Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра компьютерной безопасности

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
канд. техн. наук, доцент
_____ С.А. Останин
«__22__» ____января____ 2020 г.

ДИПЛОМНАЯ РАБОТА

РАЗРАБОТКА И РЕАЛИЗАЦИЯ МОДУЛЯ ДЛЯ ОБЕСПЕЧЕНИЯ ЗАЩИЩЕННОГО ХРАНЕНИЯ И ПОИСКА КОНФИДЕНЦИАЛЬНЫХ ДАННЫХ

по основной образовательной программе подготовки специалистов
«Анализ безопасности компьютерных систем»
специальности подготовки 10.05.01 Компьютерная безопасность

Абрамов Сергей Александрович

Руководитель ВКР
канд. техн. наук, доцент
_____ В.А.Беляев
«__20__» ____января____ 2020 г.

Автор работы
студент группы №__1145__
_____ С.А. Абрамов

Томск – 2020

Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра компьютерной безопасности

УТВЕРЖДАЮ
Руководитель ООП
канд. техн.наук, доцент
_____ С.А. Останин
«__9__» _____сентября_____ 2019 г.

ЗАДАНИЕ

по подготовке ВКР специалиста студенту Абрамову Сергею Александровичу группы №1145.

1. Тема ВКР «Разработка и реализация модуля для обеспечения защищенного хранения и поиска конфиденциальных данных».

2. Срок сдачи студентом выполненной ВКР:
а) на кафедре 22 января 2019 года;
б) в ГЭК 31 января 2019 года.

3. Исходные данные к работе:
Цель работы: разработка и реализация настраиваемого модуля с использованием технологии Node.js для защищенного хранения конфиденциальных данных с возможностью поиска:

- по точному совпадению;
- по ключевым словам.

Объект исследования: схемы шифрования с возможностью поиска.
Методы исследования: теоретические, практические.

4. Краткое содержание работы: обзор существующих схем шифрования с возможностью поиска и выбор критериев для сравнения и анализа достоинств и недостатков (к 01.10.2019), выбор алгоритмов шифрования на основе проведенного анализа (к 01.11.2019), разработка и реализация настраиваемого модуля согласно выбранным алгоритмам поиска (к 01.12.2019), тестирование, анализ результатов тестирования и написание ВКР (к 31.12.2019).

5. Работа выполняется по заданию кафедры компьютерной безопасности Томского государственного университета.

6. Дата выдачи задания «__5__» _____сентября_____ 2019 г.

Руководитель ВКР
канд. техн. наук, доцент _____/ Беляев В.А./

Задание принял к исполнению, _____/ Абрамов С.А./
студент гр.1145

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ 8
1 Обзор схем шифрования с возможностью поиска в зашифрованных данных 10
1.1 Симметричное шифрование с сохранением свойств 10
1.2 Функциональное шифрование 11
1.3 Шифрование с использованием забывчивой памяти 12
1.4 Гомоморфное шифрование 13
1.5 Симметричное шифрование с возможностью поиска (SSE) 14
1.6 Сравнение существующих схем шифрования 14
2 Особенности существующих SSE схем шифрования 16
3 Описание динамической SSE схемы шифрования 18
3.1 Построение зашифрованного индекса 20
3.2 Алгоритм поиска по ключевому слову 21
3.3 Алгоритм добавления файла 22
3.4 Алгоритм удаления файла 23
4 Слепое усеченное индексирование 25
5 Программная реализация 26
5.1 Выбор технологий 26
5.2 Структура программы «SecureDataStorage» 26
5.3 Используемые криптографические примитивы 29
6 Оценка скорости работы модуля и анализ полученных результатов 30
6.1 Оценка скорости при разном количестве уникальных слов в документах 30
6.2 Оценка скорости при разной конфигурации модуля и без использования модуля 32
6.3 Анализ полученных результатов 35
ЗАКЛЮЧЕНИЕ 37
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 38
ПРИЛОЖЕНИЕ А. Основные функции «SecureDataStorage» 41
ПРИЛОЖЕНИЕ Б. Класс Engine 42
ПРИЛОЖЕНИЕ В. Класс Encryption 48
ПРИЛОЖЕНИЕ Г. Класс WIndex 50

РЕФЕРАТ

Дипломная работа содержит 57 страниц, 6 разделов, 1 рисунок, 4 прилож., 30 использованных источников.

КЛЮЧЕВЫЕ СЛОВА: ШИФРОВАНИЕ С ВОЗМОЖНОСТЬЮ ПОИСКА, ЗАШИФРОВАННЫЙ ИНДЕКС, SSE, БЕЗОПАСНОСТЬ В ОБЛАЧНЫХ БАЗАХ ДАННЫХ, ШИФРОВАНИЕ БАЗЫ ДАННЫХ, ШИФРОВАНИЯ НА КЛИЕНТСКОЙ СТОРОНЕ.

Область исследования: шифрование данных, которое позволяет в дальнейшем производить процедуру поиска в зашифрованной коллекции документов без предварительной расшифровки.

Цель работы: разработка и реализация настраиваемого модуля с использованием технологии Node.js для защищенного хранения конфиденциальных данных с возможностью поиска:

а) по точному совпадению;

б) по ключевым словам.

В работе приведен обзор современных подходов поискового шифрования и осуществлен их сравнительный анализ. На основе полученных результатов теоретической части работы реализован настраиваемый модуль, который позволяет шифровать данные с возможностью дальнейшего поиска и проведено тестирования данного модуля на скорость работы.

ВВЕДЕНИЕ

В связи с резким увеличением популярности облачных технологий, в частности облачных баз данных, становится актуальным вопрос защиты конфиденциальных данных в интернете.

Согласно статистике компании Infowatch с 2007 по 2019 год утекло более 30 млрд записей конфиденциальных данных, из них 20 млрд за последние два года [1]. На фоне таких неутешительных данных в сочетании с повышенной заботой о конфиденциальности отдельных лиц, отрасли и правительства во всем мире предприняли серию инициатив, направленных на повышение безопасности, снижение уровня мошенничества и защиту конфиденциальной информации, включая:

- NIST: Guide to protecting the Confidentiality of PII;
- General Data Protection Regulation (GDPR);
- Payment Card Industry Data Security Standard (PCI DSS);
- Health Insurance Portability and Accountability Act (HIPAA);
- Федеральный закон РФ № 152-ФЗ «О персональных данных».

К требованиям стандартов и законов по защите конфиденциальных данных относится обеспечение конфиденциальности и целостности данных с помощью применения криптографических алгоритмов и протоколов, в том числе алгоритмов шифрования данных.

Несоблюдение международных стандартов и федеральных законов может повлечь наложению больших штрафов и отказ от сотрудничества большинства компаний и клиентов. Поэтому компании, которые хранят информацию попадающие под эти стандарты и законы, заинтересованы в её грамотной обработке.

Шифрование в компьютерных системах может быть реализовано как аппаратными, так и программными средствами. С помощью программных средств можно настроить шифрование как целых разделов файловой системы, так и отдельных файлов. Примерами таких программных средств являются: eCryptfs, EncFS, PEFS, VeraCrypt и др. Использование таких средств шифрования может снизить производительность и усложнить администрирование, так как системному администратору требуется доступ к операционной системе для использования шифрования на уровне файлов.

Подход шифрования на уровне базы данных отчасти устраняет вышеперечисленные недостатки. Популярные базы данных Oracle, SQL Server, PostgreSQL и MongoDb поддерживают шифрование выбранных столбцов и технологию прозрачного шифрования (TDE), которая позволяет шифровать данные перед записью на диск и расшифровывать во время чтения в память. Преимуществом TDE является то, что шифрование и расшифровывание выполняются прозрачно для приложений, то есть их модификация не требуется.

Если использовать шифрование на уровне базы данных и хранить данные на не доверенном удаленном сервере, то возникает угроза утечки данных, так как ключи шифрования хранятся на сервере. Несмотря на многочисленные заявления облачных провайдеров о безопасности их сервисов мы не можем считать их до конца безопасными, поскольку:

- при передаче по сети данные могут быть перехвачены с использованием специального программного обеспечения;
- сервер может быть подвержен хакерским атакам, что позволит злоумышленникам похитить информацию;
- недобросовестный администратор сервера может ее расшифровать и воспользоваться в личных целях.

За последние годы рынок облачных баз данных стремительно увеличился. По оценке аналитиков Gartner к 2022 году 75 % баз данных либо уже станут облачными, либо будут пребывать в процессе миграции в облако [2].

Шифрование на клиентской стороне является решением многочисленных проблем сохранности конфиденциальной информации в облачных базах данных. При шифровании на клиентской стороне процесс шифрования и расшифрования осуществляется приложением, которое создает или изменяет данные перед записью в базу данных и во время чтения из нее. Этот подход является более гибким, так как приложению известны правила доступа пользователей, а также информация о том, какие данные являются конфиденциальными. Преимуществом такого шифрования является то, что нет необходимости использовать дополнительные решения для защиты данных при передаче по сети, так как они отправляются уже зашифрованными. Ещё один плюс такого метода — это то, что хищение конфиденциальной информации становится сложнее, так как злоумышленник должен

иметь доступ к приложению для того, чтобы расшифровать данные, хранящиеся в базе данных.

Однако существует проблема, которая появляется при использовании данного подхода для поиска данных в зашифрованной коллекции, находящейся в удаленной базе данных. Для решения данной проблемы существует несколько подходов. Один из них предлагает загрузить всю базу данных на клиентский компьютер, расшифровать ее локально и произвести поиск уже по открытым данным. Данное решение существенно снижает скорость поиска, нагружает компьютерную сеть и требует наличие большого объема памяти. Другой подход позволяет удаленной базе данных расшифровать данные, выполнить запрос на стороне сервера и вернуть пользователю результат. В этом случае снижается уровень безопасности, так как данные, защищенные шифрованием, раскрываются серверу базы данных.

В данной работе рассматривается решение проблемы поиска информации в зашифрованной базе данных с использованием поискового шифрования (searchable encryption - SE). Данный подход позволяет обеспечить максимально возможную функциональность поиска при минимальной потери конфиденциальности.

В качестве языка программирования, с помощью которого решается проблема поиска в зашифрованной на уровне приложения базе данных, выбран JavaScript (JS), выполняемый в среде Node.js. Программная платформа Node.js с каждым годом стремительно развивается и набирает популярность. Стандартная библиотека Node.js обладает широким набором возможностей, в том числе содержит реализацию популярных криптографических алгоритмов. Наличие общедоступного репозитория npm позволяет использовать готовые решения в виде отдельных модулей. Среди существующих решений для Node.js не было найдено настраиваемого модуля, который позволяет безопасно хранить информацию и одновременно обеспечивать эффективный поиск в зашифрованных документах.

Создание настраиваемого модуля для защищенного хранения конфиденциальных данных с возможностью поиска позволит упростить разработку будущих проектов на Node.js.

Целью данной работы является разработка и реализация настраиваемого модуля с использованием технологии Node.js для защищенного хранения конфиденциальных данных с возможностью поиска:

а) по точному совпадению;

б) по ключевым словам.

В ходе достижения цели были решены следующие задачи:

1. Обзор существующих схем шифрования с возможностью поиска;
2. Сравнение схем шифрования по выбранным критериям с анализом их достоинств и недостатков;
3. Выбор алгоритмов шифрования на основе проведенного анализа:

а) с возможностью поиска по точному совпадению;

б) с возможностью поиска по ключевым словам в зашифрованном документе;

4. Разработка модели реализации настраиваемого модуля для защищённого хранения и обработки в базе данных;
5. Реализация настраиваемого модуля согласно разработанной модели и выбранными алгоритмами поиска;
6. Тестирование реализованного модуля на примере базы данных MongoDB;
7. Анализ результатов тестирования.

В основной части работы в главе 1 описаны существующие модели поиска по зашифрованным данным. Во второй главе определены критерии и проведено сравнение схем шифрования выбранной модели. В главе 3 приведено описание работы выбранного алгоритма для поиска по ключевым словам. В главе 4 описан алгоритм поиска по точному совпадению. В 5 главе перечислены особенности программной реализации: структура программы, используемые компоненты, варианты использования системы. В главе 6 описана методология проведения экспериментов, их результаты и анализ применимости разработки. В заключении приведены выводы по проделанной работе и возможные дальнейшие направления работы в этой области.

ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ

Конфиденциальная информация – совокупность данных, хранящихся в виде бумажных документов или сведений на цифровых носителях, доступных для передачи заранее определенному узкому кругу лиц.

Модуль – функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом, предназначенный для использования в других программах. Модули позволяют разбивать сложные задачи на более мелкие.

База данных (БД) – это организованная структура, предназначенная для хранения, изменения и обработки информации.

JSON (JavaScript Object Notation) - формат обмена данными, основанный на JavaScript.

Документная БД - эти тип баз данных, предназначенный для хранения данных в виде документов, подобных JSON.

Коллекция – объект БД, содержащий в себе набор документов.

