary cipher algorithms" (requirement 6) the design of an adapter enables to directly plug in any specific cipher algorithm in the Storage Gateway.

Every time the Storage Gateway needs to encrypt or decrypt values it uses this component. This single point of access to the cipher algorithm has the advantage that the whole component can be replaced easily without having to modify any other components.

Encrypt

Parameter	Type	Description
plain text	INPUT	The plain text that has to be encrypted.
encrypted text	OUTPUT	The encrypted text.

Table 3.4: Parameters of operation "Encrypt"

The encrypt operation receives a plain text, encrypts the plain text using the cipher algorithm that is configured for the Storage Engine and returns the result.

Decrypt

Parameter	Type	Description
encrypted text	INPUT	The encrypted text.
plain text	OUTPUT	The plain text.

Table 3.5: Parameters of operation "Decrypt"

This is the inverse functionality of the encrypt operation. This operation receives an encrypted text, decrypts the encrypted text using the decryption algorithm that is configured for the Storage Engine and returns the result.

3.2.1.2.2 Hash Generation Component

The hash generation component is a pluggable implementation of a hash algorithm (H) that is used by the Storage Gateway to generate cipher ids.

Generate hash

Parameter	Type	Description
text	INPUT	A text that has to be hashed.
hash value	OUTPUT	The hash value of the given text.

Table 3.6: Parameters of operation "Generate hash"

This operation gets arbitrary text as input and returns a hash value as a result. This functionality is shown in figure 3.7.

ciperhid := H(identifier)

Figure 3.7: Hash generation

This operation is used by the Storage Gateway to transform identifiers into cipher ids.

3.2.1.2.3 Nonce Generation Component

The nonce generation component is a pluggable implementation of a nonce generation algorithm (*N*). This component is used by the Storage Gateway to generate nonce values. The nonce values are then used for encryption or decryption in conjunction with the cipher key that is stored in the configuration of the Storage Engine.

Generate nonce

Parameter	Type	Description
nonce	OUTPUT	The generated nonce.

Table 3.7: Parameters of operation "Generate nonce"

This operation does not receive any input but returns a unique number each time it is invoked. This behavior is displayed in figure 3.8

$$nonce := N()$$

Figure 3.8: Nonce generation

3.2.1.2.4 Cache Component

To conform to strategy 5 which suggests caching to avoid hotspots in the data store and to improve performance, a caching mechanism is designed as part of the Storage Gateway architecture. The cache handling is separated in one single component to make it pluggable.

There are two approaches of how caching can be designed for the Storage Engine.

- Cache Solution 1 - Caching decrypted identifier-value pairs. Every single value that is passed to the Storage Engine is placed into a cache along with its identifier. In this case the cache holds plain text. The advantage of this solution is that there is minimal processing between the request of the value (requested by a service in the SemCrypt Database Manager) and the return of the value because the identifier-value pair does not have to be decrypted each time it is requested.

- Cache Solution 2 - Cache cipher rows which means that the identifier-value pairs are cached encrypted. This implies that even if a value is retrieved from the cache, the encryption of the cipher row and the resolving of hash collisions has to be done. The advantage of this solution is that, if the cache swaps cipher rows to disk, the values and identifiers are still encrypted.

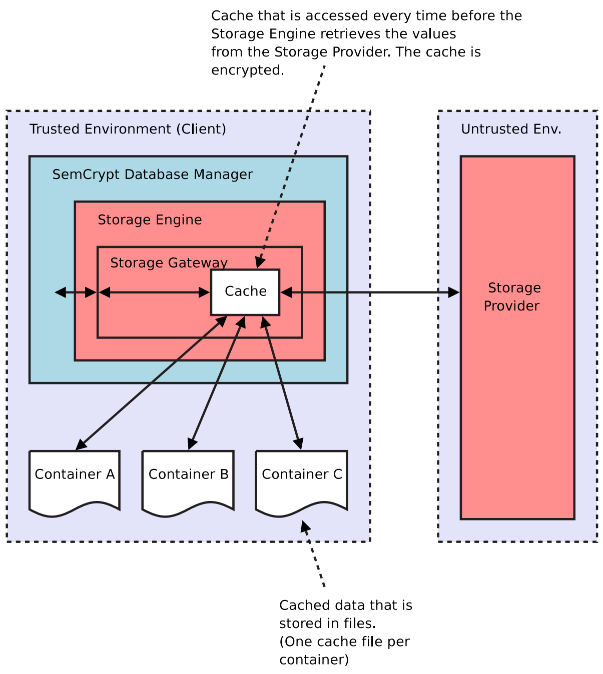
Cache that is accessed every time before the Storage Engine retrieves the values from the Storage Provider. The cache is encrypted.
Trusted Environment (Client)
SemCrypt Database Manager
Storage Engine
Storage Gateway
Cache
Untrusted Env.
Storage Provider
Container A
Container B
Container C
Cached data that is stored in files. (One cache file per container)

Figure 3.9: Cache component

Since the improved security of Cache Solution 2 in comparison to Cache Solution 1, Cache Solution 2 is used within the Storage Engine architecture. Figure 3.9 shows the cache in the Storage Gateway that holds the encrypted cipher rows.

The cipher id that identifies a cipher row in a container also identifies the corresponding cipher row in the cache. When using just one cache this leads to a problem because the cipher id is *unique per container*. If there are the same cipher ids in multiple containers, the cached cipher rows are mixed up in the cache thus one cache per container will be used.

Reading and writing cipher rows:

- When reading a cipher row, the Storage Engine first queries the cache using the same cipher id that is used to store the cipher rows in the Storage Provider. In case a cipher row is found, the cached cipher row is used. Otherwise the Storage Engine retrieves the cipher row from the Storage Provider.

- During modification operations, cipher rows are created or updated. In these situations the cipher rows are written to the cache before the cipher rows are stored in the SemCrypt Store.

Put element

Parameter	Type	Description
id	INPUT	The id for the element to be cached.
element	INPUT	The element that has to be stored in the cache.

Table 3.8: Parameters of operation "Put element"

This operation is used by the Storage Gateway to put cipher rows into the cache. A cipher id and the cipher row are passed as parameters to this operation and are then stored in the cache.

Get element

Parameter	Type	Description
id	INPUT	The id of the cached element.
element	OUTPUT	The element that was found in the cache or NULL if no matching cipher row as found

Table 3.9: Parameters of operation "Get element"

This operation is used by the Storage Gateway to get cipher rows from the cache. A cipher id is passed as a parameter to this operation and the corresponding cipher row is returned if it is found. If no cipher row is found NULL is returned.

3.2.1.2.5 Conversion Component

This component performs the conversion of a value associated with an identifier to a cipher row and vice versa. It is used by the Storage Gateway every time a cipher row is retrieved from the Storage Engine or submitted to the Storage Engine.

To ensure a convenient use of the Storage Engine operations in the SemCrypt Database Manager, datatypes are supported. Using datatypes has the following advantages:

- *High level services can operate on typed data* - This avoids having high level services to convert data to the SemCrypt Store internal data representation (cipher row) and back to values with datatypes.

- *Typechecks* can be performed when loading data from the SemCrypt Store. Typechecks ensure that the value which is stored in a byte array format can be converted back to the appropriate data type.

Also *value lists* are supported making it possible to not only store a single value together with an identifier but also a whole list of values of the same type. Value lists not only make the handling of multiple values possible but also increase the performance when retrieving the values because the elements of a list are guaranteed to be stored together in one cipher row.

Build cipher row

Parameter	Type	Description
identifier	INPUT	The identifier of the value
value	INPUT	The value to be converted.
cipher row	OUTPUT	The cipher row that was created.

Table 3.10: Parameters of operation ”Build cipher row”

This operation creates a cipher row using an identifier and a value. Information about the value’s datatype as well as the number of values in case of a value list is stored in the cipher row. If the datatype can contain data of variable length the length information is stored in the cipher row as well.

Get identifier-value from cipher row

Parameter	Type	Description
cipher row	INPUT	The cipher row containing encrypted data.
identifier	OUTPUT	The identifier of the value
value	OUTPUT	The value that was extracted from the cipher row.

Table 3.11: Parameters of operation ”Get identifier-value from cipher row”

This operation is the inverse operation of the ”Build cipher row” operation and extracts the identifier and it’s value from a cipher row.

3.2.2 Transaction Controller

To enable the high level services of the SemCrypt Database Manager to access the transaction capabilities of the Storage Provider the Transaction Controller provides the following operations.

Begin transaction

Parameter	Type	Description
status	OUTPUT	A status information whether the operation was successful or not.

Table 3.12: Parameters of operation ”Begin transaction”

This operation has no input parameters and begins a transaction. Beginning a transaction is only possible if there is no other transaction currently running on this client.

Commit transaction

Parameter	Type	Description
status	OUTPUT	A status information whether the operation was successful or not.

Table 3.13: Parameters of operation ”Commit transaction”

This operation has no input parameters and commits a transaction. If this operation is called and there is no active transaction this operation fails and returns an error status. The operation is called by high level services of the Storage Engine

Rollback transaction

Parameter	Type	Description
status	OUTPUT	A status information whether the operation was successful or not.

Table 3.14: Parameters of operation ”Rollback transaction”

This operation has no input parameters and tells the Storage Provider to revert all changes of the current transaction. If there was no transaction started using the "begin transaction" operation, the rollback operation fails and returns an error status. The operation is called by high level services of the Storage Engine

3.3 Storage Provider

The Storage Provider itself is not a data store by its own but it provides unified access to an external data stores. Figure 3.10 shows that the SemCrypt Database Manager does not directly communicate with the data store. A direct communication between the SemCrypt Database Manager and an existing data store is also possible but to conform to the requirement "Use arbitrary data sources" (requirement 7) it is necessary to have a layer like the Storage Provider in between the Storage Engine and the data store.

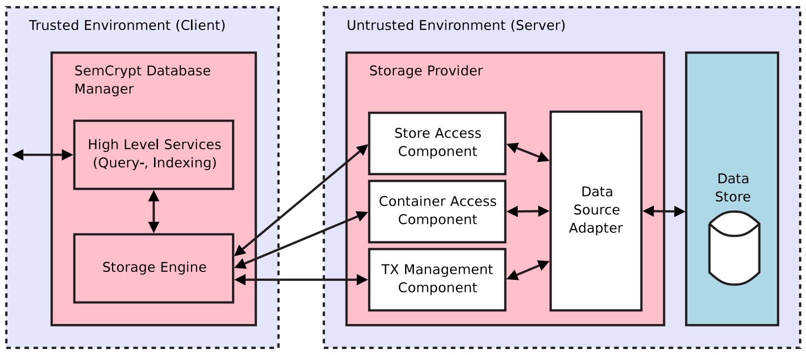
Trusted Environment (Client)
SemCrypt Database Manager
High Level Services (Query-, Indexing)
Storage Engine
Untrusted Environment (Server)
Storage Provider
Store Access Component
Container Access Component
TX Management Component
Data Source Adapter
Data Store

Figure 3.10: Storage Provider architecture

A data store that is accessible using the Storage Provider is called *store*. One store holds multiple containers. The Storage Provider provides access to multiple stores.

