МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

На правах рукописи

Русаловский Илья Дмитриевич

РАЗРАБОТКА МЕТОДОВ И СРЕДСТВ РЕАЛИЗАЦИИ АЛГОРИТМОВ ГОМОМОРФНОГО ШИФРОВАНИЯ

Специальность 2.3.6 – Методы и системы защиты информации, информационная безопасность

ДИССЕРТАЦИЯ

на соискание ученой степени кандидата технических наук

Научный руководитель: Бабенко Людмила Климентьевна
проф. каф. БИТ ИКТИБ ЮФУ,
д. т. н., профессор

Таганрог
2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире информационные технологии активно применяются во всех сферах жизни. В результате процесса информатизации вырос объем информации, возросли информационные потоки. В условиях бурного роста информационных технологий как никогда ранее стала актуальна проблема обеспечения информационной безопасности. Одна из основных задач информационной безопасности – это обеспечение конфиденциальности информации.

Актуальность необходимости обеспечения конфиденциальности информации обострилась с появлением и широким распространением облачных технологий. Облачные технологии предоставляют простой и удобный механизм хранения и обработки данных с использованием удаленных систем. Облачные системы могут предоставлять мощности для выполнения вычислений. При этом системы имеют клиент-серверную архитектуру, где сервер представляет собой облачный сервис с большими вычислительными мощностями, а клиент – некоторое программное обеспечение, позволяющее пользователю взаимодействовать с серверной частью. При этом клиентская часть, как правило, имеет низкие требования к вычислительным мощностям и необходимо только качественное и стабильное подключение по некоторому каналу связи, как правило – Интернету. Классические криптографические средства обеспечивают необходимый уровень защиты данных при их передаче от клиента на облачный сервис по незащищенному каналу связи, такому как интернет. Однако после передачи сервис получает неограниченный доступ к данным клиента, так как эти данные необходимо обработать и вернуть результат клиенту. В этом кроется потенциальная уязвимость, так как сервис может быть оказаться недобросовестным, либо может быть подменен или скомпрометирован. Для

решения этой проблемы может быть применено одно из молодых направлений в криптографии – гомоморфная криптография [1-9].

Гомоморфное шифрование – это особый вид шифрования, позволяющий выполнять операции над зашифрованными данными и получать зашифрованный результат, соответствующий результату выполнения операции над открытыми данными. Эта особенность позволяет эффективно применять данный вид шифрования для решения любых задач обработки данных без раскрытия самих данных, что актуально, например, для реализации защищенных облачных вычислений и защищенного поиска информации. Данные в этом случае шифруются на стороне клиента, а передаются и обрабатываются в зашифрованном виде. Ключ расшифрования при этом никуда не передается, следовательно, при соблюдении криптографической стойкости алгоритма выполнить расшифрование может только клиент.

В настоящее время ведутся активные исследования в области гомоморфной криптографии. Большинство работ посвящены разработке новых, более эффективных, гомоморфных криптосистем, а также улучшениям эффективности существующих криптосистем [10-54]. Также есть и работы, в которых рассматриваются вопросы прикладного применения гомоморфной криптографии [55-63], однако эта область все еще недостаточно проработана, не хватает готовых прикладных решений, алгоритмов и протоколов. Ряд работ посвящен вопросам изучения и анализа криптографической стойкости существующих алгоритмов [64-67]. Разработанные на основе гомоморфного шифрования библиотеки и сервисы [68-71] имеют ограниченный функционал – как правило реализуются только криптографические примитивы, в то время как для прикладного использования необходим как можно более полный перечень поддерживаемых гомоморфных арифметических и логических операций. В случае отсутствия некоторых операций разработчику прикладного сервиса будет необходимо разработать и реализовать их самостоятельно, что усложняет процесс разработки, требует более высокой

квалификации разработчика, а также требует больших временных, а следовательно, финансовых вложений.

На основании вышесказанного можно сделать вывод о нехватке решений для прикладного применения гомоморфного шифрования, позволяющих удобно и эффективно применять его в таких сферах, как безопасные облачные вычисления.

Степень разработанности темы. Тематика исследования достаточно молода, первая полностью гомоморфная схема шифрования была предложена только в 2009 году. На данный момент существует несколько полностью гомоморфных схем шифрования, на основе некоторых из них разработаны программные криптографические комплексы:

- Библиотека HElib, разработанная Шаем Хавели и Виктором Шоуп, предоставляет возможность тонкой настройки режимов работы схем гомоморфного шифрования. Поддерживаются криптосистемы BGV и CKKS.
- Библиотека гомоморфного шифрования SEAL разработана исследователями Microsoft Research, поддерживает операции сложения и умножения над целыми и вещественными числами.
- Библиотека HEAAN разработана авторами системы CKKS, предоставляет возможность выполнения гомоморфных приближенных вычислений над вещественными числами.