Шифрование – обратимое преобразование информации в целях скрыть его суть. Шифрованное сообщение называется шифротекстом [3].

Расшифрование – процесс преобразования множества зашифрованных сообщений во множество открытых сообщений с использованием ключа [4].

Криптографические примитивы – низкоуровневые криптографические алгоритмы, которые часто используются для построения криптографических протоколов. В узком смысле это операции и процедуры, определяющие требуемые свойства криптосистемы [4].

Криптографическая стойкость – свойство криптографического шифра противостоять криптоанализу, то есть анализу, направленному на изучение шифра с целью его расшифрования. Криптостойкость шифра есть его важнейшая характеристика, которая отражает то, насколько успешно алгоритм решает задачу шифрования.

Симметричное шифрование – это способ шифрования, в котором для шифрования и расшифровывания применяется один и тот же криптографический ключ.

Блочное шифрование – разновидность симметричного шифрования, оперирующего группами бит фиксированной длины - блоками.

Шифрование с возможностью поиска (searchable encryption - SE) – класс криптографических алгоритмов шифрования, в которых существует возможность осуществлять поиск по зашифрованным данным (файлам, записям базы данных и т. д.) [5].

Криптосистема - семейство криптографических функций в совокупности с используемыми криптографическими протоколами [6].

Токен - это сгенерированное значение, связанное с конкретным запросом к БД.

1 Обзор схем шифрования с возможностью поиска в зашифрованных данных

Впервые формальное научное описание схемы шифрования с возможностью поиска было предложено в 2000 году в статье Song, Wagner, Perrig [7]. Сегодня академическим сообществом предложено более 50 описаний схем с разнообразной функциональностью.

Схемы SE позволяют производить поиск по зашифрованным данным без получения информации об исходных текстах. Некоторые схемы реализуют поиск с помощью техник шифрования, в других схемах на стороне клиента генерируется специальный индекс, используемый для поиска и обработки, который сохраняется вместе с зашифрованными данными на стороне сервера [29, 30].

1.1 Симметричное шифрование с сохранением свойств

Симметричное шифрование с сохранением свойств (Property-preserving encryption - PPE) – это криптосистема, сохраняющая определённые свойства исходного сообщения в шифротексте. Известны различные схемы PPE алгоритмов.

Простейшим примером является детерминированное шифрование [8], при котором каждому открытому тексту сообщения ставится в соответствие ровно один шифротекст, то есть совпадение шифров говорит о том, что соответствующие им открытые тексты были одинаковыми [9].

К другим примерам PPE криптосистем относятся схемы шифрования, сохраняющие порядок (Order-preserving encryption - OPE). Схема шифрования с сохранением порядка шифрует набор сообщений m_1 и m_2, $m_1 < m_2$, таким образом, что их шифротексты c_1 и c_2 раскрывают порядок сообщений: $c_1 < c_2$.

С помощью PPE схем шифрования можно реализовать поиск как по точному совпадению, так и поиск по ключевым словам в тексте. Применительно к поиску по ключевым словам в тексте, на примере детерминированного шифрования, используется следующая схема: в базе данных хранится соответствие между документами коллекции и зашифрованными словами, содержащимися в них; во время поиска слова, сервер ищет вхождение ключевого слова по каждому документу коллекции.

Преимуществом схем шифрование с сохранением свойств является скорость работы. При поиске по точному совпадению - скорость работы алгоритма O $(\log(n))$. Однако, при поиске по ключевым словам скорость работы алгоритма значительно ниже, O (nm), где n - количество документов, m - количество слов. Если хранить детерминированные шифры с помощью структур данных, поддерживающий быстрый поиск, например бинарное дерево поиска, то скорость работы увеличится.

Недостатком этого класса алгоритмов является то, что детерминированность шифров позволяет отслеживать частоту появления ключевых слов и целых зашифрованных документов, что делает базу данных уязвимой к анализу частоты. Также возможна утечка повторяемости поисковых запросов [10].

1.2 Функциональное шифрование

В 2001 г Boneh и Franklin предложена схема Identity-based encryption (IBE) [12], которая реализует криптосистему с открытым ключом. Для данной схемы существует главная пара ключей (master private key, master public key) и набор отдельных секретных ключей, которые сформированы на основе главного закрытого ключа и некоторого идентификатора. Сообщения шифруются с помощью главного открытого ключа, а с помощью отдельных секретных ключей вычисляется некоторая функция от идентификатора и зашифрованного сообщения.

В 2004 г в статье [25] было предложено использовать схему IBE для поиска по зашифрованным данным. Идея заключается в использовании отдельных ключевых слов, содержащихся в документе, в качестве идентификаторов для секретных ключей. Шифрование производится на основе ключевого слова в качестве идентификатора. Набор шифров, каждый из которых соответствует отдельному слову в файле, хранится на удаленном компьютере. С помощью секретного ключа, выработанного с помощью ключевого слова в качестве идентификатора, сервер распознает принадлежность слова отдельному документу.

В 2010 году в статье авторами Boneh, Sahai и Waters [11] сформулировано определение функционального шифрования (Functional encryption - FE) в основе которого лежит асимметричная криптосистема, которая помимо главной пары ключей (master private key, master public key) поддерживает дополнительный секретные

ключи. Дополнительные секретные ключи позволяют узнать лишь значение некоторой функции от зашифрованных данных.

Рассматриваемая схема имеет несколько недостатков:

- нотация IBE [25] не гарантирует, что шифр скрывает информацию об идентификаторе.
- медленная работа алгоритма поиска, связанная с необходимостью расшифровать все зашифрованные элементы, соответствующие паре документ – слово.
- риск атаки удаленного компьютера путем генерации шифров с помощью главного открытого ключа и определенного словаря [14].

1.3 Шифрование с использованием забывчивой памяти

С точки зрения безопасности одним из наиболее надежных является подход к поиску в зашифрованных данных с использованием забывчивой памяти (Oblivious RAM - ORAM). Данный протокол взаимодействия с памятью был предложен в 1996 году [15] и предполагает запутывание шаблона доступа к хранилищу, то есть при считывании и записи информации само устройство не знает, в каких областях памяти были произведены операции.

Протокол сокрытия схемы доступа к хранилищу реализован следующим образом: система включает в себя клиент, у которого есть маленький объем защищенной памяти (забывчивой памяти), и сервер с большим объемом незащищённой памяти. Клиент может использовать память сервера для хранения и получения данных. Клиент хранит внутренний секретный ключ симметричного шифрования, который использует для зашифрования данных перед записью на сервер и расшифрования данных после их получения. Клиент при помощи двустороннего протокола может получить i-й элемент памяти в качестве выходного значения, в то время как входными параметрами являются номер элемента памяти и секретный ключ. На сервер протокол чтения ничего не возвращает, получая на вход забывчивую память – некую область памяти. Протокол записи в свою очередь получает от пользователя номер элемента памяти и значение для записи, ничего ему не возвращая. С сервера протоколу записи передается забывчивая память, и ее обновленная версия отправляется обратно на сервер.

Подобные протоколы позволяют сконструировать решение для поиска по зашифрованным данным. Со стороны сервера для этого необходимо наличие двух элементов забывчивой памяти ORAM1 и ORAM2. ORAM1 хранит в себе зашифрованные документы, а ORAM2 – структуру данных, поддерживающую операцию поиска, например, обратный индекс в зашифрованной форме. В результате чего, процедура поиска сводится к набору обращений к серверу через протокол чтения, а обновление коллекции файлов – через протокол записи. «Забывчивость» используемых структур определяет безопасность данного подхода. Подобный подход к поиску скрывает от сервера практически все. Однако относительно низкая скорость поиска является недостатком использования ORAM, что накладывает ограничения на размер хранимых коллекций и вычислительные ресурсы [16].

1.4 Гомоморфное шифрование

Понятие гомоморфного шифрования впервые было формализовано в 1978 году Ronald Rivest, Len Adlemanи и Michael Dertouzos [17]. Данный тип шифрования позволяет делать произвольные вычисления на зашифрованных данных без их расшифровки. Гомоморфное шифрование различается на частично гомоморфные и полностью гомоморфные системы. Первые поддерживает только одну операцию: сложение или умножение, а вторые гомоморфны относительно двух операций [18].

Определение. Полностью гомоморфное шифрование – это криптосистема, которая позволяет по зашифрованным данным $Enc(m_1),\ldots,Enc(m_t)$ и произвольной функции f вычислить шифртекст $Enc(f(m_1,\ldots,m_t))$ без знания ключа расшифровывания sk и исходных данных $m_1,\ldots,m_t$ [19].

Задача создания полностью гомоморфной криптосистемы оставалось нерешенной свыше 30 лет. Только в 2009 году исследователь из IBM Craig Gentry теоретически обосновал возможность создания гомоморфной криптосистемы шифрования и предложил одну такую криптосистему [19]. Криптосистема Gentry на практике оказалась нереализуемой, и вскоре появились работы, предлагающие модификации, которые улучшают производительность криптосистемы Gentry.

Частным примером гомоморфного шифрования является поиск по ключевым словам в зашифрованных данных. В данном случае удаленный компьютер получает зашифрованный поисковой запрос и применяет алгоритм поиска для нахождения

документов, содержащих определенное слово. Недостатком такого подхода является линейное время выполнения запросов по отношению к коллекции зашифрованных документов, так как для поиска документов по ключевому слову требуется проверить все зашифрованные файлы.

1.5 Симметричное шифрование с возможностью поиска (SSE)

Симметричное шифрование с возможностью поиска (symmetric searchable encryption - SSE) - это подход, который с самого начала был направлен на решения проблемы взаимодействия клиента с сервером, который не может считаться доверенным.

SSE схемы позволяет клиенту шифровать данные таким образом, чтобы впоследствии он мог генерировать токены поиска в качестве запросов на сервер. По отправленному токену сервер может выполнять поиск по зашифрованным данным и возвращать соответствующие зашифрованные файлы.

Данный подход пытается найти баланс между эффективностью и безопасностью. В частности, SSE схемы шифрования столь же эффективны, как и решение на основе детерминированного шифрования, но при этом они уступают по безопасности лишь решению на основе Oblivious RAM [20].

1.6 Сравнение существующих схем шифрования

При выборе SE-схемы приходится находить компромисс между безопасностью, функциональностью запросов и производительностью. Функциональность запросов определяется типами запросов, которые поддерживаются, а производительность характеризуется структурами данных и механизмами индексирования в базе данных. Безопасность поиска по зашифрованным данным означает, что зашифрованная коллекция файлов не раскрывает серверу информацию об оригинальных данных, а также поисковые запросы, посылаемые на сервер, не раскрывают оригинальные ключевые слова.

Самым безопасным среди указанных подходов считается ORAM [20], так как данный подход скрывает от сервера всю возможную информацию вплоть до результатов каждого отдельного запроса поиска. В основе PPE лежит детерминированное шифрование, которое позволяет серверу наблюдать за

повторяемостью документов и запросов, что отрицательно сказывается на безопасность этого подхода. В свою очередь функциональное шифрование основано на рандомизированном шифровании и не позволяет отслеживать закономерности в данных и запросах. Но в силу того, что такой подход использует криптосистему с открытым ключом, есть вероятность, что он может подвергаться словарным атакам со стороны сервера.

Среди описанных моделей PPE схемы показывают логарифмическое время работы алгоритма $O(log(n))$, где n – размер коллекции документов, хранимых на сервере. Данный показатель достигается за счет использования специальных структур для хранения данных (binary search tree). Такие схемы как ORAM, функциональное шифрование и полностью гомоморфное шифрование показывают линейное время поиска $O(n)$ относительно размера коллекции, что связано с необходимостью в процедуре поиска проверять всё содержимое документов. При использовании структур вроде обратного индекса в SSE, при поиске зашифрованных документов по ключевому слову, время работы алгоритма сводится к линейной $O(m)$ относительно m, где m – подмножество коллекции документов, содержащих ключевое слово.

Немаловажным критерием также является практическая применимость той или иной схемы. В работах [20, 21] отмечено, что гомоморфное шифрование и концепция ORAM применимы к небольшим коллекциям документов и являются скорее теоретическими.

Таким образом, среди всех перечисленных моделей поиска по зашифрованным данным, наиболее предпочтительной моделью является SSE, в связи с тем, что многие схемы шифрования являются трудно применимыми на практике ввиду низкого уровня безопасности и высоких требований к вычислительным ресурсам. В следующем разделе будет проведен сравнительный анализ некоторых схем шифрования, основанных на этой модели.

2 Особенности существующих SSE схем шифрования

Для сравнения существующих SSE схем шифрованием воспользуемся нотациями безопасности, которые были формализованы в работах [22, 23]: CPA, CKA1, CKA2.

CPA (Chosen-plaintext attack) – определяет надежность схемы при атаке на основе подобранного открытого текста. Предполагается, что злоумышленник имеет доступ к шифрующей функции, а также имеет возможность отправлять оригинальные входные данные и получать зашифрованные. Полученные зашифрованные сообщения не должны выдавать информацию об оригинальном сообщении.