As shown in figure 3.10, the Storage Provider consists of the following components:

- Store Access Component - This component handles the communication between the Storage Engine and an underlying data store that is accessed using the Storage Provider.
- Container Access Component - This component handles the communication between the Storage Engine and a container of the Storage Provider.

- Transaction Management Component - Manages the transaction handling like starting, stopping and rollback operations.
- Data Source Adapter - This adapter acts as a data abstraction layer to the underlying data store.

The Store Access-, the Container Access- and the Transaction Management Component can be used by external components like those of the Storage Engine. They provide access to stores, containers and the transaction management of the Storage Provider. The Data Source Adapter is an internal component and is only used by the components of the Storage Provider

3.3.1 Store Access Component

This component provides functionality for maintaining containers in a store.

3.3.1.1 Create container

Parameter	Type	Description
store	INPUT	The store that hosts the container.
containername	INPUT	The name of the new container.
container	OUTPUT	A reference to the newly created container. If the operation fails null is returned.

Table 3.15: Parameters of operation "Create container"

This operation creates an empty container with a given name in the specified store. If the operation was successful a reference to the new container is returned.

3.3.1.2 Find container

Parameter	Type	Description
store	INPUT	The store that hosts the container.
containername	INPUT	The name of the container.
container	OUTPUT	The container that was found or NULL if no container was found with the given name.

Table 3.16: Parameters of operation "Find container"

This operation searches a container in a store using the container's name. If a container is found the operation returns a reference to the container. The container reference can be used in subsequent "store value", "retrieve value" and "remove value" operations. In case the "find container" operation fails NULL is returned.

3.3.1.3 Remove Container

Parameter	Type	Description
store	INPUT	The store that hosts the container.
containername	INPUT	The name of the container.
status	OUTPUT	A result value that indicates whether the operation was successful or not.

Table 3.17: Parameters of operation "Remove container"

This operation deletes a container named by the parameter containername with all its cipher rows from the specified store.

3.3.2 Container Access Component

This component provides operations that can be performed with cipher rows inside the containers. The interface only operates on cipher row level because the identifier-value pairs are already stored and encrypted by the Storage Engine in the cipher row and the Storage Provider has no access to decrypted data.

3.3.2.1 Insert cipher row

Parameter	Type	Description
container	INPUT	The container for the new cipher row.
cipher row	INPUT	The cipher row that has to be stored in the container
status	OUTPUT	A status information indicating whether the operation was successful or not.

Table 3.18: Parameters of operation "Insert cipher row"

This operation inserts a new cipher row in a container. If a cipher row with the same cipher id already exists in the specified container it is overwritten.

3.3.2.2 Remove cipher row

Parameter	Type	Description
container	INPUT	The container that contains the cipher row.
cipher id	INPUT	The cipher id that identifies the cipher row.
status	OUTPUT	A result value that indicates whether the operation was successful or not.

Table 3.19: Parameters of operation "Remove cipher row"

This operation removes a cipher row identified by its cipher id from the specified container. In case no cipher row with the specified cipher id is found in the container an error status is returned.

3.3.2.3 Check for cipher row existence

Parameter	Type	Description
container	INPUT	The container that contains the cipher row.
cipher id	INPUT	The cipher id that identifies the cipher row.
result	OUTPUT	Indicates if the cipher row was found or not.

Table 3.20: Parameters of operation "Check for cipher row existence parameters"

Checks if a cipher row with a given cipher id is available in the specified container and returns true if a matching cipher row was found. Otherwise false is returned.

3.3.3 Transaction Management Component

This component provides *remote access* to transactional functionality of the underlying data source. It delegates the transactional requests from the Storage Engine to the internal transaction management of the Storage Provider.

Since the Transaction Controller of the Storage Engine only acts as a proxy to the Transaction Management Component the interface is equal to the one of the Transaction Controller described in section 3.2.2

3.3.4 Data Source Adapter Component

The Data Source Adapter is an internal component of the Storage Provider that encapsulates the store access. Every access to an underlying store is routed through this component. Because the data source adapter is the only component that communicates with the underlying data source, it is also the only component that has to be exchanged when a new store type (e.g. database, filesystem, etc.) has to be supported.

The Data Source Adapter takes care of the following tasks:

- Connection Handling - This task takes care of pooling connections to the underlying data source. Because opening and closing connections for each call to a data store is a time consuming processing overhead, the connections to the data store are pooled. This also enables the sharing of connections between multiple instances of the Container Access component running in parallel.

- Transaction delegation - The transactional behavior of SemCrypt is coordinated with the store.

3.4 Summary

This section summarizes the previously identified components of the SemCrypt Store architecture.

Figure 3.11 shows all components participating in the SemCrypt Store and their relations. High level services operating within the SemCrypt Database Manager communicate with the Transaction Controller to begin, commit and rollback transactions. The Transaction Controller forwards the transaction requests to the Transaction Management Component of the Storage Provider which delegates them to the Data Source Adapter. Furthermore the high level services perform value storage and retrieval operations using the Storage Gateway. The Storage Gateway communicates with the Store Access Component to create, find and remove containers. The cipher rows of containers can be accessed via the Container Access Component.

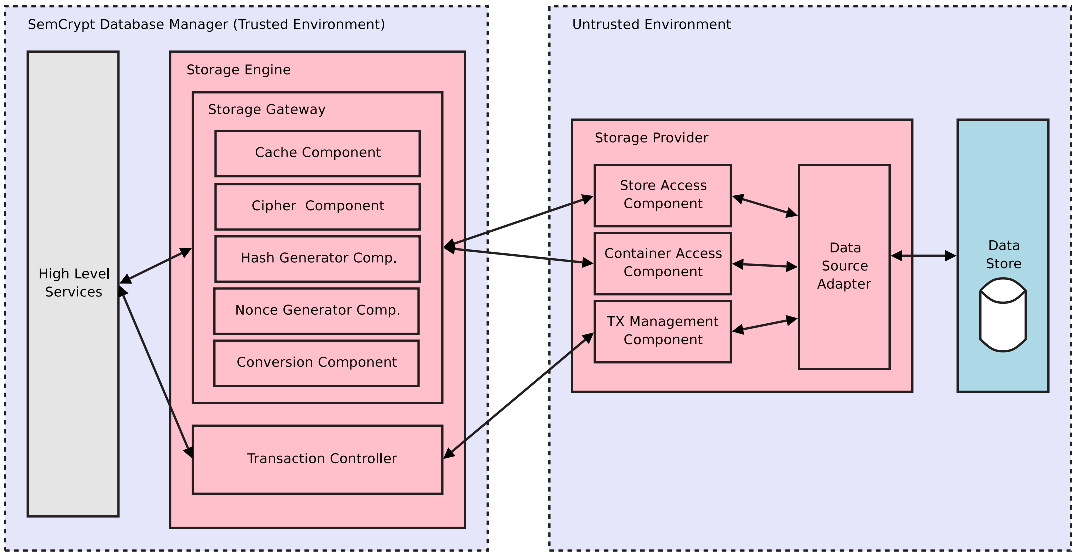

Figure 3.11: SemCrypt Store architecture summary

Chapter 4

Technologies

This chapter describes the overall technological decisions that impact the implementation of the SemCrypt Store. Each section in this chapter first legitimates the need for a technology and then briefly describes the specific technology. At the end of each section the chosen technology is outlined.

Because SemCrypt needs to run on arbitrary operating systems the Java language is chosen for the implementation of SemCrypt.

4.1 Java Enterprise Edition Platform (J2EE)

To implement the SemCrypt Store architecture, a robust server platform that is flexible and extensible is needed. Because the implementation of a distributed system like the SemCrypt Store including transaction handling and integration of third party databases is a very complex task a component based approach is chosen.

The Java Enterprise Edition (J2EE) platform is the only available enterprise platform in the Java area thus it is used as the base for the implementation of the SemCryptarchitecture. This platform already implements many features that can be used by the SemCrypt Store. Furthermore a J2EE complient architecture has to conform standard interfaces that are defined in [33] [16] [30] [15]. Using such an architecture conforms the software principles mentioned in chapter 3.

The technologies that were used in the J2EE area are listed and described in the following sections.

4.2 JBoss (Version 4.1)

Because the J2EE platform defines only the interfaces that are used to communicate between the distributed components an implementation of the J2EE Platform is needed. Such an implementation is called Application Server. There are several J2EE compliant Application Servers available[1]:

- Websphere (IBM)
- WebLogic (BEA)
- Oracle Application Server (Oracle)
- JBoss (JBoss Group)

The JBoss application server is chosen for the development of SemCryptbecause it conforms to the EJB specification (see [16]). EJBs developed in compliance with this specification can run on any J2EE compliant application server. Another reason for choosing JBoss for the development of SemCryptis that it is possible to automate the deployment of components on this application server, which shortens deployment round trips during development. The other mentioned application servers also support automated deployment but the deployment procedures are more complicated compared to the deployment on JBoss.

JBoss provides a full infrastructure for developing server side components. Beside the core JBoss Server there is also a basic EJB container and a Java Management Extension (JMX) infrastructure available allowing easy configuration of components. It also provides support for transactions and container managed persistence. The transactions are supported by an implementation of the standard Java Transaction API (JTA). [32]

In the current implementation JBoss is used to host the components of the Storage Provider.

4.3 Java Naming and Directory Interface (JNDI)

The Storage Enginehas to find the remote components that are provided by the Storage Provider. Therefore a directory service is needed enabling the Storage Providerto register its components and make them available for the Storage Engine. The Storage Engineis then able to lookup these components and use the provided functions. There are many directory service implementations available including:

[1]These are only the most common application servers

- LDAP
- Active Directory
- Java Naming and Directory Interface (JNDI)

All directory interface implementations allow clients to discover and lookup data and objects via a name that can be any string. A name is associated with an object in the directory.

Because JNDI is included in all the application server implementations mentioned above, JNDI is the directory server implementation that is used to register the components of the Storage Provider. The JNDI is used by the Storage Providerto register the components that provide external functionality (Storage Access, Container Access, Transaction Access). The Storage Enginecan use JNDI lookup queries to retrieve these objects and then are able to invoke methods of these components.

4.4 Enterprise Java Beans (EJB)

SemCryptrequires the Storage Engineto be able to control transactions. This means that all activities that are performed on the data of the Storage Provider-have to be associated with the respective transaction. Furthermore the Storage Enginehas to be able to communicate with the Storage Providerusing a remote communication mechanism.

Because enterprise java beans (EJBs) provide the ability to associate transactions with components as well as the ability to handle remote communication, EJBs are used to implement the components of the Storage Provider. This enables the Store Access, Container Access and the Transaction Management to use the following features provided by EJBs:

- Remote communication - Remote Method Invocation (RMI) makes the remote communication between Java Objects very easy.
- Transactions - EJBs enable to define very fine grained transaction boundaries. Transaction boundaries can be configured at runtime.
- Persistence - A mechanism of EJBs that swaps them to disk when they are not used anymore and restores their state when they are needed again. This approach saves memory and thus increases performance.