Однако данные программные комплексы реализуют только основные математические гомоморфные операции в рамках гомоморфного алгоритма и для их прикладного применения требуются дополнительные разработки, чтобы реализовать полный перечень математических операций.

Целью исследования является расширение круга выполняемых гомоморфных операций.

Научная задача состоит в разработке методов и алгоритмов реализации гомоморфных операций, позволяющих эффективно применять гомоморфную криптографию для защищенных облачных вычислений.

Достижение поставленной цели и научной задачи исследования требует решения следующих частных задач:

1. Проанализировать существующие методы и алгоритмы гомоморфного шифрования, а также существующие программные реализации алгоритмов гомоморфного шифрования.

2. Разработать методы и алгоритмы реализации гомоморфных операций.

3. Провести экспериментальные исследования по оценке достоверности и эффективности разработанных методов и алгоритмов.

Объектом исследования являются технологии защиты данных при обработке в облачных сервисах.

Предметом исследования являются методы и алгоритмы гомоморфного шифрования.

Методология и методы исследования. Методы исследования основаны на использовании теоретических основ математической логики, теории вероятностей, теории чисел, основ алгоритмизации, методов программирования, теории информационной безопасности, теории гомоморфного шифрования.

К основным **научным положениям, выносимым на защиту**, следует отнести:

1. Новый метод гомоморфного деления позволяет выполнять гомоморфное деления на основе любого полностью гомоморфного алгоритма над целыми числами.

2. Методы гомоморфного сравнения чисел позволяют сравнить гомоморфно зашифрованные числа и получить гомоморфно зашифрованный результат, соответствующий результату сравнения.

3. Алгоритмы реализации гомоморфных операций сложения, разности, умножения и деления над целыми числами через операции над битами с использованием любого полностью гомоморфного алгоритма шифрования над битами позволяют в рамках одной криптосистемы выполнять

и арифметические, и логические операции, что позволяет выполнить реализацию практически любого прикладного алгоритма обработки данных.

4. Алгоритмы реализации гомоморфных операций сложения, разности, умножения и деления над рациональными числами в формате с плавающей точкой с использованием любого полностью гомоморфного алгоритма шифрования над битами позволяют в рамках одной криптосистемы выполнять и арифметические, и логические операции, а также позволяют повысить точность вычислений, что важно при выполнении операции деления.

5. Алгоритм гомоморфной реализации метода Гаусса позволяет выполнить решение СЛАУ методом Гаусса над гомоморфно зашифрованными данными и получить гомоморфно зашифрованный результат, соответствующий решению системы.

Теоретическая значимость результатов исследования заключается в разработке новых методов и алгоритмов для реализации гомоморфных вычислений при решении задач обработки данных в недоверенных средах.

Научная новизна. В диссертации получены следующие новые научные результаты.

1. Разработан новый метод, позволяющий выполнять гомоморфное деление на базе любого полностью гомоморфного алгоритма над целыми числами.

2. Разработаны новые методы гомоморфного сравнения чисел. Первый метод позволяет выполнить сравнение гомоморфно зашифрованных чисел при их побитном гомоморфном шифровании. Второй метод позволяет выполнить гомоморфное сравнение чисел в гомоморфных схемах шифрования, основанных на модулярной арифметике.

3. Разработаны алгоритмы гомоморфной реализации побитовых целочисленных операций сложения, разности, умножения и деления, которые могут быть выполнены на основе любого полностью гомоморфного алгоритма шифрования над битами. Разработанные алгоритмы позволяют выполнять арифметические и логические операции в рамках одного гомоморфного

алгоритма шифрования, благодаря чему возможно выполнить гомоморфную реализацию практически любого прикладного алгоритма обработки данных.

4. Разработаны алгоритмы гомоморфной реализации побитовых операций сложения, разности, умножения и деления над числами в формате с плавающей точкой, которые могут быть выполнены на основе любого полностью гомоморфного алгоритма шифрования над битами. Разработанные алгоритмы позволяют выполнять арифметические и логические операции в рамках одного гомоморфного алгоритма шифрования, а также повышают точность вычислений по сравнению с алгоритмами над числами в формате с фиксированной точкой.