CKA1 (non-adaptive chosen-keyword attack) – неадаптивная атака на основе выбранных ключевых слов. CKA1 является нотацией, которая адресована непосредственно к SSE [10]. Данная модель атаки предполагает, что сервер не использует историю пользовательских запросов и соответствующих им результатов. Зашифрованная поисковая структура и зашифрованные документы не должны предоставлять серверу информацию об оригинальной коллекции, кроме числа файлов и их размера. В то же время, поисковый запрос и зашифрованная поисковая структура должны выдавать не более чем список идентификаторов соответствующих файлов. Рассматриваемый уровень безопасности подходит для случаев, когда в поисковых запросах нет некоторой повторяемости и, следовательно, анализ запросов и результатов не даст серверу возможности узнать о хранимых документах.

CKA2 (adaptive chosen-keyword attack) - более сильная нотация по сравнению с CKA1. Предполагается, что у сервера есть возможность наблюдать и анализировать пользовательскую историю запросов и выдаваемых результатов.

На сегодняшний день SSE схемы можно разделить на статические и динамические модели. Статическая модель подразумевает о необходимости полностью переписывать зашифрованную структуру, к примеру, индекс, при добавлении и удалении документов коллекции. Динамическая SSE схема шифрования в этом смысле является более применимой в практических условиях, когда необходимо непрерывная обработка данных - добавление и удаление файлов из зашифрованной базы данных.

В таблице 1 представлен результат сравнительного анализа среди некоторых известных SSE схем шифрования, которые позволяют осуществлять поиск в зашифрованной базе данных по ключевым словам. Здесь использованы обозначения: f – коллекция файлов, W – множество уникальных слов, содержащихся в коллекции, $|f|$ – количество бит в файле, $\#f$ – число файлов в коллекции, $\#f_w$ – число файлов, содержащих слово w, $\#W$ – количество уникальных слов, m_f – максимальное число файлов, в которых появляется ключевое слово.

Таблица 1 – Сравнение SSE схем

Схема	Динамичность или подвижность	Безопасность	Время поиска	Размер индекса
SWP00 [7]	статичная	CPA	$O\ \|f\|$	N/A
Z-IDX [22]	динамичная	CKA1	$O\ (\#f)$	$O\ (\#f)$
CM05 [24]	статичная	CKA1	$O\ (\#f)$	$O\ (\#f \cdot \#W)$
SSE-1 [13]	статичная	CKA1	$O\ (\#f_w)$	$O\ (\sum_w \#f_w + \#W)$
SSE-2 [13]	статичная	CKA2	$O\ (\#f)$	$O\ (\#f \cdot \#W)$
CK10 [26]	статичная	CKA2	$O\ (\#f_w)$	$O\ (\#W \cdot m_f)$
SSE-3 [20]	динамичная	CKA2	$O\ (\#f_w)$	$O\ (\sum_w \#f_w + \#W)$

Наиболее строгая нотация безопасности из рассматриваемых схем – CKA2 – соответствует схемам SSE-2, CK10 и SSE-3 (схема Kamara), но при этом только схема SSE-3 поддерживает динамическое обновление зашифрованного индекса. Стоит заметить, что время поиска в схеме SSE-3 линейно относительно числа файлов, содержащих ключевое слово. Таким образом, можно сделать вывод, что из всех представленных схем, работа Kamara является наиболее приемлемой с точки зрения баланса между вычислительной эффективностью, функциональностью и безопасностью.

3 Описание динамической SSE схемы шифрования

Алгоритм шифрования Kamara [20] с возможностью поиска по ключевым словам позволяет клиенту шифровать данные таким образом, чтобы впоследствии он мог генерировать токены поиска для отправки их в качестве запросов на сервер БД. С помощью токена поиска сервер может выполнить поиск по зашифрованным данным и возвратить соответствующие зашифрованные файлы.

В рамках описания алгоритма будем рассматривать данные как последовательность из n файлов $f = (f_1,..., f_n)$, где f_i представляет последовательность слов $(w_1, ..., w_m)$ из множества слов W. Предполагается, что у каждого файла есть уникальный идентификатор id (f_i), а также что данные динамические, то есть в любой момент времени может быть добавлен или удален файл. В рамках описания алгоритма данные могут иметь любой формат данных, который позволяет отобразить каждый документ в файл с ключевыми словами из множества W.

Обозначим:

f_w - множество файлов из f, которые содержат слово w,

c_w - множество зашифрованных файлов f_w.

Операции добавления и удаления файлов выполняются, также как и поиск по ключевым словам, с использованием токенов. Для добавления файла f, клиент генерирует токен добавления t_a на основе добавляемого файла и секретного ключа и отправляет его вместе с зашифрованным файлом c_f на сервер. Сервер, в свою очередь, обновляет зашифрованный индекс γ и коллекцию зашифрованных документов c. Аналогично происходит удаление - генерируется токен удаления t_d и посылается на сервер. Сервер удаляет файл из коллекции c и обновляет зашифрованный индекс γ.

Схема SSE на основе динамического индекса состоит из 9 алгоритмов $SSE = (Gen, Enc, SrchToken, AddToken, DelToken, Search, Add, Del, Dec)$.

Опишем каждый алгоритм:

1. $Gen\ (k) \rightarrow K$ - генерирует секретный ключ на основе параметра безопасности K.

2. $Enc\ (K, f) \rightarrow \gamma, c$ - генерирует последовательность зашифрованных файлов c и зашифрованного индекса γ на основе секретного ключа K и последовательности файлов f.

3. *SearchToken (K, w)* → t_s - генерирует поисковый токен t_s на основе секретного ключа *K* и ключевого слова *w*.

4. *AddToken (K, f)* → t_a, c_f - генерирует токен добавления t_a и зашифрованный файл c_f на основе секретного ключа *K* и добавляемого файла *f*.

5. *DelToken (K, f)* → t_d - генерирует токен удаления t_d на основе секретного ключа *K* и удаляемого файла *f*.

6. *Search (γ, c, t_s)* → I_w - детерминированный алгоритм, который производит поиск в зашифрованной коллекции *c* документов, содержащих ключевое слово *w*, на основе зашифрованного индекса *γ* и поискового токена t_s. Алгоритм возвращает набор найденных идентификаторов файлов I_w.

7. *Add (γ, c, t_a, c_f)* → *γ, c* - алгоритм для добавления зашифрованного файла c_f в зашифрованную коллекцию документов *c*, с использованием токена t_a. Данный алгоритм обновляет зашифрованный индекс *γ*.

8. *Delete (γ, c, t_d)* → *γ, c* - алгоритм удаления зашифрованного файла *c* из зашифрованной коллекции документов *c* на основе токена удаления t_d. Данный алгоритм обновляет зашифрованный индекс γ.

9. *Dec (K, c)* → *f* - используется, чтобы расшифровывать полученный от сервера зашифрованный файл *c* с помощью секретного ключа *K*.

Основой для функционирования схемы SSE является криптосистема с закрытым ключом, которая состоит из трех алгоритмов: *SKE=(Gen, Enc, Dec),* где *Gen* - генерирует секретный ключ на основе некоторого параметра безопасности, *Enc* - шифрует документ с помощью закрытого ключа, *Dec*- расшифровывает поученное зашифрованное сообщение с помощью закрытого ключа. Выбор SKE является частным вопросом реализации.

Кроме SKE в SSE применяются псевдослучайные функции F, G, P и хеш-функции H_1, H_2. Псевдослучайные функции и хеш-функции используются в схеме в процессе шифрования индекса файловой коллекции.

Секретный ключ в схеме SSE представляет собой совокупность четырех элементов: k_1, k_2, k_3, k_4. Ключ k_4 используется для шифрования оригинальной файловой коллекции клиента, генерации токенов t_a и t_d и расшифровки полученных документов. Ключи k_1, k_2, k_3 используются для шифрования отдельных записей индекса, хранимого на сервере.

В рассматриваемом алгоритме зашифрованный индекс представляет совокупность четырёх структур данных: поисковый массив A_s, поисковая таблица T_s, массив удаления A_d и таблица удаления T_d. Для выполнения поиска по ключевому слову используются поисковый массив и поисковая таблица, а для поддержки операций добавления и удаления файлов используются массив удаления и таблица удаления.

3.1 Построение зашифрованного индекса

Построение зашифрованного индекса состоит из создания и заполнения каждого из его четырех элементов: поисковый массив, поисковая таблица, массив удаления и таблица удаления.

Построение поискового массива A_S:

Для каждого слова $w \in W$, где W – множество уникальных слов, содержащихся в файловой коллекции, создается список узлов $N^w = \{N_1, \dots, N_m\}$. Узлы представляют собой список файлов, содержащих слово w. Каждый узел, в свою очередь, представляет собой пару файл-слово и записывается в ячейку A_s, Каждый узел хранит следующую информацию:

$$N_i = \left(id_i, addr_s(N_{i+1}), addr_d(D_i)\right) XOR\ H_1\ \left(P_{K_3}(w)\right),$$

где id_i - уникальный идентификатор файла, $addr_s(N_{i+1})$ - адрес следующего узла N^w в поисковом массиве A_s, $addr_d(D_i)$ - адрес парного узла в массиве удаления A_d, $P_{K_3}(w)$ - генератор случайных чисел от ключа K_3 и слова w. В случае, если элемент является последним в списке N^w, то $addr_s(N_{i+1}) = 0$.

Построение поисковой таблицы T_s:

Каждый список узлов N^w имеет соответствующую запись в поисковой таблице T_s:

$$T_s[F_{K1}(w)] = addr_s(N_1)\ XOR\ G_{K_2}(w),$$

где $F_{K_1}(w)$ - генератор случайных чисел от ключа K_1 и слова w, $addr_s(N_1)$- адрес первого узла списка N^w в A_s, а $G_{K_2}(w)$– генератор случайных чисел от ключа K_2 и слова w.

Построение массива удаления A_d:

Также как и поисковый массив A_s, массив удаления A_d состоит из элементов, представляющих собой соответствие файл-слово. Отличие заключается в том, что список узлов файл-слово строится для каждого файла. Иными словами, для каждого файла f строится список узлов $D = \{D_1, \dots, D_{\#f}\}$, где $\#f$ – число уникальных неповторяющихся слов в файле f. Каждый узел D_i содержит в себе следующую информацию:

$$D_i = \left(addr_d(D_{i+1}), addr_d(N^*_{-1}), addr_d(N^*_{+1}), addr_s(N), addr_s(N_{-1}), addr_s(N_{+1}), F_{K_1}(w)\ XOR\ H_2\left(P_{K_3}(f)\right)\right)$$

где $addr_d(D_{i+1})$ - адрес следующего узла D в A_d, $addr_d(N^*_{-1})$ - адрес предыдущего узла N в A_d, $addr_d(N^*_{+1})$ - адрес следующего узла D в A_d, $addr_s(N)$ - адрес парного узла в A_s, $addr_s(N_{-1})$, - адрес узла в A_s, который предшествует парному узлу, $addr_s(N_{+1})$ - адрес следующего узла в A_s, следующего за парным узлом, $F_{K_1}(w)$ - генератор случайных чисел от ключа K_1 и слова w, P_{K_3} - генератор случайных чисел от ключа K_3 и файла f.

Построение таблицы удаления T_d:

$$T_d\left[F_{K_1}(f)\right] = addr_d(D_1)\ XOR\ G_{K_2}(f),$$

где $addr_d(D_1)$ - адрес первого узла в массиве удаления A_d, а $F_{K_1}(f)$ и $G_{K_2}(f)$ – генераторы случайных чисел от файла f и ключей K_1 и K_2 соответственно.

Таким образом, для каждого уникального слова поисковая таблица указывает на первый элемент в поисковом массиве, а каждый элемент поискового массива содержит указатель на соответствующий идентификатор файла. То есть, обладая ключами K_1, K_2, K_3 можно узнать все идентификаторы файлов, содержащих ключевое слово. В это же время для каждого идентификатора файла таблица удаления содержит указатель на первый элемент массива удаления. Количество элементов в массиве удаления определяется количеством уникальных слов, содержащихся в файле.

3.2 Алгоритм поиска по ключевому слову

Для поиска по ключевому слову клиент формирует поисковый токен $t_s = \left(F_{K_1}(w), G_{K_2}(w), P_{K_3}(w)\right)$ и отправляет его на сервер. Алгоритм поиска:

1. Сервер разделяет t_s на составные части t_1, t_2, t_3.

2. С помощью t_1 в поисковой таблице T_s сервер находит запись о слове w. Если записи нет, то сервер посылает клиенту пустой массив, что означает отсутствие файлов, содержащих данное ключевое слово.

3. С помощью t_2, сервер получает адрес первого узла поискового массива A_s.

4. Используя t_3, сервер обращается по найденному адресу в A_s и получает идентификатор файла содержащего данное ключевое слово и адрес следующего узла. Таким образом, сервер доходит до конца списка в поисковом массиве для указанного ключевого слова и получает все идентификаторы файлов, содержащих данное ключевое слово.