4.5 Java Connector Architecture (JCA)

The Storage Provider needs to be able to handle arbitrary datasources. This requires a standard mechanism for the communication between the external data source and the Storage Provider.

The components operating inside an application server are only allowed to access external datasources through defined interfaces. The two interfaces are JDBC and JCA.

JDBC defines a database specific interface for storing and retrieving data in relational databases. The J2EE Connector Architecture (JCA) defines a standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EIS). The architecture is based on the technologies that are defined and standardized as part of the J2EE platform. It addresses the key issues and requirements of EIS integration by defining a set of scalable, secure, and transactional mechanisms that enable the integration of information systems with application servers and enterprise applications [31]. JCA is a solution for connecting application servers and enterprise information systems. While JDBC is specifically used to connect applications to databases, JCA is a more generic architecture to connect to legacy systems or databases.

Because JDBC is intended for use with databases only and JCA is a more generic approach the JCA interface was chosen to act as the adapter between the Storage Provider and the underlying datasources.

4.6 Bouncy Castle Cryptographic API (Version 1.2.6)

To encrypt and decrypt data the Storage Engine needs a set of robust encryption algorithms. The most common encryption libraries available for Java are the Bouncy Castle Cryptographic API and the Flexiprovider library. Both implementations support the standard Java Cryptography Architecture.

The Bouncy Castle Cryptographic API was chosen because it supports slightly more encryption algorithms and it also includes a lightweight API suitable for use in any environment including the Java Micro Edition[2]. [3]

The various encryption algorithms implemented by Bouncy Castle Cryptographic API are used by the Cipher Component of the Storage Engine.

[2]The Storage Engine was former intended to run on a mobile device

4.7 EHCache (Version 1.1)

According to the caching strategy (strategy 5) a caching mechanism is needed for the Storage Engine. The cache should be able to swap cached items to disk when not enough memory is available. Furthermore it should be possible to suspend the cache to disk when the Storage Engine is stopped. After starting the Storage Engine again, the cache should initialize itself with the data that was stored when the Storage Engine was stopped. Table 4.7 illustrates some of the most popular cache mechanisms used in Java. [6]

Cache	Type	Cluster Safe
EHCache	memory, disk	no
OSCache	memory, disk	no
SwarmCache	clustered (ip multicast)	yes (invalidation)
JBoss TreeCache	clustered (ip multicast)	yes (replication)

Table 4.1: Cache Providers

Because of its flexibility, programmer friendly interfaces and the ability to suspend itself to disk the EHCache is chosen to cache values inside the Caching Component of the Storage Engine and avoid hotspots in the datastore of the Storage Provider. EHCache also supports various caching strategies (Last Recently Used, Least Frequently Used, First in First Out).

4.8 Berkeley Database Java Edition (Version 2.0)

The SemCrypt Store does not implement a database solution by itself but uses existing databases to act as datastores. To reduce the communication overhead between the database and the Storage Provider a very lightweight database implementation is needed that can act as the underlying datastore for the Storage Provider. Two very popular lightweight databases are MySQL and the Berkeley Database Java Edition. [2]

As table 4.8 shows the Berkeley Database storage structure is very similar to the one that is used by SemCrypt this database was chosen to be the first one that is supported by SemCrypt. The Berkeley Database Java Edition is a general-purpose, transactional, embedded database written in Java. The advantage of using a embedded database is that the communication overhead between the Storage Provider and the database is very minimal.

Table 4.8 shows a comparison of the SemCrypt Store data structures to the structures of a Berkeley Database. A Berkeley key identifies a Berkeley value.

The keys and values are stored in a tabular datastructure that is called Berkeley database. Multiple Berkeley databases can be stored in a Berkeley environment.

SemCryptdata structure	Berkeley Database data
Data source	Environment
Container	Database
Cipher id	Key
Cipher row	Value

Table 4.2: Mapping of SemCrypt data structures to Berkeley Database data structures

4.9 Log4J Logging API (Version 1.2)

To be able to diagnose errors and display useful information for the operator of SemCrypt Store, log messages are displayed. To enable a flexible configuration of the logging of messages which includes changing the log level and storing the log messages to different files, a logging framework is needed. [9]

Because the log4j logging framework is also used by the JBoss application server it was chosen to be used in SemCrypt Store too which enables a smooth integration with the logging facilities of JBoss. With log4j it is possible to enable logging at runtime without modifying the application code or restarting SemCrypt. The log4j package is designed so that the log statements can remain in the code without incurring a heavy performance cost. Logging behavior can be controlled by editing a configuration file, without touching the application binary. Log4J is used in all components of SemCrypt Store.

Chapter 5

Implementation

Contents

This chapter describes the implementation of the SemCrypt Store. The technologies that are described in chapter 4 and the architecture that is described in chapter 3 are the base for the implementation of the SemCrypt Store.

This chapter is split into two sections: The first describes the implementation of the Storage Engine, the second details on the implementation of the Storage Provider.

5.1 Storage Engine

As described in section 3.2, the Storage Engine is split into the two components Storage Gateway and Transaction Controller. Look at the subsequent sections for the implementation of both components.

5.1.1 Storage Gateway

The Storage Gateway as part of the Storage Engine performs the handling of values that are passed from high level services to the Storage Engine.

According to the architecture described in section 3.2.1, the Storage Gateway contains the components described in the subsequent sections.

5.1.1.1 Cipher Component

The implementation of the Cipher Component is a Java wrapper of specific cipher algorithm implementations provided by the Bouncy Castle Crypto API, which has been described in section 4.6. Currently the following cipher algorithms are provided:

- IDEA
- Twofish
- Rijndael

5.1.1.2 Hash Generation Component

For generating cipher ids both algorithms mentioned in chapter 2 (MD5 and SHA-1) are currently available through a Java wrapper. The MD5 hash generation uses the implementation of the Bouncy Castle Crypto API, whereas the SHA-1 algorithm is available through the standard SHA-1 implementation of the Java API.

5.1.1.3 Nonce Generation Component

The nonce generation component returns a unique number every time it is invoked. To generate a unique number, the system time of the computer on which the Storage Engine is operating cannot be used because of the risk of getting duplicate ids. Therefore the current system time (in milliseconds) is only used to initialize the nonce counter when the Storage Engine is started. Subsequently the nonce counter is incremented each time a nonce is requested.

Figure 5.1 shows the nonce generation algorithm in pseudo code[1].

[1] This basic approach has to be enhanced when multiple instances of the Storage Engine are running in parallel. In this case the uniqueness of the generated nonce values is not guaranteed anymore.

```
long generate-nonce() begin
   static long nonce;
   if (nonce is not initialized) begin
      nonce := systemtime;
   else
      nonce := nonce + 1;
   end;
   return nonce;
end;
```

Figure 5.1: Algorithm for nonce generation

5.1.1.4 Cache Component

According to section 4.7 EHCache is used as the cache implementation of the Storage Engine. Since the cache is not transactional aware, the cache may get inconsistent. This can happen because the cipher rows are written to the cache before they are stored at the Storage Provider. When a cipher row has already been written to the cache and the store operation of the Storage Provider fails, the data in the cache is inconsistent. To avoid this problem, the cache is flushed every time a rollback is performed.

5.1.1.5 Conversion Component

The conversion component creates a byte representation of the identifier-value pairs before storing them. The byte representation is converted back to an identifier-value pair when the value is retrieved from the Storage Provider. This approach enables the SemCrypt Store to handle arbitrary data. The Storage Engine accepts values of various datatypes. Internally those values are treated as byte arrays. When a value is converted to a byte array, the information about the datatype has to be integrated in the byte array to be able to restore the value. Restoring a value is necessary when a value is retrieved from the Storage Provider.

The subsequent sections explain how values and their identifiers are transformed into bytes.

5.1.1.5.1 Datatypes

All values stored by the Storage Engine need to have a datatype. The datatypes are assigned implicitly by using the respective method to store the value.

To convert a datatype to a byte representation and vice versa, Java serialization can be used but because of the strategy "No language specific features" (strategy

9) it is not possible to use the standard serialization mechanisms of the Java programming language. The reason is that it is not possible to deserialize the Java classes without having access to the typesystem of Java, which is not the case for other programming languages such as C++. The second reason for not using Java built-in serialization is that implementing a custom serialization improves performance because Java reflection is time-consuming for complex datatypes. Therefore a custom serialization of datatypes is implemented.

All datatypes have to implement a custom serialization of their value to a byte array. Figure 5.2 shows the abstract base class *Value* that has to be implemented by every datatype and defines the operations for the byte conversion.

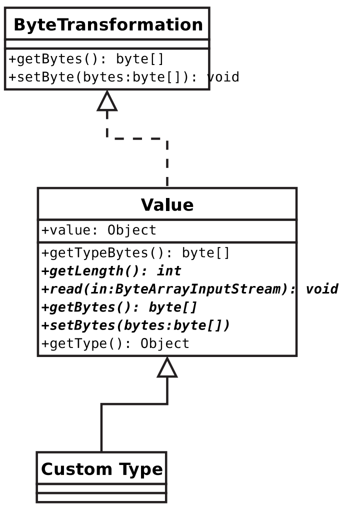

Figure 5.2: Implementation of a custom datatype

The datatype information is stored along with the identifier-value pair in the cipher row of the container. The Storage Engine distinguishes between two forms of datatypes:

- Fixed length datatypes
- Variable length datatypes

Fixed length datatypes are types whose length cannot change, even if the value is updated. This applies to datatypes like integer, double etc. Only the type information of these datatypes needs to be stored to be able to reconstruct the datatype. The length of the value can be inferred from the the type information.

Example 9 (Contract ID storage) *The identifier of Enigma's contracts is of type integer and thus has a fixed length. Figure 5.3 shows how the fixed length contract identifier of Enigma is stored in the cipher row. The data type information is stored together with the fixed length value*

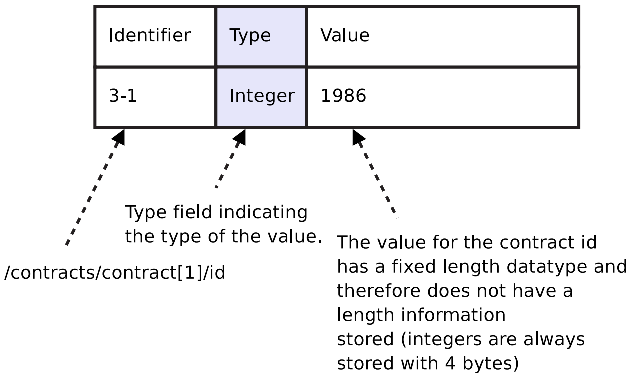
Identifier
Type
Value
3-1
Integer
1986
Type field indicating the type of the value.
/contracts/contract[1]/id
The value for the contract id has a fixed length datatype and therefore does not have a length information stored (integers are always stored with 4 bytes)

Figure 5.3: Fixed length datatype

Variable length datatypes are types are assigned to values whose length is variable (e.g. string or byte array). For variable length datatypes the length information of the value also needs to be stored. This implies that that variable length datatypes consume more space in the SemCrypt Store.