Практическая значимость работы заключается в расширении возможностей практического применения гомоморфного шифрования для решения прикладных задач. Разработанные методы и алгоритмы выполнения гомоморфной математики могут быть использованы в разработке программных продуктов, которые могут быть использованы для разработки сервисов безопасных облачных вычислений на основе гомоморфной криптографии, а также могут быть внедрены в существующие программные комплексы.

Материалы диссертационной работы были использованы в гранте РФФИ № 20-37-90140/20 на тему: “Разработка методов и средств гомоморфного шифрования для облачных сервисов”.

Степень достоверности полученных результатов подтверждается разработкой действующих методов и алгоритмов и их программной реализацией, экспериментами.

Внедрение результатов работы. Результаты диссертационных исследований, подтвержденные соответствующими актами, используются в:

1. научно-производственной деятельности ООО «Айвиаппс» (г. Таганрог), а именно: выполнена практическая апробация разработанных алгоритмов реализации гомоморфной математики на основе битовых операций;

2. гранте РФФИ № 20-37-90140/20 на тему: “Разработка методов и средств гомоморфного шифрования для облачных сервисов”.

3. учебно-исследовательском процессе на кафедре безопасности информационных технологий имени О. Б. Макаревича ИКТИБ ЮФУ.

Апробация результатов. Основные положения и выводы, полученные в представленной работе, докладывались и обсуждались на всероссийских научных конференциях: : IV Всероссийская научно-техническая конференция «Фундаментальные и прикладные аспекты компьютерных технологий и информационной безопасности» (г. Таганрог, 2017 г.), IV Всероссийская научно-техническая конференция «Фундаментальные и прикладные аспекты компьютерных технологий и информационной безопасности» (г. Таганрог, 2018 г.), Научно-методическая конференция «Современные компьютерные технологии» (г. Таганрог, 2020 г.), 15th International Conference On Security Of Information And Networks, SIN 2022 (Sousse, Tunisia, 2022 г.).

Публикации. Основные положения диссертации опубликованы в 12 научных печатных работах, в том числе: 6 – в ведущих рецензируемых научных журналах, входящих в перечень ВАК РФ, 1 – в научных рецензируемых изданиях, индексируемых в базе Scopus, 5 – в материалах конференций и других изданиях. Получено 1 свидетельство о государственной регистрации программ для ЭВМ.

1. Babenko, L. Homomorphic operations on integers via operations on bits / L. Babenko, I. Rusalovsky // Proceedings of the 2022 15th IEEE International Conference on Security of Information and Networks, SIN 2022. – 2022. – P. 01-04. – DOI 10.1109/SIN56466.2022.9970502.

2. Бабенко, Л. К. Библиотека полностью гомоморфного шифрования целых чисел / Л. К. Бабенко, И. Д. Русаловский // Известия ЮФУ. Технические науки. – 2020. – № 2(212). – С. 218-227. – DOI 10.18522/2311-3103-2020-2-218-227.

3. Бабенко, Л. К. Метод реализации гомоморфного деления / Л. К. Бабенко, И. Д. Русаловский // Известия ЮФУ. Технические науки. – 2020. – № 4(214). – С. 212-221. – DOI 10.18522/2311-3103-2020-4-212-221.

4. Бабенко, Л. К. Масштабирование цифровых изображений с применением гомоморфного шифрования / Л. К. Бабенко, И. Д. Русаловский // Вопросы кибербезопасности. – 2021. – № 3(43). – С. 2-10. – DOI 10.21681/2311-3456-2021-3-2-10.

5. Русаловский, И. Д. Разработка методов гомоморфного деления / И. Д. Русаловский, Л. К. Бабенко, О. Б. Макаревич // Известия ЮФУ. Технические науки. – 2022. – № 4(228). – С. 103-112. – DOI 10.18522/2311-3103-2022-4-103-112.

6. Бабенко, Л. К. Гомоморфная реализация метода Гаусса / Л. К. Бабенко, И. Д. Русаловский // Вопросы кибербезопасности. – 2023. – № 4(56). – С. 33-40. – DOI 10.21681/2311-3456-2023-4-33-40.

7. Бабенко, Л. К. Побитовые гомоморфные операции над числами с плавающей точкой / Л. К. Бабенко, И. Д. Русаловский // Известия ЮФУ. Технические науки. – 2023. – № 4(234). – С. 26-34. – DOI 10.18522/2311-3103-2023-4-26-34.