5. Сервер возвращает массив идентификаторов файлов, содержащих данное ключевое слово.

3.3 Алгоритм добавления файла

Для добавления файла f, который содержит множество уникальных слов $W = (w_1, \dots, w_m)$, в зашифрованную коллекцию данных, клиент формирует токен добавления следующим образом $t_a = \{F_{K_1}(f), G_{K_2}(f), \lambda_1 \dots, \lambda_m\}$, где $F_{K_1}(f)$ и $G_{K_2}(f)$ псевдослучайные функции от ключей K_1, K_2 и уникального идентификатора файла f, а

$$\lambda_i = \left(F_{K_1}(w_i), G_{K_2}(w_i), (id(f), 0, 0)\ XOR\ H_1\left(P_{K_3}(w_i)\right), \left(0,0,0,0,0,0, F_{K_1}(w_i)\right) XOR\ H_2(P_{K_3}(f))\right),$$

где $1 \le i \le m$, m – количество уникальных слов в добавляемом файле, $F_{K_1}(w_i)$, $G_{K_2}(w_i)$ и $P_{K_3}(w_i)$ – генераторы случайных чисел от ключей K_1, K_2, K_3 и слова w_i, $id(f)$ – уникальный идентификатор добавляемого файла, $P_{K_3}(f)$ генератор случайных чисел от ключа и добавляемого файла.

После генерации токена клиент шифрует добавляемый файл f и отправляет его серверу вместе с t_a. Сервер, получив зашифрованный файл и токен добавления t_a, выполняет добавление файла по следующему алгоритму:

1. Сервер разделяет t_a на составные части t_1, t_2, , $\lambda_1 \dots, \lambda_m$.
2. Для каждого $i \in [1, m]$ сервер производит следующие действия:

2.1. Генерирует адреса для элементов из *As* и *Ad*: φ, φ^*;

2.2. С помощью поисковой таблицы T_s и $F_{K_1}(w_i)$ сервер вычисляет адрес φ_{-1} последнего элемента в поисковом массиве A_s, и обновляет в $A_s[\varphi_{-1}]$ указатель

$addr_s(N_{i+1})$ на адрес φ следующего элемента в A_s. Если записи в A_s нет, то сервер добавляет запись в таблицу поиска: $T_s[\lambda_i[1]] = \varphi$;

2.3. Добавляет запись в массив поиска: $A_s[\varphi] = \lambda_i[3]\ XOR\ (0, 0, \varphi^*)$;

2.4. С помощью таблицы удаления T_d и t_1 находит адрес φ^*_{-1} последнего элемента в массиве A_d и обновляет его указатель $addr_d(D_{i+1})$ на φ^*, иначе добавляет в таблицу удаления запись: $T_d[t_1] = \varphi^*$;

2.5. Находит парный адрес $\varphi^*_{N_{-1}}$ из A_d для элемента $A_s[\varphi_{-1}]$ и обновляет указатель $addr_d(N^*_{+1})$ элемента $A_d[\varphi^*_{N_{-1}}]$ на φ^*, а указатель $addr_s(N_{+1})$ на φ;

2.6. Добавляет запись в массив удаления

$$(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \mu) = \lambda_i[4]$$

$$A_d[\varphi^*] = \left(\alpha_1, \alpha_2\ XOR\ \varphi^*_{N_{-1}}, \alpha_3, \alpha_4\ \text{XOR}\ \varphi, \alpha_5\ XOR \varphi_{-1}, \alpha_6, \mu\right).$$

3. Далее сервер добавляет зашифрованный файл *f* в коллекцию файлов.

3.4 Алгоритм удаления файла

Для удаления файла f из коллекции файлов, клиент генерирует токен удаления t_d и отправляет его на сервер.

$$t_d = \left\{F_{K_1}(f), G_{K_2}(f), P_{K_3}(f), id(f)\right\},$$

где $F_{K_1}(f)$, $G_{K_2}(f)$ и $P_{K_3}(f)$ – генераторы случайных чисел на основе уникального идентификатора удаляемого файла *f* и ключей K_1, K_2, K_3.

Алгоритм удаления файла:

1. Сервер разделяет токен удаления t_d на составные части t_1, t_2, t_3, *id(f)*.
2. С помощью *id* файла и t_1 сервер находит адрес a_1 на первый элемент в массиве удаления A_d.
3. Далее, для каждого слова $i \in [1, m]$, содержащегося в этом файле, сервер производит следующие действия:

3.1) Расшифровывает элемент D_i в списке удаления A_d: $(\beta_1, \dots, \beta_6, \mu) = D_i\ XOR\ H_2(t_3)$, где $D_i = A_d[a_i]$;

3.2) Удаляет элемент D_i;

3.3) Удаляет парный узел в поисковом массиве A_s по адресу β_4;

3.4) В элементе $A_s[\beta_5]$ обновляет указатель $addr_s(N_{i+1})$ на адрес β_6:

$$(\alpha_1, \alpha_2, \alpha_3) = A_s[\beta_5]$$

$$A_s[\beta_5] = (\alpha_1, \alpha_2\ XOR\ \beta_4 XOR\ \beta_6, \alpha_3)$$

3.5) В элементе $A_d[\beta_2]$ обновляет указатель $addr_d(N^*_{+1})$ на адрес β_3:

$$(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \mu) = A_d[\beta_2]$$

$$A_d[\beta_2] = (\alpha_1, \alpha_2, \alpha_3\ XOR\ a_i XOR\ \beta_3, \alpha_4, \alpha_5, \alpha_6\ \text{XOR}\ \beta_4\ XOR\ \beta_6, \mu)$$

3.6) В элементе $A_d[\beta_3]$ обновляет указатель $addr_d(N^*_{-1})$ на адрес β_2:

$$(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \mu) = A_d[\beta_3]$$

$$A_d[\beta_3] = A_d[\beta_3]\ XOR\ (\alpha_1, \alpha_2\ \text{XOR}\ a_i\ XOR\ \beta_2, \alpha_3, \alpha_4, \alpha_5\ \text{XOR}\ \beta_4\ XOR\ \beta_5, \alpha_6, \mu)$$

4. Сервер удаляет из своего хранилища зашифрованный файл с идентификатором *id*(*f*).

4 Слепое усеченное индексирование

Алгоритм слепого индексирования - это метод шифрования, позволяющий с помощью индекса выполнять поиск по зашифрованным данным с точным совпадением без их расшифрования.

Идея такого метода пришла Y. Chang и M. Mitzenmacher при исследовании модели безопасности CKA. Модель описывает интуитивное предположение, что злоумышленник не может получить никакой информации из индексов, кроме как результаты предыдущих запросов.

На данный момент такая модель применяется в известной библиотеке для реализации поискового шифрования на уровне поля CipherSweet [27].

Суть алгоритма слепого индексирования заключается в том, что вместе с зашифрованными данными передается индекс, который хранит результат применения криптографически стойкой (PRF) псевдослучайной функции к исходному зашифрованному документу. Шифрование и псевдослучайная функция должны использовать два различных ключа. Недостатком такой базовой модели слепого индексирования является то, что значение индекса будет одним и тем же для двух одинаковых фрагментов данных, и как следствие — потенциальная утечка, потому что сервер на основе анализа зашифрованных данных и их индексов может определить, какие записи в таблице содержат одинаковые значения. Чтобы уменьшить негативные последствия этой утечки, CipherSweet предлагает хранить на сервере усеченное значение индексов. Однако при этом вырастет вероятность ошибки включения в результат запроса неверной записи. В этом случае на приложение возлагается ответственность отфильтровать все неподходящие под данный поисковый запрос записи. При усеченном индексировании рекомендуется использовать индекс размером от 8 до 14 бит.

5 Программная реализация

5.1 Выбор технологий

В качестве сервера для хранения зашифрованных данных была выбрана база данных MongoDB. MongoDB - это нереляционная БД, которая обладает гибкими схемами, хорошей функциональностью и высокой производительностью.

Для взаимодействия с базой данных выбран модуль mongoose, который позволяет определять объекты со строго-типизированной схемой, соответствующей документу MongoDB. Mongoose предоставляет огромный набор функциональных возможностей для создания и работы со схемами. Mongoose позволяет задать пользовательскую функцию проверки данных, а также указать, что поле необходимо заполнить.

В качестве среды разработки выбран Visual Studio Code, который является открытым программным обеспечением.

5.2 Структура программы «SecureDataStorage»

В модуле «SecureDataStorage» реализованы следующие основные функции (Приложение А):

- *connect(url, schema, modelName, key, dbOptions)* - для подключения к модулю, где
    - *url=”mongodb://{user}:{password}”* - параметр для соединения с БД;
    - schema - это json-объект, который определяет структуру

хранимых данных. Схема данных задаётся в следующем виде:

```
{
   field_name: {
      validation: {Object},
      security: boolean,
      search: “entire” || “word”
   },
//здесь аналогичным образом определяются остальные поля данных
}
```

где *field_name* - название поля, *validation* определяет объект, в котором может быть объявлена пользовательская функция проверки данных, *security* - это логический параметр, который определяет как будут храниться данные - в открытом виде

(*security: false*) или в зашифрованном (*security: true*), *search* - параметр, с помощью которого задаётся то, как должны индексироваться данные для заданного поля. Если *search = "entire",* то для этого поля данных будет доступен поиск по точному совпадению, если *search = "word",* то будет возможен поиск по ключевым словам. Если поле search не задано, то поиск по заданному полю будет невозможен. Важно заметить, что для открытых данных данный параметр заполнять не стоит, а для закрытых данных данный параметр может принимать единственное значение.

○ *modelName* - с помощью данного параметра можно задать имя базы данных, под названием которой будут храниться коллекции данных и индексов.

○ *key* - параметр безопасности. Данный параметр используется для генерации ключей шифрования.

○ *dbOptions* - с помощью данного параметра можно определить дополнительные настройки для БД. Формат определения данных настроек совпадает с форматом, описанным в библиотеке mongoose [28].

- *close()* - метод для закрытия соединения;
- *isConnected()* - метод для проверки состояния подключения.

Для обработки и поиска данных в зашифрованной БД реализованы методы:

- *upsert(data)* - изменяет существующий документ при наличии id в json-объекте *data*, иначе добавляет новый документ. Общий принцип работы данного метода изображен на рисунке 1:

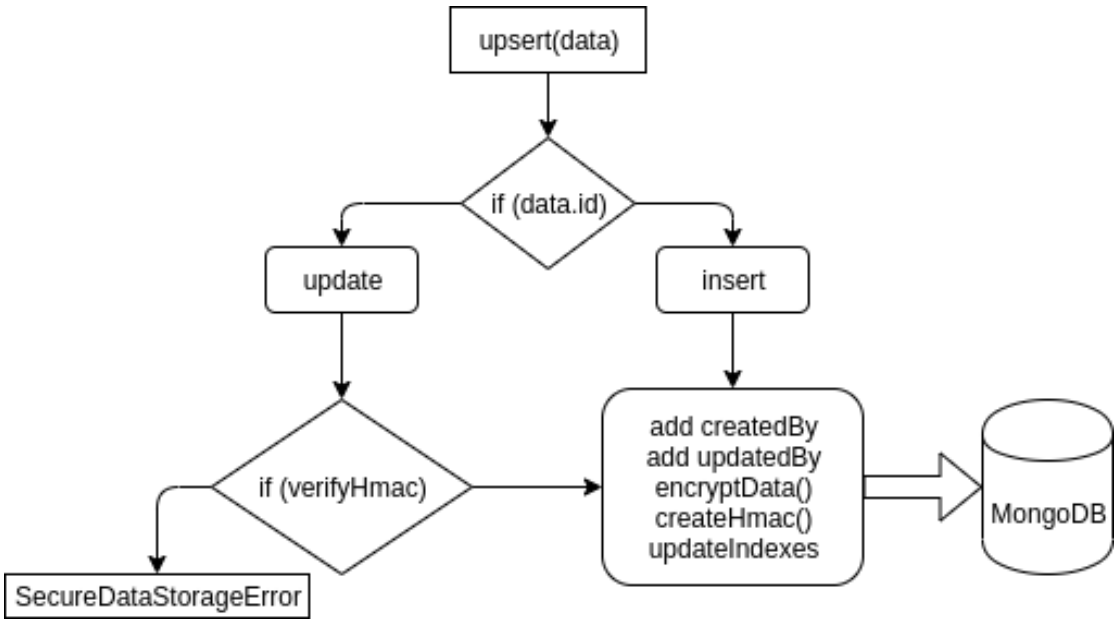


Рисунок 1 – Схема функции upsert

- *get(id)* - возвращает расшифрованные данные по их идентификатору id;
- *find(key, value)* - возвращает массив идентификатор, соответствующих значению *value* по ключу *key*. При поиске по ключевому слову в качестве value задается ключевое слово;
- *delete(id)* - удаляет документ из базы данных по id, обновляя при этом необходимые индексы.