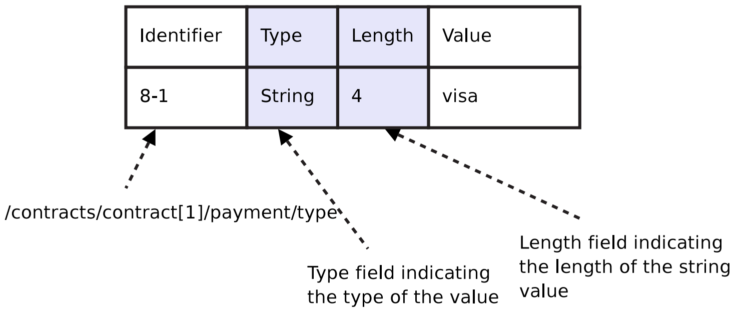
Identifier
Type
Length
Value
8-1
String
4
visa
/contracts/contract[1]/payment/type
Type field indicating the type of the value
Length field indicating the length of the string value

Figure 5.4: Variable datatype

Example 10 (Payment type storage) *The value of the payment type of Enigma's contracts can have any length, e.g. visa, master card, american express etc. Figure 5.4 shows the value "visa" whose length "4" is stored next to its type "string".*

Primitive datatypes

SemCrypt Store currently implements the followign primitive datatypes:

- String
- Integer

- Double
- Byte Array

Custom datatypes

The flexible storage implementation of the Storage Engine allows the implementation of custom datatypes. Custom datatypes enable the developers of SemCrypt Store to implement datatypes that exactly fit their needs and enable them to create own storage structures. Custom datatypes have to implement the same interfaces as primitive datatypes. Implementing custom datatypes has the advantage that the single items of complex data structures do not have to be mapped to available primitive datatypes by the high level services that use the Storage Engine. Instead the data structure can be simply passed to the Storage Engine which is responsible for serialization and deserialization.

Value lists

It is possible to implement a custom datatype that holds an array of a specific datatype. However value lists represent a generic way for storing arrays and avoid the need of an additional array-datatype for each single existing datatype.

Additionally to storing simple values, SemCrypt Store provides the concept of value lists. Value lists can be stored along with one identifier that identifies the whole value list. To support lists, the storage structure of values has to be enhanced with one more field that holds the number of values stored together with one identifier.

Example 11 (Value lists) *Figure 5.5 depicts a decrypted cipher text with an additional content field for storing lists. In this cipher text, two license terms are stored together with one identifier ("11-1"). Thus the number of the elements stored in the cipher text is "2".*

Example 12 (Conversion of identifier-value pair) *Figure 5.6 depicts how an identifier and its associated string value are converted to a byte representation which involves the following steps:*

1. *Generate a cipher id ("9327592") using the identifier "8-1".*
2. *Convert the identifier ("8-1") to a byte representation containing the length of the identifier plus the identifier itself.*
3. *Convert the value "visa" to a byte representation that holds the datatype ("String") the number of stored values ("1"), the length of the value ("4") and the value "visa".*

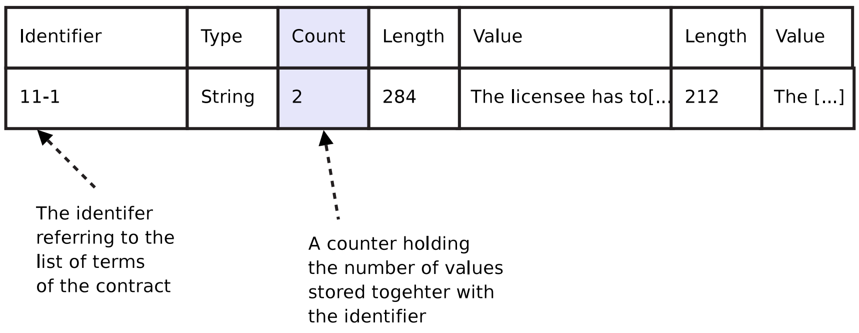
Identifier
Type
Count
Length
Value
Length
Value
11-1
String
2
284
The licensee has to[...
212
The [...]
The identifer referring to the list of terms of the contract
A counter holding the number of values stored togehter with the identifier

Figure 5.5: Value lists

5.1.2 Transaction Controller

The Transaction Management runs inside the Storage Provider but the Storage Engine exposes an interface for the SemCrypt Database Manager to control the transactional behavior. This section concentrates on how transactions are used within the SemCrypt Database Manager. An in depth description of how transactions are handled and tracked can be found in section 5.2.2. The interface exposed to the SemCrypt Database Manager by the Storage Engine conforms to the *User Transaction* interface defined in the *Java Transaction API* specification [15].

The *User Transaction* interface provides the ability to control transaction boundaries programmatically. The begin method starts a global transaction and associates the transaction with the calling thread. The transaction-to-thread association is managed transparently by the Transaction Manager, which is running on the Storage Provider.

To gain access to a transaction, the Transaction Controller has to lookup a transaction factory located on the Storage Provider. The transaction factory is then used to create a User Transaction. The Storage Engine provides access to an implementation of the User Transaction interface using a transaction factory and makes the implementation available to the SemCrypt Database Manager. The transaction factory lookup and the creation of a transaction object are shown in figure 5.7.

1. Lookup the transaction factory using the JNDI service of the Storage Provider.

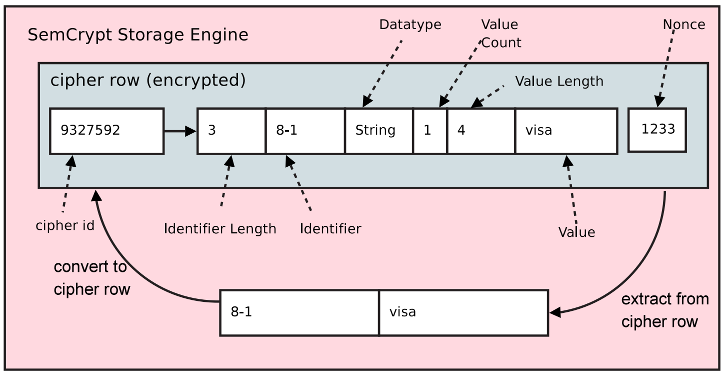

Figure 5.6: Conversion to a cipher row

2. Create a User Transaction using the Transaction Factory.
3. The User Transaction is registered at the Storage Provider.
4. A reference to the User Transaction is passed to the SemCrypt Database Manager.

The transaction object can be used to control the transactional behavior at the client side. This is done by invoking methods on the object representing the User Transaction. Here is a description of the most important methods defined by the *User Transaction* interface:

- *begin()* - The begin method of the transaction marks the start of the transaction. All the adjacent server side operations like storing and retrieving values are considered as part of one transaction.
- *commit()* - The commit method is used to mark the end of a previously started transaction. The operations performed between the commit- and the begin call are applied.
- *rollback()* - The rollback method is also used to mark the end of a previously started transaction. The operations performed within the transaction are reverted and are not applied to the underlying data source.
- *setRollbackOnly()* - Marks a transaction so that the only possible outcome of the transaction is a rollback operation.
- *getStatus()* - The getStatus method returns the status of the transaction. If no transaction is running, a status is returned indicating that there is no active transaction.

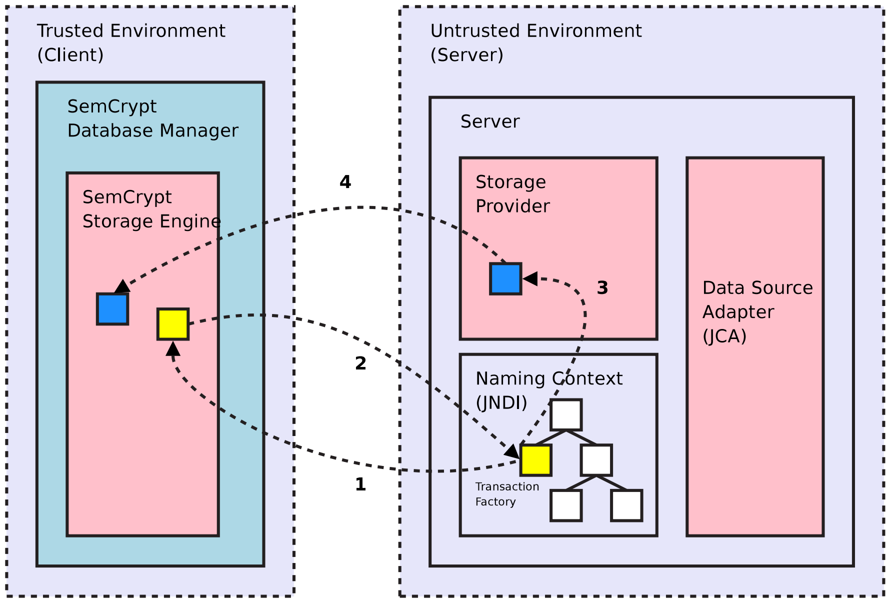

Figure 5.7: Transaction Control

- *setTransactionTimeout()* - The setTransactionTimeout method receives a numerical value representing milliseconds as a parameter and defines the maximum time a transaction is able to be active. If the transaction exceeds the given time, which means that the commit or rollback is not called within this time, an exception is thrown and the transaction is rolled back.

On the server side, the *Java thread id* is used to identify a specific session of the client. This implies that there can be only one active transaction per client thread. Or in other words, transactions are not allowed to overlap.

5.2 Storage Provider

The Storage Provider architecture is split into three main components, the Storage- & Container Access, the Transaction Management and the Data Source Adapter. These three parts are described in the subsequent sections. The Storage Provider is implemented in a modular way enabling future versions of the Storage Provider to plug in more components providing enhanced functions.

5.2.1 Storage- & Container Access Components

The storage access component and the container access component are implemented using EJBs. This enables a straight forward remote communication between the Storage Engine and these two components because remote method invocation is used.

Example 13 *Figure 5.8 depicts the lookup of a remote object by the Storage Engine. The Storage Engine gets a reference to the Store Access Component. With this reference, the Storage Engine is able to invoke operations on this component.*

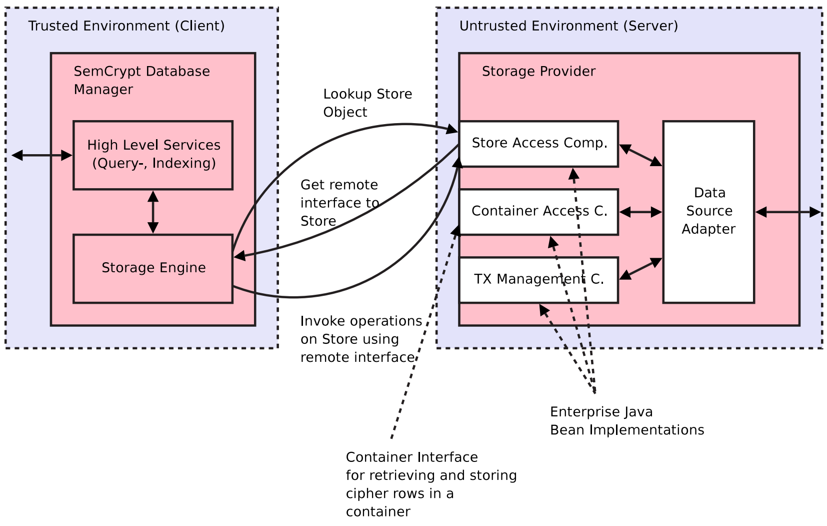
Trusted Environment (Client)
SemCrypt Database Manager
High Level Services (Query-, Indexing)
Storage Engine
Untrusted Environment (Server)
Storage Provider
Store Access Comp.
Container Access C.
TX Management C.
Data Source Adapter
Lookup Store Object
Get remote interface to Store
Invoke operations on Store using remote interface
Enterprise Java Bean Implementations
Container Interface for retrieving and storing cipher rows in a container

Figure 5.8: Accessing the Storage Provider

5.2.2 Transaction Management Component

For implementing the transaction management mechanisms between the data source and the Storage Provider, interfaces defined by JCA are used.