Модуль «SecureDataStorage» состоит из трёх классов:

Engine – класс, в котором реализована основная логика для каждой вышеперечисленной функции модуля (Приложение Б). Класс Engine формирует схему данных на основе переданного параметра *schema* при подключении, добавляя к данным дополнительные поля перед сохранением в БД:

- *signature* - значение функции *hmac*, с помощью которого проверяется целостность данных;
- *column_name_eindex* - значение индекса для поля *column_name* для поиска по точному совпадению;
- *createdBy* - при вызове метода *upsert* можно передать модулю необходимые данные, с помощью которых в последующем можно будет идентифицировать пользователя, который создал данный документ;
- *updatedBy* - при вызове метода *upsert* можно передать модулю необходимые данные, с помощью которых в последующем можно будет идентифицировать пользователя, который изменил уже существующий документ;
- *createdAt* - дата и время создания документа;
- *updatedAt* - дата и время последнего изменения документа.

Encryption – в этом классе реализованы методы, которые выполняют шифрование, расшифрование файлов (Приложение В). Здесь же определены хэш-функции, функции подписи и проверки подлинности. В классе Encryption генерируется ключ, который используется для шифрования и расшифрования документов с помощью симметричного алгоритма шифровании. В этом классе генерируются дополнительные три ключа шифрования, которые используются для индексирования, которое применяется в динамическом SSE алгоритме.

WIndex – класс, предназначенный для создания индексов динамического SSE шифрования (Приложение Г). Он предоставляет функции создания, обновления и удаления индексов, а также логику поиска данных по ключевому слову.

5.3 Используемые криптографические примитивы

Для генерации ключей при подключении к модулю, на основе переданного параметра безопасности, используется функция генерации ключа на основе стандарта PBKDF2.

В качестве симметричного алгоритма шифрования в модуле используется алгоритм AES. Данный выбор был обоснован тем, что алгоритм является стандартом шифрования, принятым Национальным институтом стандартов и технологий (The National Institute of Standards and Technology, NIST). Кроме того, алгоритм долгое время является наиболее популярным среди коммерческих и государственных организаций и показывает приемлемую криптостойкость.

Для проверки целостности используется HMAC - код проверки подлинности сообщений, использующий односторонние хеш-функции.

В качестве хеш-функции используется SHA-256 - криптографический алгоритм хеширования, разработанный National Security Agency — Агентством национальной безопасности США.

Все перечисленные криптографические алгоритмы реализованы в стандартной библиотеке Crypto, которая используется в модуле.

6 Оценка скорости работы модуля и анализ полученных результатов

В данном разделе приводится оценка быстродействия модуля.

На скорость работы модуля влияют следующие факторы:

1. состояние сети с БД;
2. настройки модуля;
3. количество слов в документе;
4. количество уникальных слов (энтропия).

При тестировании модуля на скорость работы проведено два эксперимента:

1. оценка скорости при разном количестве уникальных слов в документах;
2. оценка скорости при разной конфигурации модуля и без его использования;

Во всех вышеперечисленных экспериментах для устранения влияния сети база данных и работающий модуль были запущены на одном компьютере.

Все эксперименты проводились на персональном компьютере с процессором Intel Core i5 - 4200M с частотой 2,5 ГГц и оперативной памятью 8 Гб.

6.1 Оценка скорости при разном количестве уникальных слов в документах

При проведении экспериментов на скорость работы модуля при разном количестве уникальных слов в документах использовались настройки, представленные на схеме 1:

Схема 1.

```
1.      schema = {
2.          field1: {
3.              validation: String
4.          },
5.          field2: {
6.              validation: String
7.          },
8.          field3: {
9.              validation: String,
10.             security: true
11.         },
12.         field4: {
13.             validation: String,
14.             security: true,
15.             search: “entire”
```

```
16.         },
17.         field5: {
18.             validation: String,
19.             security: true,
20.             search: “word”
21.         }
22.     }
```

Как видно из схемы, поиск по ключевому слову возможен только по полю *field5*.

Количество уникальных слов в данных экспериментах формировалось следующим способом:

- низкая энтропия - все документы содержали одинаковые слова в разном порядке;
- средняя энтропия - все документы содержали слова, случайно взятые из файла (в файле 600 уникальных слов);
- высокая энтропия - все документы содержали случайно сгенерированные слова.

Эксперименты проводились несколько раз и затем вычислялось среднее время. Эксперименты были проведены для: 5 и 10 слов в документах.

Таблица 2 – Характеристики скорости добавления документа

Количество документов, шт	Низкая энтропия		Средняя энтропия		Высокая энтропия	
	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс
250	212	219	57	158	61	134
500	419	425	123	241	77	226
750	643	671	145	366	119	262
1000	815	823	156	557	132	488
1250	997	1001	196	570	155	543
1500	1157	1162	230	628	167	661
1750	1312	1362	240	830	189	702
2000	1492	1532	321	849	220	844

Таблица 3 – Характеристики скорости удаления документа

Количество документов, шт	Низкая энтропия		Средняя энтропия		Высокая энтропия	
	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс
50-2000	14	14	27	53	48	124

Таблица 4 – Характеристики скорости поиска документа

Количество документов, шт	Низкая энтропия		Средняя энтропия		Высокая энтропия	
	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс	Время поиска для 5 слов, мс	Время поиска для 10 слов, мс
250	169	166	10	12	2	2
500	343	330	20	20	2	4
750	536	518	35	45	2	5
1000	682	657	52	82	3	5
1250	835	834	57	122	3	6
1500	991	982	74	159	3	6
1750	1085	1169	96	182	4	11
2000	1154	1313	159	201	5	18

6.2 Оценка скорости при разной конфигурации модуля и без использования модуля

Разработанный модуль является настраиваемым и его эффективность во многом зависит от настроек. В качестве настроек выступают следующие параметры:

- количество строк в каждом документе;
- количество строк, которые необходимо шифровать;
- вариант поиска для каждой зашифрованной строки (по точному совпадению или по ключевому слову).

Для сравнения скорости работы модуля с разными настройками использовались схемы: в качестве настроек модуля с возможностью поиска по ключевому слову и по точному совпадению использовалась схема 1; в качестве настроек модуля с возможностью поиска по точному совпадению использовалась схема 2:

Схема 2.

```
1.      schema = {
2.          field1: {
3.              validation: String
4.          },
5.          field2: {
6.              validation: String
7.          },
8.          field3: {
9.              validation: String,
10.             security: true
11.         },
12.         field4: {
13.             validation: String,
14.             security: true,
15.             search: “entire”
16.         }
17.     }
```

Чтобы показать разницу в скорости обработки и поиска данных при использовании модуля «SecureDataStorage» и без его использования, проведены измерения скорости модуля mongoose, который используют для взаимодействия с базой данных MongoDB.

При проведении данных экспериментов, использовалась высокая энтропия и количество слов в документе – 5.

Таблица 5 – Время добавления документа

Количество документов, шт	Время поиска по ключевому слову и точному совпадению, мс	Время поиска по точному совпадению, мс	Время поиска без использования модуля «SecureDataStorage», мс
2000	260	≤6	≤1
4000	559		
6000	908		
8000	1793		
10000	1886		
12000	2152		
14000	2893		
16000	3502		
18000	4176		

Таблица 6 – Время удаления документа

Количество документов, шт	Время поиска по ключевому слову и точному совпадению, мс	Время поиска по точному совпадению, мс	Время поиска без использования модуля «SecureDataStorage», мс
2000	41	≤2	≤1
4000	113		
6000	184		
8000	221		
10000	243		
12000	321		
14000	418		
16000	471		
18000	537		

Таблица 7 – Время поиска документа

Количество документов, шт	Время поиска по ключевому слову и точному совпадению, мс	Время поиска по точному совпадению, мс	Время поиска без использования модуля «SecureDataStorage», мс
2000	3	9	3
4000	4	11	4
6000	5	13	6
8000	6	14	7
10000	9	15	9
12000	14	19	10
14000	23	20	11
16000	26	22	13
18000	33	24	14

6.3 Анализ полученных результатов

Испытание разработанного модуля «SecureDataStorage» позволило сформулировать следующие результаты:

1. С ростом энтропии возрастает скорость добавления и поиска по ключевому слову, но при этом скорость удаления уменьшается;
2. Скорость поиска и добавления прямо пропорциональны количеству документов в базе данных. Количество документов не влияет на скорость удаления;
3. С ростом количества слов в документе скорость добавления, удаления и поиска уменьшаются;
4. При использовании модуля с возможностью поиска по ключевому слову скорость добавления и удаления документов снижается при увеличении количества документов в базе данных. В то время как скорость модуля, настроенного только для поиска по точному совпадению, остается постоянной и незначительно ниже скорости добавления и удаления без использования модуля;

5. Скорость поиска при использовании модуля в разных конфигурациях прямо пропорциональна количеству документов в БД, также как и без использования модуля;

6. Скорость поиска при использовании модуля в разных конфигурациях незначительно ниже скорости поиска без его использования.

ЗАКЛЮЧЕНИЕ

В результате работы было проделано следующее:

1. Сформулированы критерии сравнения и проведен сравнительный анализ современных подходов к поиску по зашифрованным данным;
2. На основе проведенного анализа предложен подход к разработке и реализации настраиваемого модуля «SecureDataStorage»;
3. В рамках предложенного подхода реализован настраиваемый модуль «SecureDataStorage» с возможностью поиска:
   - по точному совпадению;
   - по ключевым словам в тексте.
4. Проведена экспериментальная оценка производительности реализованного модуля.

Реализованный модуль «SecureDataStorage» был интегрирован в web-приложение, которое работает в тестовом окружении, для организации защищенного хранения персональных данных. Результаты испытания показали возможность его практического использования.

В рамках развития предложенного в работе подхода можно выделить следующие направления:

- Расширение поддержки логических запросов;
- Использование других симметричных алгоритмов шифрования и баз данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. InfoWatch [Электронный ресурс] // URL: https://www.infowatch.ru/resources/analytics/digest/15281 (дата обращения 30.01.2020).
2. Gartner Says the Future of the Database Market Is the Cloud [Электронный ресурс] // URL : https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the (дата обращения 30.01.2020).
3. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы и исходный код на С / Б. Шнайер – Вильямс, 2002. – 8 с.
4. Молдовян Н. А. Криптография: от примитивов к синтезу алгоритмов / Н. А. Молдовян. — СПб: БХВ-Петербург, 2004. – 448 с.
5. Bosch C. A Survey of Provably Secure Searchable Encryption // Bosch C. Hartel P, Jonker W, Peter A. // ACM Computing Surveys. – 2014. – Vol. 47. – № 2. – 51 p.
6. Фомичев В. М. Методы дискретной математики в криптологии / В. М. Фомичев. – ДИАЛОГ-МИФИ, 2010. – 424 с.
7. Song D., Wagner D., Perrig A. Practical Techniques for Searches on Encrypted Data [Электронный ресурс]: Berkeley, Electrical engineering and computer sciences. URL: https://people.eecs.berkeley.edu/~dawnsong/papers/se.pdf (дата обращения 30.01.2020).
8. Bellare M., Boldyreva A., Oneill A. Deterministic and Efficiently Searchable Encryption [Электронный ресурс]: Cryptology ePrint Archive, 2006. URL: https://eprint.iacr.org/2006/186.pdf (дата обращения 30.01.2020).
9. Deterministic Encryption: Definitional Equivalences and Constructions without [Электронный ресурс]: Random Oracles. URL: https://eprint.iacr.org/2008/267.pdf (дата обращения 30.01.2020).
10. Naveed M., Kamara S., Wright C. Inference Attacks on Property-Preserving Encrypted Databases [Электронный ресурс]: Computer Science. URL: https://cs.brown.edu/~seny/pubs/edb.pdf (дата обращения 30.01.2020).
11. Boneh D., Sahai A., Waters B. Functional Encryption: Definitions and Challenges [Электронный ресурс]: ePrint Archive, 2010. URL: https://eprint.iacr.org/2010/543.pdf (дата обращения 30.01.2020).
12. Boneh D. Identity-Based Encryption from the Weil Pairing / D. Boneh, M. Franklin // SIAM Journal on Computing. – 2003. – № 3. – P. 586-615.

13. Curtmola R. Searchable symmetric encryption: improved definitions and efficient constructions / R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky // CCS '06 Proceedings of the 13th ACM conference on Computer and communications security. – 2006. – P. 79-88.

14. Lewko A. Fully Secure Functional Encryption: Attribute-Based Encryption and (Hierarchical) Inner Product Encryption / Lewko A. et al. // EUROCRYPT 2010: Advances in Cryptology, – 2010. – P. 62-91.

15. Goldreich, O. Software protection and simulation on oblivious RAMs / O. Goldreich, R. Ostrovsky // Journal of the ACM. – 1996. – № 3. – С. 431–473.

16. Goodrich M. T., Mitzenmacher M. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation [Электронный ресурс]: Internet Archive, library. URL: https://archive.org/details/arxiv-1007.1259/mode/2up (дата обращения 30.01.2020).

17. Ronald L. Rivest, Len Adleman, Michael L. Dertouzos. On Data Banks And Privacy Homomorphisms [Электронный ресурс]: Academic Press, 1978. URL: https://people.csail.mit.edu/rivest/pubs/RAD78.pdf (дата обращения 30.01.2020).

18. Бабенко Л. К. и др. Полностью гомоморфное шифрование [Электронный ресурс]: Обзор, ФГУП "ВИМИ" URL: WWW.VIMI.RU (дата обращения 30.01.2020).

19. Gentry C. A fully homomorphic encryption scheme [Электронный ресурс]: Stanford University, Applied Cryptography Group, 2009. URL: https://crypto.stanford.edu/craig/craig-thesis.pdf (дата обращения 30.01.2020).

20. Kamara S. Dynamic searchable symmetric encryption / S. Kamara, C. Papamanthou, T. Roeder // CCS '12 Proceedings of the 2012 ACM conference on Computer and communications security. – 2012. – С. 965-976.

21. Chang Y. Privacy preserving keyword searches on remote encrypted data / Y. Chang, M. Mitzenmacher // ACNS'05 Proceedings of the Third international conference on Applied Cryptography and Network Security. – 2005. – С. 442-455.

22. Goh E. Secure Indexes [Электронный ресурс]: Stanford University, Applied Cryptography Group. URL: https://crypto.stanford.edu/~eujin/papers/secureindex/secureindex.pdf (дата обращения 30.01.2020).

23. Curtmola R et al. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions [Электронный ресурс]: Cryptology ePrint Archive URL:

https://eprint.iacr.org/2006/210.pdf (дата обращения 30.01.2020).

24. Chang Y. Mitzenmacher M. Privacy preserving keyword searches on remote encrypted data // Applied Cryptography and Network Security (ACNS), 2005. – P. 442-455.

25. Boneh D. Public Key Encryption with Keyword Search / D. Boneh, G. D. Crescenzo, R. Ostrovsky, G. Persiano // IACR Cryptology ePrint Archive. – 2003, – P. 506-522.

26. Chase M., Kamara S. Structured encryption and controlled disclosure / In Proc. Int. Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT), 2010. – P. 577-594.

27. CipherSweet [Электронный ресурс] // URL: https://github.com/paragonie/ciphersweet (дата обращения 30.01.2020).

28. Mongoose [Электронный ресурс] // URL: http://mongoosejs.com (дата обращения 30.01.2020).

29. Baccam T. Transparent Data Encryption: New Technologies and Best Practices for Database Encryption // SANS Institute. – 2010. – 12 p.

30. Song D., Wagner D., Perri A. Practical Techniques for Searches on Encrypted Data [Электронный ресурс] Berkeley, Electrical engineering and computer sciences URL: https://people.eecs.berkeley.edu/~dawnsong/papers/se.pdf (дата обращения 30.01.2020).