To control transactions the two types of transaction demarcations available in a J2EE environment are used:

- Programmatic transaction demarcation
- Automatic transaction demarcation

With *programmatic* transaction demarcation and the User Transaction interface defined by the JTA specification, it is possible to explicitly demarcate the transaction boundaries. This approach is used in most of the SemCrypt Database Manager calls when storing values. It forces the user of the Storage Engine to explicitly open a transaction with the method call *begin()* and close the transaction either with *commit()* or *rollback()*. If a data manipulation operation like insert or update is performed without opening the transaction first, an error is thrown.

When *automatic* transaction demarcation is used, the Storage Provider manages the transaction automatically. By default, all the read only operations (all get operations) use an automatic transaction demarcation if there is no programmatic transaction available. This means that it is not necessary to explicitly open a transaction before invoking read-only operations. Read-only operations start and stop a transaction implicitly for the duration of the method call.

5.2.3 Data Source Adapter

This section explains how the SemCrypt Store handles the connections to the underlying data source. The Data Source Adapter operates inside the SemCrypt Store and handles the connection management to the underlying data source. The data store currently used is a Berkeley Database. The Berkeley Database was chosen because it is a very lightweight database implementation that has a similar access mechanism as the one that is used by SemCrypt Store.

As displayed in figure 5.9, the components inside the SemCrypt Store do not directly access the data store but use the Data Source Adapter for accessing the external data store. In particular this section focuses on the need of connection pooling and describes the different scenarios under which connection pooling is accomplished.

5.2.3.1 Connection Management

The SemCrypt Database Manager uses a connection factory to obtain a connection. It then uses the connection to connect to the provided data source. Because connections are expensive to create and destroy, they are pooled and managed by the Data Source Adapter. This leads to better scalability and performance. It is very common that the number of connections to the Storage Provider is much higher than the connections between the Data Source Adapter and the data source. The Data Source Adapter enables the physical connections to the Berkeley Database to be shared among the logical connections provided to the applications accessing the adapter. The Java Connector Architecture (JCA)

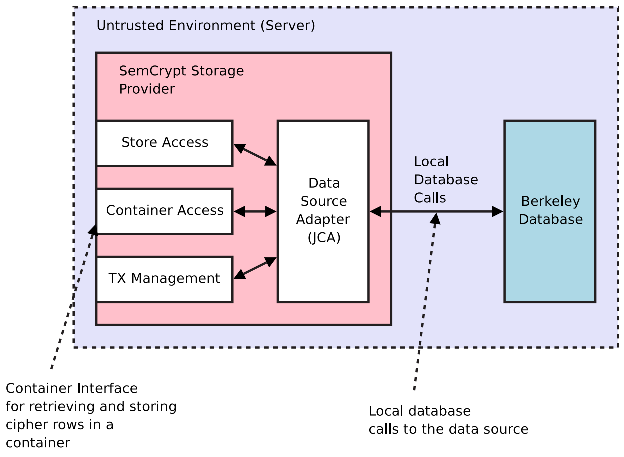
Untrusted Environment (Server)
SemCrypt Storage Provider
Store Access
Container Access
TX Management
Data Source Adapter (JCA)
Local Database Calls
Berkeley Database
Container Interface for retrieving and storing cipher rows in a container
Local database calls to the data source

Figure 5.9: Data Source Adapter

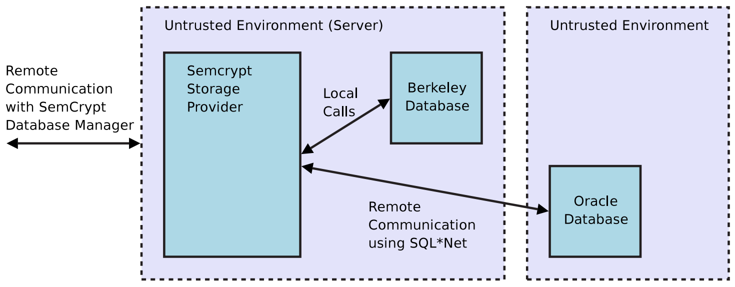
Untrusted Environment (Server)
Untrusted Environment
Remote Communication with SemCrypt Database Manager
Semcrypt Storage Provider
Local Calls
Berkeley Database
Remote Communication using SQL*Net
Oracle Database

Figure 5.10: Different Datasources

provided by the Java Enterprise Edition Platform (J2EE) support connection pooling. The connection pooling support is transparent to the methods inside the Storage Provider.

5.2.3.2 Data storage

One data source represents one database instance that is used by SemCrypt Store. There can be various different data source instances configured for the use with the Storage Provider. Each of these data source instances has to provide a data source configuration including specific parameter settings and a JNDI name that is used to access the data source. The different data sources are accessible by the Storage Engine using simple names that uniquely identify the data source.

Figure 5.10 shows one local database that operates on the same host as the SemCrypt Store and a different database that provides remote access and therefore is able to run on a different server than the SemCrypt Store.

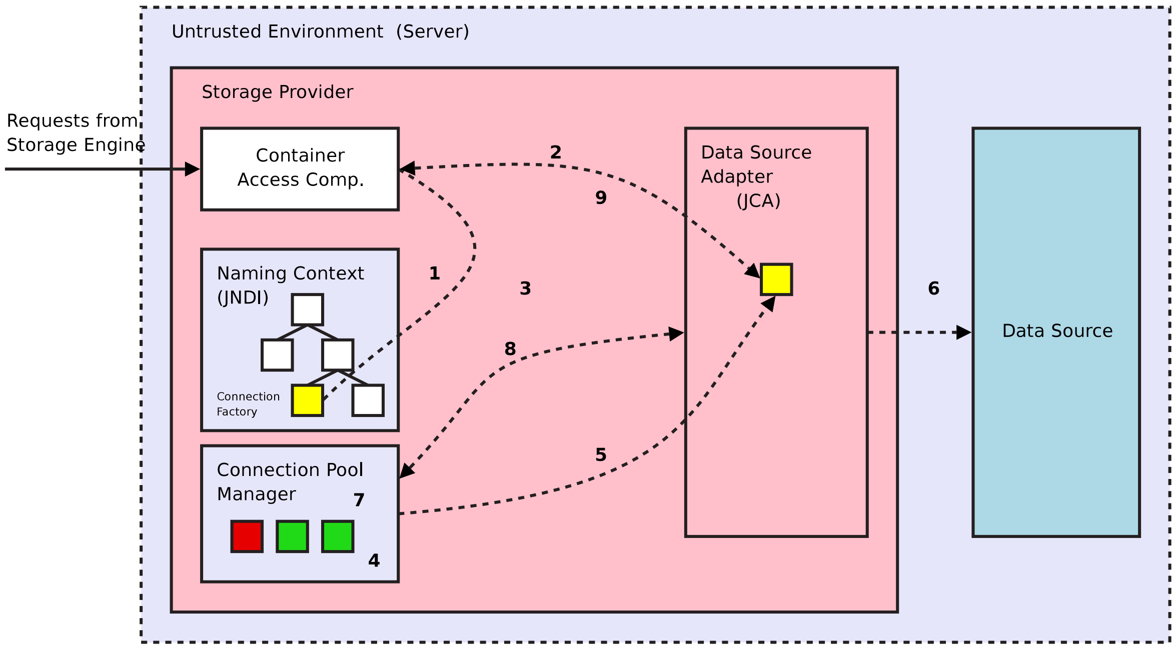

Figure 5.11: Connection Management

The advantage of this concept is that the Storage Engine that accesses the Storage Provider does not need to know the details of the databases used. The Storage Engine does not even have to know the type and the amount of databases used by the Storage Provider.

5.2.3.3 Connection Management Architecture

The Data Source Adapter provides interfaces for the Storage Provider to create a connection to the underlying data source using a connection factory. The Storage Provider accesses the data source using the Data Source Adapter. The steps of accessing the data source are defined in the Java Connector Architecture specification. However, before using the Data Source Adapter is has to be configured. Configuring means that the Data Source Adapter has to be informed which data source it has to use and what parameters are used when accessing the data source. Thereby, a connection factory has to be configured in the JNDI (Java Naming and Directory Interface) namespace of the Storage Provider.

Figure 5.11 briefly describes the scenario of how to establish a connection to the data source via the Data Source Adapter:

1. A JNDI lookup is performed to retrieve the configured connection factory (the connection factory is configured using an XML file).

2. After retrieving the connection factory, a method is invoked on the connection factory object to obtain a connection to the data source.
3. The connection to the data source is not established immediately, but first the connection request is forwarded to the connection pool manager.
4. An attempt is made to find a suitable existing connection in the connection pool.
5. If no suitable connection is found in the pool, the Data Source Adapter is used to create a new physical connection (also known as a managed connection).
6. The Data Source Adapter creates a new managed connection by establishing a physical connection to the data source.
7. Then the newly created connection is added to the connection pool.
8. The Container component uses the connection to access the data source.
9. When the connection is not used anymore by the Container component, it is closed and marked free in the connection pool.

The connection management used by the SemCrypt Store offers the following benefits:

- The components operating within the Storage Provider do not have to care of connection pooling because the Data Source Adapter takes care of transparent connection pooling.
- Using connections to the data source is very simple.
- It is possible to install multiple Data Source Adapters all accessing different data sources.
- The access to the external data source is controlled by one component.

Chapter 6

Related Work

This chapter describes work related to the SemCrypt Store. Various secure data store approaches are described and their advantages as well as their disadvantages are presented. At the end of this chapter, the approaches are compared to SemCrypt Store.

6.1 Hacigümüs - Executing SQL over Encrypted Data

Hakan Hacigümüs et al. [19] address the problem of storing and querying encrypted relational data. They suggested to store encrypted tuples of relations. The tuples are never decrypted on the server. To store a relation in an encrypted way, it is suggested to split the domain values of the attributes of the relation into partitions. The tuples are then assigned to those partitions. When requesting a tuple from the server the whole partition has to be fetched.