ПРИЛОЖЕНИЕ А. Основные функции «SecureDataStorage»

```
module.exports = {
  connect: async (url, schema, modelName, key, dbOptions, schemaOptions) => {
    if (!key) throw new PersonalDataStorageError("Key must not be undefined!");
    engine = new Engine(key);
    await engine.generateModel(schema, modelName, schemaOptions);
    await connect(url, dbOptions);
    return engine.setIndexSettings(schema, dbOptions);
  },

  close: () => {
    disconnect();
  },

  isConnected: () => {
    return getConnect();
  },

  upsert: async (data, opts) => {
    if (!getConnect()) throw new PersonalDataStorageError("No connection to MongoDb!");
    if (opts && typeof opts !== "object"){
      console.log("\x1b[33m%s\x1b[0m","\n[personal-data-storage] - WARNING - options must be as object. Applied options by default\n");
      opts = undefined;
    }
    deleteUndefinedKey(data);
    if (!data) throw new PersonalDataStorageError("No data for upsert");
    // await validateData(data);
    return data.id ? engine.update(data.id, data, opts) : engine.insert(data, opts); // promise
  },

  get: async (id) => {
    if (!id) throw new PersonalDataStorageError("No id to find");
    if (!getConnect()) throw new PersonalDataStorageError("No connection to MongoDb!");
    if (!ObjectId.isValid(id)) throw new PersonalDataStorageError("id is not ObjectId!", 400);
    return engine.getDataByID(id); //promise
  },

  find: async (key, value) => {
    if (!key) throw new PersonalDataStorageError("No key to find");
    if (!value) throw new PersonalDataStorageError("No value to find");
    return engine.find(key, value); //promise
  },

  delete: async (id) => {
    if (!id) throw new PersonalDataStorageError("No id to delete");
    if (!getConnect()) throw new PersonalDataStorageError("No connection to MongoDb!");
    if (!ObjectId.isValid(id)) throw new PersonalDataStorageError("id is not ObjectId!", 400);
    return engine.deleteById(id); //promise
  }
};
```

```
1. class Engine {
2.   #encryptor;
3.   #wIndex;
4.   #validate_schema;
5.   #open_keys;
6.   #secure_keys;
7.   #model;
8.   #search = {};
9.
10.  constructor(secret){
11.    this.#encryptor = new Encryption(secret);
12.    this.#wIndex = new WIndex(secret);
13.  }
14.
15.  _checkSignature(obj) {
16.    let objVerify = {
17.      _id: obj._id,
18.      createdBy: obj.createdBy,
19.      updatedBy: obj.updatedBy
20.    };
21.    Object.keys(this.#validate_schema).forEach(key => {
22.      if (obj[key]) objVerify[key] = obj[key]
23.    });
24.
25.    if (!this.#encryptor.verifySignature(objVerify, obj.signature))
26.      throw new PersonalDataStorageError("Verify signature failed!");
27.  }
28.
29.  _encryptSecureFields(keys, data, initIv){
30.    let object = {};
31.    keys.forEach(key => {
32.      const iv = parseInt((initIv).toString().slice(0, 16), 16) +
   this.#secure_keys.indexOf(key);
33.      object[key] = data[key] ? this.#encryptor.encryptData(data[key], String(iv)) :
   undefined;
34.    });
35.    return object;
36.  }
37.
38.  _decryptSecureFields(data){
39.    let resObj = {};
40.    this.#secure_keys.forEach((key, index) => {
41.      const iv = parseInt((data._id).toString().slice(0, 16), 16) + index;
42.      resObj[key] = data[key] ? this.#encryptor.decryptData(data[key], String(iv)) :
   undefined;
43.    });
44.    return resObj;
45.  }
46.
47.  _keyValueOpenSchema(object){
48.    let openPart = {};
49.    this.#open_keys.forEach(key => openPart[key] = object[key]);
50.    return openPart;
51.  }
52.
53.  _updateOpenData(objectFromDb, newData){
54.    let resObject = {};
55.    this.#open_keys.forEach(key => {
56.      if (objectFromDb[key]) resObject[key] = objectFromDb[key];
```

```
57.   });
58.     this.#open_keys.forEach(key => {
59.       if (Object.keys(newData).includes(key)) resObject[key] = newData[key];
60.     });
61.     return resObject;
62.   }
63.
64.   async _createEIndex(object, data){
65.     let promises = [];
66.     Object.keys(this.#search).forEach(async key => {
67.       promises.push(new Promise (async (resolve) => {
68.         if (this.#search[key] === "entire"){
69.           promises.push(object[`${key}_eindex`] =
   this.#encryptor.createIndexForEntireSearch(data[key]));
70.           resolve();
71.         } else resolve();
72.       }));
73.     });
74.     return await Promise.all(promises);
75.   }
76.
77.   async _updateWIndex(data, id, opts){
78.     const tmp = [];
79.     Object.keys(data).forEach(async key => {
80.       tmp.push(new Promise( async resolve => {
81.         if (this.#search[key] === "word"){
82.           let promises = [];
83.           unique(data[key].split(" ")).forEach(word =>
   promises.push(this.#wIndex.AddAs(word, id, key, opts)));
84.           const arrAsId = await Promise.all(promises);
85.
86.           let objAsId;
87.           arrAsId.map(obj => objAsId = {...objAsId, ...obj});
88.
89.           await this.#wIndex.AddAd(id, objAsId, key, opts);
90.           resolve();
91.         } else resolve();
92.       }));
93.     });
94.     await Promise.all(tmp);
95.   }
96.
97.   async _deleteWIndex(fileId){
98.     let promises = [];
99.     Object.keys(this.#search).forEach(async key => {
100.            promises.push(new Promise(async resolve => {
101.              if (this.#search[key] === "word"){
102.                await this.#wIndex.Delete(fileId, key);
103.                resolve();
104.              } else resolve();
105.            }))
106.          });
107.          await Promise.all(promises);
108.        }
109.
110.        async setIndexSettings(schema, opts){
111.          Object.keys(schema).forEach(key => {
112.            if (schema[key].security) {
```

```
113.              this.#search[key] = schema[key].search && (schema[key].search ===
   "entire" || schema[key].search === "word") ? schema[key].search : "nosearch";
114.            }
115.          });
116.
117.          let promises = [];
118.          Object.keys(this.#search).forEach(async key => {
119.            promises.push(new Promise(async resolve => {
120.              if (this.#search[key] === "word") await
   this.#wIndex.setWIndexSettings(key, opts);
121.              resolve();
122.            }));
123.          });
124.          await Promise.all(promises);
125.          return "Ok";
126.        }
127.
128.        async generateModel(initSchema, modelName, schemaOptions){
129.          this.#validate_schema = await generateValidationScheme(initSchema);
130.          this.#open_keys = generateOpenKeys(initSchema);
131.          this.#secure_keys = generateSecureKeys(initSchema);
132.
133.          let tmp_schema = {};
134.          this.#open_keys.forEach((item) => tmp_schema[item] =
   this.#validate_schema[item]); //add to schema open data
135.          this.#secure_keys.forEach((item) => tmp_schema[item] = { type: String });
   //add to schema secure data
136.
137.
138.          //add index field for entire search
139.          this.#secure_keys.forEach((item) => {
140.            if (initSchema[item].search === "entire"){
141.              tmp_schema[`${item}_eindex`] = {
142.                type: Buffer,
143.                required: true
144.              };
145.            };
146.          });
147.
148.          const newSchema = new mongoose.Schema({
149.            ...tmp_schema,
150.            signature: { type: Buffer },
151.            createdBy: { type: String },
152.            updatedBy: { type: String }
153.          },{
154.            timestamps: {
155.              createdAt: "createdAt",
156.              updatedAt: "updatedAt"
157.            }, ...schemaOptions });
158.
159.          this.#model = mongoose.model(modelName, newSchema);
160.        }
161.
162.        async insert(data, opts){
163.          if (data){
164.            const id = mongoose.Types.ObjectId();
165.            let object = {
166.              _id: id,
167.              ...this._keyValueOpenSchema(data),
```

```
168.               ...this._encryptSecureFields(this.#secure_keys, data, id),
169.               createdBy: data._author || "PersonalDataStorage",
170.               updatedBy: data._author || "PersonalDataStorage"
171.             };
172.
173.             object.signature = this.#encryptor.createSignature(object);
174.
175.             //add eindex
176.             await this._createEIndex(object, data);
177.             await this._updateWIndex(data, id, opts);
178.
179.             const instance = new this.#model(object);
180.             const resp = await instance.save(opts).catch(err => {
181.               throw new PersonalDataStorageError(err);
182.             });
183.
184.             return resp._id;
185.           }
186.           throw new PersonalDataStorageError("Data is undefined");
187.         }
188.
189.         async update(id, data, opts = {}){
190.           if (id && data){
191.             const [ object ] = await this.#model.find({ _id: id }).exec().catch(err =>
    {
192.               throw new PersonalDataStorageError(err);
193.             });
194.
195.             if (!object) throw new PersonalDataStorageError("Record in
    personalDataStorage does not exists!");
196.             await this._checkSignature(object);
197.
198.             let resObject = {
199.               _id: object._id,
200.               ...this._updateOpenData(object, data),
201.               ...this._encryptSecureFields(matchKeys(this.#secure_keys, data), data,
    object._id),
202.               createdBy: data._author || "PersonalDataStorage",
203.               updatedBy: data._author || "PersonalDataStorage"
204.             };
205.
206.             resObject.signature = this.#encryptor.createSignature(resObject);
207.             opts.useFindAndModify = false;
208.             const resp = await this.#model.findOneAndUpdate({ _id: id }, resObject,
    opts).catch(err =>{
209.               throw new PersonalDataStorageError(err);
210.             });
211.
212.             return resp._id;
213.           }
214.           else throw new PersonalDataStorageError("id || data is undefined");
215.         }
216.
217.         async getDataByID(id){
218.           const [ object ] = await this.#model.find({ _id: id }).exec().catch(err => {
219.             throw new PersonalDataStorageError(err);
220.           });
221.
```

```
222.        if (!object) throw new PersonalDataStorageError("Data in personalDataStorage not
   exists by this id!");
223.          await this._checkSignature(object);
224.
225.          return {
226.            ...this._keyValueOpenSchema(object),
227.            ...this._decryptSecureFields(object),
228.            createdAt: object.createdAt,
229.            updatedAt: object.updatedAt
230.          }
231.        }
232.
233.        async find(key, value){
234.          //search in open data
235.          if (this.#search[key] === undefined) {
236.            const objects = await this.#model.find({ [key]: value }).exec().catch(err
   => {
237.              throw new PersonalDataStorageError(err);
238.            });
239.
240.            let result = [];
241.            objects.forEach(async object => {
242.              await this._checkSignature(object);
243.
244.              result.push({
245.                ...this._keyValueOpenSchema(object),
246.                ...this._decryptSecureFields(object),
247.                createdAt: object.createdAt,
248.                updatedAt: object.updatedAt
249.              });
250.            });
251.
252.            return result;
253.          }
254.          else if (this.#search[key] === "entire"){
255.            let obj = {};
256.            obj[`${key}_eindex`] = await
   this.#encryptor.createIndexForEntireSearch(value)
257.
258.            const objects = await this.#model.find(obj).exec().catch(err => {
259.              throw new PersonalDataStorageError(err);
260.            });
261.
262.            let result = [];
263.            objects.forEach(async object => {
264.              await this._checkSignature(object);
265.
266.              result.push({
267.                id: object.id,
268.                ...this._keyValueOpenSchema(object),
269.                ...this._decryptSecureFields(object),
270.                createdAt: object.createdAt,
271.                updatedAt: object.updatedAt
272.              });
273.            });
274.
275.            result.forEach((resObj, index) => {
276.              if (resObj[key] !== value) result.splice(index, 1);
277.            });
```

```
278.
279.             return result;
280.           }
281.           else if (this.#search[key] === "word"){
282.             return await this.#wIndex.Search(value, key);
283.           }
284.         }
285.
286.         async deleteById(id){
287.           const [ object ] = await this.#model.find({ _id: id }).exec().catch(err => {
288.             throw new PersonalDataStorageError(err);
289.           });
290.
291.           if (!object) return {};
292.           await this._checkSignature(object);
293.           await this._deleteWIndex(id);
294.
295.           return this.#model.deleteOne({ _id: id });
296.         }
297.       }
```

```
1. class Encryption {
2.   #algorithm = "aes-256-cbc";
3.   #iterations = 4096;
4.   #hash = "sha256";
5.   #key;
6.
7.   constructor(secret) {
8.     this.#key = crypto.pbkdf2Sync(secret, secret, 4096, 32, "sha256");
9.   }
10.
11.  _getIv(iv){
12.    if (!iv) return Buffer.alloc(16, 0);
13.    else if (iv.length < 16) return Buffer.from(iv.concat("0",repeat(16 - iv.length)));
14.    else return Buffer.from(iv.slice(0, 16));
15.  }
16.
17.  encryptData(plaintext, iv){
18.    try {
19.      const cipher = crypto.createCipheriv(this.#algorithm, this.#key, this._getIv(iv));
20.      return cipher.update(JSON.stringify(plaintext), "utf8", "hex") +
   cipher.final("hex");
21.    } catch (err) {
22.      throw new PersonalDataStorageError(err);
23.    }
24.  }
25.
26.  decryptData(ciphertext, iv){
27.    try {
28.      const decipher = crypto.createDecipheriv(this.#algorithm, this.#key,
   this._getIv(iv));
29.      return JSON.parse(decipher.update(ciphertext, "hex", "utf8") +
   decipher.final("utf8"));
30.    } catch (err) {
31.      throw new PersonalDataStorageError(err);
32.    }
33.  }
34.
35.  createSignature(data){
36.    try {
37.      const hmac = crypto.createHmac(this.#hash, this.#key);
38.      hmac.write(JSON.stringify(sortObject(data)));
39.      hmac.end();
40.      return hmac.read();
41.    } catch (err) {
42.      throw new PersonalDataStorageError(err);
43.    }
44.  }
45.
46.  verifySignature(data, oldSign){
47.    try {
48.      const newSign = this.createSignature(data, this.#key);
49.      return Buffer.compare(newSign, oldSign) === 0 ? true : false;
50.    } catch (err) {
51.      throw new PersonalDataStorageError(err);
52.    }
53.  }
54.
55.  createIndexForEntireSearch(data){
56.    try {
```

```
57. const cipher = crypto.createCipheriv(this.#algorithm, this.#key, this._getIv());
58.       let ciphertext = cipher.update(JSON.stringify(data), "utf8", "hex");
59.       ciphertext += cipher.final("hex");
60.       return ciphertext;
61.     } catch (err) {
62.       throw new PersonalDataStorageError(err);
63.     }
64.   }
65.
66.   generateKey(secret, salt){
67.     return crypto.pbkdf2Sync(secret, salt, this.#iterations, 64, this.#hash);
68.   }
69.
70.   H1(data){
71.     return crypto.createHash(this.#hash).update(data).digest("hex");
72.   }
73.
74.   H2(data){
75.     return crypto.createHash(this.#hash).update(data).digest("hex");
76.   }
77.
78.   G(word, key){
79.     const hmac = crypto.createHmac(this.#hash, key);
80.     hmac.update(word)
81.     return hmac.digest("hex");
82.   }
83.
84.   P(word, key){
85.     const hmac = crypto.createHmac(this.#hash, key);
86.     hmac.update(word)
87.     return hmac.digest("hex");
88.   }
89.
90.   F(word, key){
91.     const hmac = crypto.createHmac(this.#hash, key);
92.     hmac.update(word)
93.     return hmac.digest("hex");
94.   }
95. }
```

```
1. class WIndex {
2.   #indexTModel;
3.   #indexAModel;
4.   #word_index_id = {};
5.   #opts;
6.   #key1;
7.   #key2;
8.   #key3;
9.   #null = "0".repeat(24);
10.  #encryptor;
11.
12.  constructor(secret){
13.    this.#encryptor = new Encryption(secret);
14.    this.#key1 = this.#encryptor.generateKey(secret, secret);
15.    this.#key2 = this.#encryptor.generateKey(this.#key1, secret);
16.    this.#key3 = this.#encryptor.generateKey(this.#key2, secret);
17.  }
18.
19.  async _findAsIndex({ _id }, dataForXor){
20.    const As = await this.#indexAModel.find({ _id });
21.    if (!dataForXor) return As[0].t;
22.    const t = JSON.parse(xor(As[0].t, dataForXor));
23.    return {
24.      _id: As[0]._id,
25.      file: t[0] !== this.#null ? t[0] : 0,
26.      addrDn: t[1] !== this.#null ? t[1] : 0,
27.      next: t[2] !== this.#null ? t[2] : 0
28.    };
29.  }
30.
31.  async _createAsIndex({ _id, file, addrDn, next }, dataForXor){
32.    const instance = new this.#indexAModel({
33.      _id: _id || mongoose.Types.ObjectId(),
34.      t: xor(JSON.stringify([file || this.#null, addrDn || this.#null, next ||
   this.#null]), dataForXor)
35.    });
36.    const As = await instance.save().catch(err => {
37.      throw new PersonalDataStorageError(err);
38.    });
39.    return { _id: As._id };
40.  }
41.
42.  async _updateAsIndex({ _id, file, addrDn, next }, dataForXor){
43.    const As = await this.#indexAModel.findOneAndUpdate({ _id }, {
44.      t: xor(JSON.stringify([file || this.#null, addrDn || this.#null, next ||
   this.#null]), dataForXor)
45.    }).catch(err =>{
46.      throw new PersonalDataStorageError(err);
47.    });
48.    return {
49.      _id: As._id,
50.      addrDn: As.addrDn !== this.#null ? As.addrDn : 0
51.    };
52.  }
53.
54.  async _findAndUpdateAsIndex({ _id, file, addrDn, next }, dataForXor, update){
55.    let As = await this._findAsIndex({ _id }, dataForXor);
56.    if (!dataForXor){
57.      As = await this.#indexAModel.findOneAndUpdate({ _id }, {
58.        t: xor(As, update)
```

```
59. }).catch(err => {
60.         throw new PersonalDataStorageError(err);
61.       });
62.       return;
63.     }
64.     if (file !== undefined) As.file = file;
65.     if (addrDn !== undefined) As.addrDn = addrDn;
66.     if (next !== undefined )As.next = next;
67.     return await this._updateAsIndex(As, dataForXor);
68.   }
69.
70.   async _findAdIndex({ _id }, dataForXor){
71.     const Ad = await this.#indexAModel.find({ _id });
72.     if (!dataForXor) return Ad[0].t;
73.     const t = JSON.parse(xor(Ad[0].t, this.#encryptor.H2(this.#key2)));
74.     return {
75.       _id: Ad[0]._id,
76.       addrD_plus: t[0],
77.       addrNd_minus: t[1],
78.       addrNd_plus: t[2],
79.       addrN: t[3],
80.       addrN_minus: t[4],
81.       addrN_plus: t[5],
82.       word: t[6]
83.     };
84.   }
85.
86.   async _updateAdIndex({ _id, addrD_plus, addrNd_minus, addrNd_plus, addrN, addrN_minus,
    addrN_plus, word }, dataForXor){
87.     As = await this.#indexAModel.findOneAndUpdate({ _id }, {
88.       t: xor(JSON.stringify([ addrD_plus||this.#null, addrNd_minus||this.#null,
    addrNd_plus||this.#null, addrN||this.#null, addrN_minus||this.#null,
    addrN_plus||this.#null, word||this.#null ]), dataForXor)
89.     }).catch(err =>{
90.       throw new PersonalDataStorageError(err);
91.     });
92.     return {
93.       _id: As._id,
94.       addrDn: As.addrDn
95.     };
96.   }
97.
98.   async _findAndUpdateAdIndex({ _id, addrD_plus, addrNd_minus, addrNd_plus, addrN,
    addrN_minus, addrN_plus, word }, dataForXor, update){
99.     const Ad = await this._findAdIndex({ _id }, dataForXor);
100.          if (!dataForXor){
101.            await this.#indexAModel.findOneAndUpdate({ _id }, {
102.              t: xor(Ad, update)
103.            }).catch(err => {
104.              throw new PersonalDataStorageError(err);
105.            });
106.            return;
107.          }
108.
109.          if (addrD_plus !== undefined) Ad.addrD_plus = addrD_plus;
110.          if (addrNd_minus !== undefined) Ad.addrNd_minus = addrNd_minus;
111.          if (addrNd_plus !== undefined )Ad.addrNd_plus = addrNd_plus;
112.          if (addrN !== undefined )Ad.addrN = addrN;
113.          if (addrN_minus !== undefined )Ad.addrN_minus = addrN_minus;
```

```
114.       if (addrN_plus !== undefined )Ad.addrN_plus = addrN_plus;
115.           if (word !== undefined )Ad.word = word;
116.           return await this._updateAdIndex(Ad, dataForXor);
117.         }
118.
119.         async _findOneAndRemoveAdIndex({ _id }, dataForXor){
120.           const Ad = await this.#indexAModel.findOneAndRemove({ _id }).catch(err =>{
121.             throw new PersonalDataStorageError(err);
122.           });
123.           const t = JSON.parse(xor(String(Ad.t), dataForXor));
124.           return {
125.             _id: Ad._id,
126.             addrD_plus: t[0],
127.             addrNd_minus: t[1],
128.             addrNd_plus: t[2],
129.             addrN: t[3],
130.             addrN_minus: t[4],
131.             addrN_plus: t[5],
132.             word: t[6]
133.           };
134.         }
135.
136.         _generateDataForXorAs(word){
137.           return this.#encryptor.H1(this.#encryptor.F(word, this.#key3));
138.         }
139.
140.         _generateDataForXorAd(file){
141.           return this.#encryptor.H2(this.#encryptor.P(file, this.#key3));
142.         }
143.
144.         async setWIndexSettings(key, opts){
145.           const indexASchema = new mongoose.Schema({
146.             t: { type: Buffer }
147.           }, { strict: false });
148.           const indexTSchema = new mongoose.Schema({
149.             _index: mongoose.Mixed,
150.             s: Object,
151.             d: Object
152.            }, { strict: false });
153.           this.#indexAModel = mongoose.model("indexA", indexASchema);
154.           this.#indexTModel = mongoose.model("indexT", indexTSchema);
155.           const indexT = await this.#indexTModel.find({ _index: key }).exec().catch(err
    => {
156.             throw new PersonalDataStorageError(err);
157.           });
158.
159.           if (!indexT.length) {
160.             this.#word_index_id[key] = mongoose.Types.ObjectId();
161.             const instance = new this.#indexTModel({
162.               _id: this.#word_index_id[key],
163.               _index: key,
164.               Ts: {}
165.             });
166.
167.             await instance.save(opts).catch(err => {
168.               throw new PersonalDataStorageError(err);
169.             });
170.           } else {
```

```
171.             if (indexT.length !== 1) throw new PersonalDataStorageError(`Find more than
   1 index for ${key}`);
172.             this.#word_index_id[key] = indexT[0]._id;
173.           }
174.         }
175.
176.         async AddAs(word, file_id, key, opts){
177.           this.#opts = opts;
178.           const dataForXor = this._generateDataForXorAs(word);
179.           let arrAsId = [];
180.           const Dn_id = mongoose.Types.ObjectId();
181.
182.           //check if this word exists in Ts
183.           const Fk1_word = this.#encryptor.F(word, this.#key1);
184.           const record_in_Ts = `s.${Fk1_word}`;
185.           let [ Ts ] = await this.#indexTModel.aggregate([
186.             { $match: { _id: this.#word_index_id[key], [record_in_Ts]: { $exists: true
   } } },
187.             { $project: { [record_in_Ts]: 1 } }
188.           ]);
189.
190.           let As;
191.           //if the word doesn't exist then add an entry to the indexAs collection
192.           if (!Ts){
193.             As = await this._createAsIndex({
194.               file: file_id,
195.               addrDn: Dn_id,
196.               next: 0
197.             }, dataForXor);
198.
199.             //add entry to Ts
200.             Ts = await this.#indexTModel.findOneAndUpdate({ _id:
   this.#word_index_id[key] }, {
201.               $set: { [record_in_Ts]: String(xor(String(As._id),
   this.#encryptor.G(word, this.#key2))) }
202.             }, opts).catch(err =>{
203.               throw new PersonalDataStorageError(err);
204.             });
205.
206.             arrAsId.push(As._id);
207.             return { [word]: {
208.               Ad_id: Dn_id,
209.               arrAs: arrAsId,
210.               As_minus : 0
211.             } };
212.           }
213.           else {
214.             As = await this._findAsIndex({ _id: String(xor(Ts.s[Fk1_word],
   this.#encryptor.G(word, this.#key2))) }, dataForXor);
215.             arrAsId.push(As._id);
216.
217.             //find latest record in As
218.             while (As.next !== 0){
219.               As = await this._findAsIndex({ _id: As.next }, dataForXor);
220.               arrAsId.push(As._id);
221.             }
222.
223.             const id = mongoose.Types.ObjectId();
224.             arrAsId.push(id);
```

```
225.         await this._createAsIndex({
226.               _id: id,
227.               file: file_id,
228.               addrDn: Dn_id,
229.               next: 0
230.             }, dataForXor);
231.
232.             await this._findAndUpdateAsIndex({
233.               _id: As._id,
234.               next: id
235.             }, dataForXor);
236.
237.             return { [word]: {
238.               Ad_id: Dn_id,
239.               arrAs: arrAsId,
240.               As_minus: As.addrDn
241.             }};
242.           }
243.         }
244.
245.         async AddAd(fileId, objAsId, key, opts){
246.           fileId = String(fileId);
247.           const dataForXor = this._generateDataForXorAd(String(fileId));
248.           //create Ad
249.           const arrayAdId = Object.keys(objAsId).map(key => objAsId[key].Ad_id);
250.           const As_minus_arr = Object.keys(objAsId).map(key => objAsId[key].As_minus);
251.           Object.keys(objAsId).map(key => {
252.             objAsId[key] = objAsId[key].arrAs
253.           });
254.
255.           await Promise.all(Object.keys(objAsId).map(async (word, index) => {
256.             if (As_minus_arr[index]){
257.               return this._findAndUpdateAdIndex({
258.                 _id: ObjectId(As_minus_arr[index]),
259.               }, undefined, Buffer.concat([
260.                 new Buffer.alloc(56),
261.                 xor(String(arrayAdId[index]), this.#null),
262.                 new Buffer.alloc(57),
263.                 xor(String(objAsId[word][objAsId[word].length - 1]), this.#null),
264.                 new Buffer.alloc(93)
265.               ]))
266.             }
267.           }));
268.
269.           const res = await Object.keys(objAsId).map((word, index) => {
270.             const obj = {
271.               addrD_plus: Object.keys(objAsId).length - 1 > index ?
   String(arrayAdId[index + 1]) : this.#null,
272.               addrNd_minus: As_minus_arr[index] ? String(As_minus_arr[index]) :
   this.#null,
273.               addrNd_plus: this.#null,
274.               addrN: objAsId[word][objAsId[word].length - 1] ?
   String(objAsId[word][objAsId[word].length - 1]) : this.#null,
275.               addrN_minus: objAsId[word].length > 1 ?
   String(objAsId[word][objAsId[word].length - 2]) : this.#null,
276.               addrN_plus: this.#null,
277.               word: this.#encryptor.F(word, this.#key1)
278.             };
279.             return {
```

```
280.               _id: arrayAdId[index],
281.               t: xor(JSON.stringify([ obj.addrD_plus, obj.addrNd_minus,
   obj.addrNd_plus, obj.addrN, obj.addrN_minus, obj.addrN_plus, obj.word ]), dataForXor)
282.             }
283.           });
284.
285.           await this.#indexAModel.insertMany(res).catch(err => {
286.             throw new PersonalDataStorageError(err);
287.           });
288.
289.           //create Td
290.           const Fk1_file = this.#encryptor.F(fileId, this.#key1);
291.           const record_in_Td = `d.${Fk1_file}`;
292.
293.           await this.#indexTModel.findOneAndUpdate({ _id: this.#word_index_id[key] }, {
294.             $set: { [record_in_Td]: String(xor(String(arrayAdId[0]),
   this.#encryptor.G(fileId, this.#key2))) }
295.           } , opts).catch(err =>{
296.             throw new PersonalDataStorageError(err);
297.           });
298.         }
299.
300.         async Search(word, key, opts){
301.           //find record in Ts
302.           const dataForXor = this._generateDataForXorAs(word);
303.           const Fk1_word = this.#encryptor.F(word, this.#key1);
304.           const record_in_Ts = `s.${Fk1_word}`;
305.           let [ Ts ] = await this.#indexTModel.aggregate([
306.             { $match: { _id: this.#word_index_id[key], [record_in_Ts]: { $exists: true
   } } },
307.             { $project: { [record_in_Ts]: 1 } }
308.           ]);
309.
310.           if(!Ts) return null;
311.           let arrId = []
312.
313.           //if there is, then find the first element in As
314.           let As = await this._findAsIndex({ _id: String(xor(Ts.s[Fk1_word],
   this.#encryptor.G(word, this.#key2))) }, dataForXor);
315.           arrId.push(As.file);
316.
317.           //find all elements yet (next! == 0)
318.           while(As.next !== 0){
319.             As = await this._findAsIndex({ _id: As.next }, dataForXor);
320.             arrId.push(As.file);
321.           }
322.           return arrId;
323.         }
324.
325.         async Delete(fileId, key){
326.           fileId = String(fileId);
327.           const dataForXor = this._generateDataForXorAd(fileId);
328.
329.           //calculate hash from id file
330.           const Fk1_file = this.#encryptor.F(fileId, this.#key1);
331.           const record_in_Td = `d.${Fk1_file}`;
332.
333.           const Td = await this.#indexTModel.findOneAndUpdate(
334.             { _id: this.#word_index_id[key] },
```

```
335.        { $unset: { [record_in_Td]: 1 }},
336.             { projection: { [record_in_Td]: 1 }}
337.           ).catch(err =>{
338.             throw new PersonalDataStorageError(err);
339.           });
340.
341.           let Ad_addr = String(xor(String(Td.d[Fk1_file]), this.#encryptor.G(fileId,
   this.#key2)));
342.
343.           while (Ad_addr !== this.#null){
344.             let Ad = await this._findOneAndRemoveAdIndex({ _id: Ad_addr }, dataForXor);
345.
346.             //update Nd_plus and N+1 in index Ad at address Nd_minus on Nd_plus and N+1
347.             if (Ad.addrNd_minus !== this.#null){
348.               await this._findAndUpdateAdIndex({
349.                 _id: Ad.addrNd_minus,
350.                 addrNd_plus: Ad.addrNd_plus,
351.                 addrN_plus: Ad.addrN_plus
352.               }, undefined, Buffer.concat([
353.                 new Buffer.alloc(56),
354.                 xor(Ad.addrNd_plus, String(Ad._id)),
355.                 new Buffer.alloc(57),
356.                 xor(Ad.addrN_plus, Ad.addrN),
357.                 new Buffer.alloc(93)
358.               ]))
359.             }
360.
361.             //updae Nd_minus and N-1 in index Ad at address Nd_plus on Nd_minus and N-1
362.             if (Ad.addrNd_plus !== this.#null){
363.               await this._findAndUpdateAdIndex({
364.                 _id: Ad.addrNd_plus,
365.                 addrNd_minus: Ad.addrNd_minus,
366.                 addrN_minus: Ad.addrN_minus
367.               }, undefined, Buffer.concat([
368.                 new Buffer.alloc(29),
369.                 xor(Ad.addrNd_minus, String(Ad._id)),
370.                 new Buffer.alloc(57),
371.                 xor(Ad.addrN_minus, Ad.addrN),
372.                 new Buffer.alloc(120)
373.               ]))
374.             }
375.
376.             //update next in As at address N_1 to N + 1
377.             if (Ad.addrN_minus !== this.#null){
378.               await this._findAndUpdateAsIndex({
379.                 _id: Ad.addrN_minus,
380.                 next: Ad.addrN_plus
381.               }, undefined, Buffer.concat([ new Buffer.alloc(56), xor(Ad.addrN_plus,
   Ad.addrN), new Buffer.alloc(2) ]));
382.             }
383.
384.             //if this is the first element, that is, N_minus is zero, and N_plus is not
   zero, then fix the pointer in Ts to N_plus
385.             if (Ad.addrN_minus === this.#null && Ad.addrN_plus){
386.               const tmp = `s.${Ad.word}`;
387.               const [ Ts ] = await this.#indexTModel.aggregate([
388.                 { $match: { _id: this.#word_index_id[key], [tmp]: { $exists: true } }
   },
389.                 { $project: { [tmp]: 1 } }
```

```
390.               ]);
391.
392.               await this.#indexTModel.find ({ _id: this.#word_index_id[key]
   }).updateOne({
393.                 $set: {
394.                   [tmp]:  String(xor(xor(Ad.addrN_plus, Ad.addrN),
   String(Ts.s[Ad.word])))
395.                 }
396.               });
397.             }
398.
399.             if (Ad.addrN_minus === this.#null && Ad.addrN_plus === this.#null){
400.               const tmp = `s.${Ad.word}`;
401.               await this.#indexTModel.find({ _id: this.#word_index_id[key]
   }).updateOne({
402.                 $unset: { [tmp]: 1 }
403.               });
404.             }
405.
406.             let promises = []
407.
408.             //delete record in As at address N
409.             promises.push(this.#indexAModel.deleteOne({ _id: Ad.addrN }));
410.             await Promise.all(promises);
411.
412.             //go to the next word in the file to be deleted
413.             Ad_addr = Ad.addrD_plus;
414.           }
415.         }
416.       }
```

АНТИПЛАГИАТ
ТВОРИТЕ СОБСТВЕННЫМ УМОМ

Отчет о проверке на заимствования №1

Автор: Абрамов Сергей sergey.abramov996@gmail.com / ID: 7360367
Проверяющий: Абрамов Сергей (sergey.abramov996@gmail.com / ID: 7360367)

Отчет предоставлен сервисом «Антиплагиат»- http://users.antiplagiat.ru

ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 2
Начало загрузки: 30.01.2020 07:45:02
Длительность загрузки: 00:00:02
Имя исходного файла: ДИПЛОМ 30 ЯНВАРЯ.pdf
Название документа: ДИПЛОМ
Размер текста: 1008 кБ
Символов в тексте: 58653
Слов в тексте: 7235
Число предложений: 394

ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Последний готовый отчет (ред.)
Начало проверки: 30.01.2020 07:45:04
Длительность проверки: 00:00:02
Комментарии: не указано
Модули поиска: Модуль поиска Интернет



ЗАИМСТВОВАНИЯ	САМОЦИТИРОВАНИЯ	ЦИТИРОВАНИЯ	ОРИГИНАЛЬНОСТЬ
5,66%	0%	0%	94,34%

Заимствования — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.
Самоцитирования — доля фрагментов текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа, по отношению к общему объему документа.
Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общеупотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.
Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которым шла проверка, по отношению к общему объему документа.
Заимствования, самоцитирования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.
Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Источник	Ссылка	Актуален на	Модуль поиска
[01]	0%	Шифрование базы данных — Википедия	https://ru.wikipedia.org	07 Мар 2019	Модуль поиска Интернет
[02]	1,3%	Шифрование базы данных	http://ru.wikipedia.org	21 Сен 2017	Модуль поиска Интернет
[03]	0,18%	Шифрование базы данных	http://ru.wikipedia.org	21 Мар 2018	Модуль поиска Интернет

Еще источников: 17
Еще заимствований: 4,19%