Example 14 (Hacigümüs - Partitions) *As an example, consider a relation "contracts" that stores information about license contracts with the attributes id, licensee and price. The relation is shown in table 6.1.*

id	licensee	price
23	basf	2300
860	ibm	86000
320	dell	32000
875	hp	87500

Table 6.1: Relation contracts

Regarding the attribute id, assume that the domain values of id lie in the range [0,1000]. This range is divided into the following 5 partitions [0,200], (200,400], (400,600], (600,800], (800,1000].

Identification functions (i) are used to assign a unique identifier to each partition allowing to determine in which partition a value is stored.

Example 15 (Hacigümüs - Identification function) *Each partition of the previous example is assigned an identifier: i([0,200])=2, i((200,400])=7, i((400,600])=5, i((600,800])=1, i((800,1000])=4*

Given the above partitions and identifier functions, mapping functions (m) are defined that map values to the identifiers of the partitions to which the values belong.

Example 16 (Hacigümüs - Mapping function) *Table 6.2 shows some contract ids and the corresponding map values.*

id value	23	860	320	875
m(id value)	2	4	7	4

Table 6.2: Mapping functions

Mapping functions are applied to the attributes of all tuples that will either be selected or participate in query statements. The map values act as a coarse index and are stored together with the respective encrypted tuple. All attributes of the tuple are encrypted and stored in one cipher text.

Example 17 (Hacigümüs - Encrypted relation) *Table 6.3 shows the encrypted relation "contracts" as it is stored on the insecure server.*

encrypted tuple	id	licensee	price
123423423423422	2	12	50
274379742472322	4	17	21
345453453534545	7	5	65
377907987998732	4	23	80

Table 6.3: Encrypted relation

To retrieve values, this technique allows partial execution of a query on the server side where the map values are used to retrieve the correct partitions of values.

The correct result of the query is found by decrypting the result of the server side query and executing a compensation query on the client side. This approach has also been adopted by Jammalamadaka [20].

However this approach has several potential security holes. According to Fong [17], the scheme is semantically insecure and not robust against frequency analysis attacks because the encryption is always performed with the same encryption key (no nonce is stored together with the tuples). Additionally, an adversary can figure out the identification functions if there are only few different input values, enabling the adversary to perform statistical analysis on the indices. Fong also claims that the approach of encrypting all fields of a tuple together raises a potential security leak. If the client only needs one particular attribute value of a tuple, the corresponding encrypted tuple has to be entirely decrypted and thus all attribute values are revealed to the client.

6.2 Oracle Database Encryption Techniques

Oracle uses authentication mechanisms to secure data in the database, but not in the operating system files where the data is stored. To protect the data files of the database, Oracle provides transparent data encryption. Thereby, Oracle encrypts sensitive data in database columns stored in operating system files. To prevent unauthorized decryption, it stores encryption keys in a security module external to the database. It is possible to define which columns of a table need to be encrypted [25].

To prevent equal encrypted storage patterns when the input of the cipher algorithm is equal, Oracle adds a value to the data that makes encrypted values different even if the input data is the same. The value that is added is called "salt".

Example 18 (Oracle encryption) *Figure 6.1 illustrates how to create a table with encrypted columns. The statement creates a table that holds the contract information of Enigma including the id and the licensee that are stored in plain text. The price and the payment information are encrypted. The payment card id is encrypted with the "no salt" option which suppresses the use of the "salt" value for this column. This is because the column payment_cardid is indexed and the "salt" option based encryption cannot be used for indexed columns.*

```
CREATE TABLE contracts (
     id NUMBER(5) NOT NULL,
     licensee VARCHAR2(128),
     price NUMBER(6) ENCRYPT USING '3DES168',
     payment_type VARCHAR2(32) ENCRYPT USING '3DES168',
     payment_cardid VARCHAR2(32) ENCRYPT NO SALT,
     payment_valid DATE ENCRYPT USING '3DES168'
);
CREATE INDEX idx01 ON contracts(payment_cardid);
```

Figure 6.1: Contracts table

This approach has some major drawbacks. The structure of the storage is not hidden making it possible to figure out the values of columns, specifically if they contain very few different values. It is also possible to determine the number of rows stored making it possible for an adversary to figure out the number of contracts that the software company stores. Oracle uses "salt" values that are added to the regular data to avoid the same encrypted text patterns for the equal plain texts in the database. However this feature can only be applied to columns that are not indexed.

6.3 XML Encryption

XML Encryption [12] is a recommendation of the W3C Consortium to encrypt the content of XML documents. W3C develops XML encryption as an enabler for trusted and secure semantic web services. The recommended approach aims at not encrypting the XML document as a whole but only data that needs to be kept secure.

Example 19 (XML Encryption) *Figure 6.2 shows Engima's contracts encrypted and stored in XML.*

```
<?xml version='1.0'?> <contracts
xmlns='http://www.engima.com/contracts'>
   <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
      xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
         <CipherValue>A23B45C56</CipherValue>
      </CipherData>
   </EncryptedData>
</contracts>
```

Figure 6.2: An XML containing all encrypted contracts

This solution only covers the encryption of XML documents. However it does not provide a solution for querying the encrypted data, thus it is inappropriate for providing a secure data store since this approach does not support searching within encrypted text.

6.4 Encrypting files & filesystems

Encrypting files can be performed with two different approaches:

- *Encryption of the whole disk* - Cryptographic filesystems, e.g. the *cryptographic filesystem (cfs)* [4], implement encryption at system level through a standard filesystem interface to encrypted files. Files in directories (as well as their pathname) are transparently encrypted and decrypted with the specified key without further user intervention.
- *Encryption of single files* - File based encryption systems, e.g. EncFS [7], encrypt single files instead of whole filesystems. The advantage in comparison to a filesystem encryption approach is that files can easily be backed up and the encryption mechanism is separated from the filesystem.

Encrypted filesystems and encryption of single files provide security against offline attacks, like a stolen notebook or stolen backups.

Example 20 (Encrypted files) *An encrypted storage approach for contracts using encrypted files is depicted in figure 6.3. Every time data of a contract is needed at the client the whole contract has to be requested and transferred to the client.*

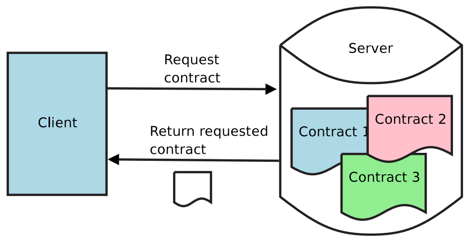

Figure 6.3: Contracts stored as files

Both filesystem approaches have the disadvantage that they do not support transactional behavior. Furthermore they always use the same key for encrypting files or data resulting in repetitive storage patterns which can be exploited using frequency analysis. The main disadvantage however is that only whole files and not parts of them can be retrieved from the server.

6.5 Summary

Table 6.5 compares secure data store appraoches described in this chapter with SemCrypt Store. The comparison is based on the requirements listed in chapter 2. Only those requirements that need to be fulfilled by a secure data store are taken into account. Additionally features relevant for the secure operation of each approach are compared. These additional features are compared in the second part of table 6.5.

Requirements	Hacigümüs	Oracle	Files	XML	SemCrypt
Protect structural information	no	no	no	no	yes
Encrypt/Decrypt data on client only	yes	yes	no	no	yes
Avoid statistical analysis	no	yes	no	no	yes
Support arbitrary cipher alg.	yes	yes	yes	yes	yes
Ensure ACID principles	yes	yes	no	no	yes
Granularity of encryption	row	field	file	field	field
Support of index structures on encrypted data	yes	no	no	no	yes
Lightweight components	yes	no	yes	yes	yes
Secure operation in untrusted environments	yes	no	no	yes	yes
Secure cipher key management	yes	no	yes	yes	yes

Table 6.4: Comparison of secure data stores

Protect structural information This requirement is only supported by SemCrypt Store. Hacigümüs's approach does not support this requirement because the attributes that are required to query data have to be stored in addition to the encrypted tuples. In Oracle the structure of the table is always visible. When encrypting data in files, the file names are not encrypted and can be used to infer the content or structure of the file. The XML Encryption approach does not hide the structure of a document because usually only sensible data is encrypted.

Encrypt/Decrypt data on client only Hacigümüs's approach only encrypts data on the client. Oracle also transfers the encrypted data to the client without decrypting it on the server. Encrypted files are decrypted on the server and then transmitted to the client in plain text. XML Encryption approach does not support this requirement either because it only concentrates on the encryption of XML and does not provide a storage solution involving a client and a server.

Avoid statistical analysis Only Oracle and SemCrypt Store provide a strategy to avoid statistical analysis. Oracle meets this requirement by using it's "salt" option, whereas SemCrypt Store uses a nonce based approach encrypting every value with a different cipher key.

Support arbitrary cipher alg. All above solutions support at least a variety of encryption algorithms. Not all are as flexible as SemCrypt and allow the use of custom encryption algorithms but all support at least a set of predefined algorithms.

Ensure ACID principles Hacigümüs's approach has ACID support because this approach can be implemented using any relational database. Oracle also supports ACID transaction support. The file storage and the XML Encryption, however, do not consider ACID support.

Granularity of encryption The compared approaches encrypt values using different levels of granularity. Hacigümüs encrypts all attributes of a tuple together in one cipher text. The file based approach encrypts whole files. All other approaches perform the encryption on field level.

Support of index structures on encrypted data Hacigümüs supports index structures using map values. Oracle does only support indexes on data that is not encrypted using the "salt" option, thus Oracle does not support maximum security when using indexes. Encrypted Files do not support index structures since the data is stored in one single cipher text. XML Encryption also does not support index structures inside encrypted XML tags (only plain XML can be indexed). SemCrypt Store supports arbitrary index structures. To store index structures, the same strategies as for storing data using SemCrypt Store are applied.

Lightweight components Hacigümüs's approach can be implemented using lightweight components that operate on mobile devices like e.g. handhelds or cellphones. The Oracle database does not support such environments. Furthermore encrypted files and encrypted XML documents can be used with devices that require lightweight implementations. Since the SemCrypt Store is built very flexible it is possible to implement it to use lightweight components and thus being able to be embedded in applications operating on mobile devices.

Secure operation in untrusted environments This feature requires that no data is encrypted in a untrusted environment. Hacigümüs supports this feature because data is never decrypted on the server. Oracle stores encrypted data in files but has to decrypt the data on the server to be able to process queries. This represents a potential security leak because the decrypted data is available in the server-memory. Encrypted files are usually decrypted on the server before they are submitted to the client, thus do not support this feature. Encrypted XML documents can be transferred to the client before they are encrypted and thus are not decrypted on the server. SemCrypt Store supports this feature as well because data is only encrypted by the Storage Engine operating in a client environment.

Secure cipher key management File- and XML Encryption as well as the approach of Hacigümüs can implement a secure cipher key management. According to Kornbrust [23], Oracle maintains it's cipher key in plain text in the memory of the database server, thus does not support a secure cipher key management. The SemCrypt Store never exposes the cipher key to the server. The cipher key is always kept in a trusted environment which implies that SemCrypt Store meets the feature "secure cipher key management".

The comparison shows that SemCrypt Store is the only approach that supports the requirements listed in table 6.5. Thus SemCrypt Store is the most adequate solution to build a secure remote data store.

Chapter 7

Conclusion and Outlook

Contents

This chapter concludes this thesis and gives an outlook on future enhancements and additional features that are not yet implemented. The ideas for additional features outlined in section 7.2 are the base for further research and development activities.

7.1 Conclusion

This thesis has outlined the security problems in the ASP business that prevent organizations to outsource their data storage. The most common security problems that were identified are: eavesdropping, data theft, data tampering, data loss and privacy protection.

According to these problems, the requirements for a secure datastore like *data encryption*, *securing the structure of XML documents* and *hiding the associations of stored data* were stated. The strategies to meet these requirements were described. These strategies include *encryption of data*, storing data in an *identifier value based approach* and using *hash functions* to hide the structure of XML documents.

Based on the developed strategies an architecture for a secure data store named SemCrypt Store was introduced. SemCrypt Store is designed to perform the encryption and decryption of data in a trusted environment, whereas the encrypted data is stored in an untrusted environment. SemCrypt Store consists of two main components - the Storage Engine, which operates in an untrusted environment and performs encryption and decryption of values, and the Storage Provider, which operates in an untrusted environment and enables unified access to arbitrary databases. The storage structures used by SemCrypt Store were designed in a way that it is possible to query the encrypted data.

The technologies used to implement SemCrypt Store were described and it was shown how these technologies were used to implement remote communication and transaction control, encryption/decryption, caching and data storage.

The important implementation details, including the usage of the J2EE platform and the use of an application server were described, thereby aiming at reflecting the architecture design.

Furthermore existing secure storage solutions of Hacigümüs et al, the Oracle encryption approach, encrypted filesystems and the approach of storing encrypted XML were briefly described. The different approaches were compared to the SemCrypt Store approach. The comparison revealed that only SemCrypt Store meets the requirements of a secure remote data store.

7.2 Future Work

This section presents suggestions for improving and extending the current implementation as well as extending the current architecture of SemCrypt Store.

7.2.1 Improving the implementation

The current implementation can be improved in the following ways:

- *More primitive data types* - The implementation of more primitive datatypes will be necessary to make the overall implementation of the current SemCrypt Store more powerful and usable.
- *Decentralized Cache* - In the current implementation, the Storage Engine only works with caching enabled when it is the only Storage Engine accessing the Storage Provider because the current cache implementation is not cluster aware. This means that the cache does not synchronize its content with the caches of other Storage Engine instances. The implementation of a more sophisticated caching will allow to connect multiple Storage Engine instances to the Storage Provider without running into inconsistency problems because of different caches operating inside different Storage Engine instances.

7.2.2 Extending the implementation

The implementation of the SemCrypt Store can be extended by bulk messages. Grouping or bulking multiple storage requests into one single request will speed up the overall communication of the Storage Engine with the Storage Provider. This is because network overhead that normally is applied to each single message that is sent between those two components can be reduced when sending one larger message instead of several small messages to the Storage Provider. Furthermore bulk messages help to reduce calls inside the Storage Provider, thus saving processing time.

7.2.3 Extending the architecture

The architecture of the SemCrypt Store can be extended by the following features:

- *Tamper evident storage* - A tamper-evident storage enables to detect whether data has been altered outside the SemCrypt Store. Cryptographic hash functions or cryptographic signatures can be used to add a tamper evident layer of protection to the Storage Provider. The usage of such hash functions is often referred to as an electronic signature. Tamper control in SemCrypt can be achieved by generating a signature (hash value) for each cipher row and store it together with the cipher row. Any change to the cipher row will cause it to have a different hash which will make the signature invalid. Changes can be detected by comparing stored signatures to the computed hash values of the stored data. If the hashes do not match, data has been changed.

- *Tamper resistant storage* - In comparison to tamper-evident stores where changes to the store can be performed but are able to be detected, a tamper-proof or tamper-resistant storage cannot be changed when not permitted [13]. Tamper resistance is currently researched in the area of smart cards and electronic devices.

- *Security* - In a future version of SemCrypt Store, the Storage Engine can be split into an trusted and a untrusted part. Only encryption related operations like the cipher component of the current Storage Engine will then execute in a trusted environment which may be a smartcard. This requires the cipher component to be a very lightweight and very small implementation of the cipher module. The remaining parts of the SemCrypt Store can then operate in an untrusted environment [21].

- *Hashvalues* - In the current approach, hash functions are used to generate cipher ids. Because it is possible to figure out the value that was used by the hash function to generate the cipher id by using a brute-force search [1], the generation of cipher ids has to be improved. A possible solution is to additionally encrypt the cipher id before it is stored.

- *Authorization* - Authorization as part of SemCrypt protects the operations that can be performed with data by only allowing users to perform these operations that have been granted authority to use them. For example, certain users may be granted permissions to alter the data maintained by SemCrypt while others are only allowed to view the data. It is therefore necessary to restrict the access to certain data according to user groups.

  It has to be defined how fine grained the *access control* has to be implemented considering the *operations* that can be performed with SemCrypt and the *users* operating SemCrypt using different *roles*.

  Example 21 *Regarding the example of the software company Enigma, authorization is required to allow sales representatives to view or modify only their contracts but not the contracts of other sales people*

[1]Brute force search is performed by systematically enumerating every possible identifier and comparing the hash value of the identifier to the stored hash value until a value possible value is found.

List of Figures

List of Tables

Bibliography

[1] Ant. http://ant.apache.org, October 2005.

[2] Berkeley Database. http://www.sleepycat.com, October 2005.

[3] Bouncy Castle Cryptographic API. *http://www.bouncycastle.org*, 2005.

[4] CFS - The cryptographic filesystem. http://net-tex.dnsalias.org/ stefan/nt/unix/cfs.html, November 2005.

[5] Eclipse. http://www.eclipse.org, October 2005.

[6] EHCache. http://ehcache.sourceforge.net/, November 2005.

[7] EncFS Encrypted Filesystem. http://encfs.sourceforge.net/, October 2005.

[8] JUnit. http://www.junit.org, October 2005.

[9] Log4J. http://logging.apache.org, September 2005.

[10] Osterman Research Reveals Security as Prime Obstacle in Outsourcing, and more. *Wall Street and Technology*, 28 July 2005. http://www.wstonline.com (ArticleID 166403368).

[11] XDoclets. http://xdoclet.sourceforge.net, October 2005.

[12] XML Encryption. http://www.w3.org/Encryption/2001/, October 2005.

[13] Anderson R. and Kuhn M. Tamper Resistance - a Cautionary Note. *USENIX Association*, 2, 21 November 1996. (ISBN 1-880446-83-9).

[14] Brinkman R., Doumen J., and Jonker W. Using secret sharing for searching in encrypted data. *Secure Data Management*, 2004.

[15] Cheung S. and Matena V. Java Transaction API. Technical report, Sun Microsystems, 1 November 2002.

[16] DeMichiel L. Enterprise Java Beans Specification, Version 2.1. Technical report, Sun Microsystems, Inc., 12 November 2003.

[17] Fong K. Potential Security Holes in Hciguemues Scheme of Executing SQL over Encrypted Data. http://www.cs.siu.edu/ kfong/research/database.pdf, April 2005.

[18] Gruen K., Karlinger M., and Schrefl M. Schema-aware Labelling of XML Documents for Efficient Query and Update Processing in SemCrypt. *Technical Reports, University of Linz Department of Data and Knowledge Engineering*, 2005.

[19] Haciguemues H. Executing SQL over Encrypted Dat ain the Database-Service-Provider Model. *ACM SIGMOD Int. Conf. on Management of Data*, pages 216–227, 2002.

[20] Jammalamadaka R. Querying Encrypted XML Documents. Master's thesis, University of California, Irvine, 2004.

[21] Karlinger M. and Gruen K. Design Specification - WP 2 - Encrypted XML Processing and Design of Basic Prototype.

[22] Kemper A. and Eickler A. *Datenbanksysteme eine Einfuehrung*, volume 3. Oldenbourg, 1999.

[23] Kornbrust A. Circumvent Oracle's Database - Encryption and Reverse Engineering of Oracle Key Management Algorithms. July 2005.

[24] Microsoft. Encrypting File System overview. *http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/encrypt_overview.mspx*, 2005.

[25] Oracle. Oracle 10g Documentation. October 2005.

[26] Pernul and Unland. *Datenbanken im Unternehmen.* Oldengourg, 2003.

[27] Schneier B. *Angewandte Kryptographie.* Addison-Wesley, 1996.

[28] Schneier B. and Ferguson N. *Practical Cryptography.* John Wiley & Sons, 2003.

[29] Schrefl M., Dorn J., and Gruen K. SemCrypt Ensuring privacy of electronic documents through semantic-based encrypted query processing. *PDM 2005 International Workshop on Privacy Data Management*, 8 April 2005.

[30] Shannon B. Java2 Platform Enterprise Edition Specification, v1.4. Technical report, Sun Microsystems, Inc., 24 November 2003.

[31] Sharma R., Stearns B., and Ng T. *J2EE Connector Architecture and Enterprise Application Integration.* Addison Wesley, 2001.

[32] Stark S. *JBoss Administration and Development Third Edition*. JBoss Group, December 2003.

[33] Sun Microsystems Inc. J2EE Connector Architecture Specification. Technical report, Sun Microsystems, Inc., November 2003.

[34] Weinstein L. Outsourced and Out of Control. *Communications of the ACM*, 47(2), February 2004.

[35] Xiaoyun W., Dengguo F., Xuejia L., and Hongbo Y. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. 17 August 2005.

Appendix A

Development Environment

Contents

This chapter describes the development environment used to implement the SemCrypt Store prototype. The development environment consists of tools for Java code generation, unit testing and for building the prototype. The tools and APIs are shown and briefly described. Each section outlines the purpose of the tool, then gives a brief explanation of the tool and finally states in which part of the prototype it has been used.

A.1 Eclipse (Version 3.1)

Eclipse [5] supports the development of Java applications including testing and debugging.

Eclipse is an open source community project with the focus on providing an extensible development platform as well as application frameworks for building software. Its tools and frameworks span the software development life cycle, including support for modeling, language development environments for Java, C/C++ and other programming languages, testing and performance, business intelligence, rich client applications and embedded development.

Eclipse was used as development environment to implement the SemCrypt Store prototype in Java.

A.2 XDoclet (Version 1.2)

XDoclet [11] is a set of tools to automatically generate additional Java code and XML files required for EJBs.

XDoclet is an open source code generation engine. It enables attribute-oriented programming for Java. In short, this means that one can add more significance to code by adding meta data (attributes) with the help of special JavaDoc tags. The tags are then used to instruct XDoclet which Java code to generate.

XDoclet tags were used to generate the deployment descriptors and remote interfaces for the EJBs that are implemented in the Storage Provider.

A.3 JUnit (Version 3.8)

JUnit [8] is a unit testing framework enabling developers to quickly and easily implement test cases for unit and regression testing.

The prototype uses JUnit to test the existing functionality and to perform regression tests.

A.4 Ant (Version 1.6)

Ant enables to build Java projects automatically. This includes generating source code, building and packaging the project.

Ant [1] is a pure Java build tool. Like the make tool, it allows to define a buildsystem and dependencies among the targets within this buildsystem. This makes it easy to compile Java code because Ant is fully integrated into the Java environment.

Ant is used within the SemCrypt Store project as a buildsystem which automatically generates source code (interfaces) and JavaDoc and compiles source files.

Appendix B

Installation and Configuration

Contents

This chapter describes the installation of the SemCrypt Store. Furthermore the configuration of the two components of the SemCrypt Store, theStorage Engine and the Storage Provider is described.

B.1 Installation

The Storage Provider is stored as a preconfigured package on the installation CD. The only requirement is that the environment variable JAVA_HOME points to a Java home directory containing a JDK 1.5 installation.

To start the Storage Provider listening on the address 127.0.0.1, change to the bin directory of the Storage Provider and run *run.bat -b 127.0.0.1* on a command shell.

The Storage Engine is also stored as a preconfigured package on the installation CD in the directory "storageengine". Because the Storage Engine is an API and thus is not designed to be an executable program, testcases are provided that use the Storage Engine API. To start the testcases, execute the "run-test.bat" script in the "storageengine" directory.

For a more detailed installation description, refer to the README.TXT file included in the main directory of the SemCrypt Store CD.

B.2 Configuration

This section describes the configuration capabilities of the Storage Engine and the Storage Provider. The properties of each configuration are described and a sample configuration is given.

B.2.1 Configuring the Storage Engine

To configure the Storage Engine, a properties file is provided that holds the following settings:

- The Context settings hold the information about how to connect to the SemCrypt Store. The context setting includes the following configuration values:

  java.naming.factory.initial - This is the class that is used for the JNDI lookups.

  java.naming.provider.url - This is the address of the host where the JNDI server can be found.

  jnp.disableDiscovery - If set to true, this property enables the automatic discovery of a Storage Provider in the network.

- Hash settings

  semcrypt.hashimpl - Specifies the hash implementation class that is used by the Hash Generation Component of the Storage Engine.

- Encryption settings

  semcrypt.cipherimpl - Specifies the encryption algorithm implementation class that is used by the Storage Engine.

  semcrypt.cipherkey - The cipher key that is used for encryption and decryption.

- Nonce settings

  semcrypt.nonceimpl - Specifies the nonce implementation class that is used by the Storage Engine.

- Cache settings

  semcrypt.enablecache - If set to true, this property activates the cache for the Storage Engine. The cache properties are configured in a separate configuration described in B.2.1.1.

The following is a typical configuration of the Storage Engine that can be found in the file "semcrypt.properties" in the "storageengine" directory of the installation.

```
################################################################################
# Context settings
################################################################################
java.naming.factory.initial =
org.jnp.interfaces.NamingContextFactory java.naming.provider.url =
jnp://127.0.0.1:1099
jnp.disableDiscovery = false

################################################################################
# Hash settings
################################################################################
semcrypt.hashimpl = com.semcrypt.dbmanager.hash.MD5Hash

################################################################################
# Encryption settings
################################################################################
# semcrypt.cipherimpl = com.semcrypt.dbmanager.cipher.DESede
# semcrypt.cipherimpl = com.semcrypt.dbmanager.cipher.IDEA
# semcrypt.cipherimpl = com.semcrypt.dbmanager.cipher.Rijndael
# semcrypt.cipherimpl = com.semcrypt.dbmanager.cipher.Twofish
semcrypt.cipherimpl = com.semcrypt.dbmanager.cipher.DES
semcrypt.cipherkey = 1234567890123456789

################################################################################
# Nonce settings
################################################################################
semcrypt.nonceimpl = com.semcrypt.dbmanager.nonce.SimpleNonce

################################################################################
# Cache settings
################################################################################
semcrypt.enablecache = false
```

B.2.1.1 Cache configuration

The cache can be configured to meet the requirements of the system in which SemCrypt Store is operating.

The mandatory configuration values are listed and described below:

- *diskStore path* - The path that is used for storing the cache elements when they are swapped to disk.
- *maxElementsInMemory* - The maximum number of elements to store in memory (Note: not on disk!). This configuration value may hold integer values between 0 and the maximum possible value of an integer. Due to performance reasons, it is strongly recommended that this value is at least set to 1. If not, a warning will be issued when the cache is created.
- *eternal* - Defines whether or not the cache is eternal. An eternal cache does not expire its elements. Possible values are true or false.

- *overflowToDisk* - Indicates whether or not to use the disk when the number of elements exceeds the maxElementsInMemory of the memory. When an overflow occurs, the elements that are removed from memory are determined using a specified eviction policy (last recently used, last frequently used, first in first out), whereby used means stored in the cache or retrieved from the cache. This attribute can be true or false.

The optional configuration values for the cache are:

- *timeToIdleSeconds* - This is the number of seconds the element should live after being accessed. The value can be any valid integer starting at 0. The default value is 0 (which means forever).
- *timeToLiveSeconds* - This is the number of seconds that the element should live since it has been inserted into the cache (this does not include read operations on this element). The default value is 0 (which means forever).
- *diskPersistent* - If set to true, the elements of the cache are preserved between shutdowns of the Java VM.
- *diskExpiryThreadIntervalSeconds* - This the interval of a thread that runs and checks the values on the disk if they have expired. The value can be any valid integer starting at 0. Setting this value to 0 is not recommended because this will lead to a very high CPU usage. The default value is 120.
- *memoryStoreEvictionPolicy* - This is the policy that is enforced upon reaching the maxElementsInMemory limit.

The following cache configuration is the default configuration for the cache of SemCrypt Store and is stored in the file ”semcrypt-nodecache.xml” in the ”storageengine” directory.

```
<ehcache>
    <!-- disk store location -->
    <diskStore path="java.io.tmpdir"/>
    <!-- default cache configuration -->
    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        overflowToDisk="true"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
        />
</ehcache>
```

B.2.1.2 Logging configuration

The log output of the Storage Engine is produced using Log4J 4.9. It is possible to configure the amount of logs to be written and where to store or display the logs. The configuration is stored in a file named *log4j.xml* in the "storageengine" directory.

B.2.2 Configuring the Storage Provider

The following transactional parameters can be configured for the Storage Provider:

- *Locking timeout* - The locking timeout is used to define a timeout for locks that are held by SemCrypt Store.
- *Read only* - Defines if the data in the store is writable or not.
- *Transactional* - A boolean parameter that defines whether the specific store in the Storage Provider is transactional or not.
- *Transaction timeout* - This parameter is used to configure the timeout for transactions to complete.
- *Nowait* - This parameter is used to configure the transaction to not wait if a lock request cannot be immediately granted. If set to true, the Storage Provider will not wait for a lock.
- *Read uncommitted* - This parameter can be true or false and configures the isolation level which defines if dirty reads are allowed. If set to true, a transaction may see uncommitted changes made by some other transaction.

The following is the default configuration of the Data Source Adapter. Note that if certain properties are not configured, the default properties of the underlying database are applied.

```
<connection-factories>
   <tx-connection-factory>
      <jndi-name>semcrypt/adapter</jndi-name>
      <!--use-java-context>false</use-java-context-->
      <xa-transaction/>
      <rar-name>semcrypt.adapter.rar</rar-name>
      <connection-definition>com.semcrypt.adapter.AdapterConnectionFactory</connection-definition>
      <adapter-display-name>SemCrypt JCA Adapter</adapter-display-name>
      <config-property name="EnvironmentHome" type="java.lang.String">
      ${jboss.server.data.dir}${/}semcrypt</config-property>
      <config-property name="ReadOnly" type="java.lang.Boolean">false</config-property>
      <config-property name="AllowCreate" type="java.lang.Boolean">true</config-property>
      <config-property name="Transactional" type="java.lang.Boolean">true</config-property>
   </tx-connection-factory>
</connection-factories>
```

B.2.2.1 Logging configuration

The logging configuration of the Storage Provider follows the same concepts as the logging of the Storage Engine that is described in B.2.1.2.

Appendix C

Programmatic use of the Storage Engine

Contents

This chapter shows the programmatic use of the Storage Engine.

C.1 Storing and retrieving a value

The example below illustrates the usage of the Storage Engine and shows which information needs to be provided to store and retrieve a value. The example stores a value "visa" with the identifier "8-1" in the container "Contracts". After the value "visa" is stored, it is retrieved from the container by searching it using the identifier "8-1".

1. The Storage Provider URL is retrieved from a configuration file.
2. The name of the *store* is provided (It identifies one store that is made accessible by via the SemCrypt Store.)
3. The container is found by searching it using its name "Contracts".
4. The identifier "8-1" and the value "visa" are passed to the setString function.
5. Retrieve value with identifier "8-1".

```
// Step 1 - the retrieval of the storageprovider URL is not shown in this
// example because the URL is read automatically from a config file.
TransactionFactory txFactory = new TransactionFactory(); Transaction
tx = txFactory.getTransaction();

tx.begin(); Store store = new ClientStore("ejb/ContractStore"); //
Step 2 Container container = store.createContainer("Contracts"); //
Step 3 tx.commit();

// normal transaction, set the credit card type
tx.begin(); container.setString("8-1".getBytes(), "visa"); // Step 4
tx.commit();

String value = container.getString("8-1".getBytes()); // Step 5
// value = "visa"
```

C.2 Transaction handling

This section presents examples of how to use the transaction functionality of SemCrypt Store.

The following code shows an example of how to use a manipulation operation with a transaction.

```
...
// get a transaction object from the server
Transaction tx = txFactory.getTransaction();

// lookup the store
Store store = new ClientStore("ejb/Store");

// create a container
tx.begin();
Container container = store.createContainer("USContracts");
tx.commit();

// begin a transaction, store a value in the container
// and commit the transaction
tx.begin();
container.setString("8-1".getBytes(), "visa");
tx.commit();
...
```

The next example results in an error when applying the value to the container (setString) because no transaction has been previously initiated.

```
...
// get a transaction object from the server
Transaction tx = txFactory.getTransaction();

// lookup the store
Store store = new ClientStore("ejb/Store");

// create a container
tx.begin();
Container container = store.createContainer("EMAContracts");
tx.commit();
```

```
// ERROR!
container.setString("8-1".getBytes(), "visa");
...
```

Note that in the following example, no explicit transaction is initiated. However the *getString()* operation successfully reads the string from the container because the Storage Provider automatically creates the transaction before the method is invoked and closes the transaction when the method has finished.

```
...
// get a transaction object from the server
Transaction tx = txFactory.getTransaction();

// lookup the store
Store store = new ClientStore("ejb/Store");

// create a container
tx.begin();
Container container = store.createContainer("EMAContracts");
tx.commit();

String value = container.getString("8-1".getBytes());
...
```

Overlapping transactions are not allowed:

```
 ...
// begin a transaction, and try to begin a second
// transaction
tx.begin(); container.setString("8-1".getBytes(), "visa");
tx.begin(); // ERROR!!! container.setString("8-1".getBytes(),
"mastercard"); tx.commit(); tx.commit(); ...
```