8. Русаловский, И. Д. Библиотека гомоморфного шифрования Helib / И. Д. Русаловский // III Всероссийская научно-техническая конференция "Фундаментальные и прикладные аспекты компьютерных технологий и информационной безопасности" : материалы Всероссийской научно-технической конференции, Таганрог, 03-09 апреля 2017 г. : [в 2 ч.]. Ч. 1. – Ростов-на-Дону : Издательство Южного федерального университета, 2017. – С. 73-76.

9. Русаловский, И. Д. Проблема гомоморфного деления / И. Д. Русаловский // Студенческая наука для развития информационного общества : сборник материалов VII Всероссийской научно-технической конференции (г. Ставрополь, 26-28 декабря 2017 года) : [в 2 ч.]. Ч. 1. – Ставрополь : СКФУ, 2018. – С. 434-437.

10. Русаловский, И. Д. Гомоморфная реализация алгоритма Гаусса / И. Д. Русаловский // IV Всероссийская научно-техническая конференция "Фундаментальные и прикладные аспекты компьютерных технологий и информационной безопасности" : материалы Всероссийской научно-практической конференции. – Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2018. – С. 364-367.

11. Русаловский, И. Проблема полностью гомоморфной обработки целых чисел / И. Русаловский, Л. Бабенко // Безопасность информации и компьютерных сетей : материалы 12-й Международной научной конференции (SIN 2019), 12-15 сентября 2019 г., Сочи, Россия. – Сочи : РИЦ ФГБОУ ВО "СГУ", 2019. – С. 41-43.

12. Бабенко, Л. К. Гомоморфное шифрование. Теоретические основы. Области применения / Л. К. Бабенко, И. Д. Русаловский // Современные компьютерные технологии : сборник статей Научно-методической конференции, Таганрог, 25-29 февраля 2020 г. – Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2020. – С. 58-65. – DOI 10.18522/mod.comp.tech.2020.1.9.

13. Свидетельство о государственной регистрации программы для ЭВМ № 2020611853 Российская Федерация. Реализация программы полностью гомоморфной обработки целых чисел : № 2020610864 : заявл. 31.01.2020 : опубл. 11.02.2020 / Л. К. Бабенко, И. Д. Русаловский ; заявитель федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет» (Южный федеральный университет). (Приложение А).

Связь работы с научными программами, темами, грантами. Исследования выполнялись при финансовой поддержке РФФИ в рамках научного проекта № 20-37-90140/20 на тему: “Разработка методов и средств гомоморфного шифрования для облачных сервисов”.

Соответствие паспорту специальности. Диссертационная работа посвящена разработке методов и алгоритмов выполнения гомоморфной

математики, что соответствует паспорту специальности 2.3.6 «Методы и системы защиты информации, информационная безопасность», п. 15 «Принципы и решения (технические, математические, организационные и др.) по созданию новых и совершенствованию существующих средств защиты информации и обеспечения информационной безопасности.», п. 19 «Исследования в области безопасности криптографических алгоритмов, криптографических примитивов, криптографических протоколов. Защита инфраструктуры обеспечения применения криптографических методов».

Структура и объем работы. Представленная работа состоит из введения, пяти разделов, заключения, списка используемой литературы и приложений, изложенных на 143 страницах, содержит 14 рисунков, 12 таблиц, список литературы из 91 наименования и 3 приложения.

ГЛАВА 1. АНАЛИЗ ТЕКУЩИХ ТЕНДЕНЦИЙ И ИССЛЕДОВАНИЙ В ОБЛАСТИ ГОМОМОРФНОЙ КРИПТОГРАФИИ

1.1. Теоретические основы

Гомоморфное шифрование – это особая форма шифрования, которая позволяет выполнять некоторые математические операции над зашифрованными данными и получать зашифрованный результат, который будет соответствовать результату операций, выполняемых над открытыми данными. В общем виде гомоморфную криптографию можно представить следующим образом:

Пусть E(m) – некоторая функция шифрования, D(c) – функция расшифрования, обратная функции E, m – открытые данные, c – зашифрованные данные. Функция E называется гомоморфной относительно некоторой операции $\circledast$ над открытыми данными, если существует эффективный алгоритм M, который удовлетворяет условию:

$$m_1 \circledast m_2 = D(M(E(m_1), E(m_2))) \quad (1.1)$$

1.1.1. Классификация гомоморфных схем шифрования

В гомоморфных схемах шифрования над целыми числами обычно рассматривается гомоморфизм относительно сложения и умножения. Схема шифрования называется гомоморфной относительно операции сложения (аддитивный гомоморфизм), если выполняется следующее условие:

$$m_1 + m_2 = D(E(m_1) \oplus E(m_2)) \quad (1.2)$$

Схема шифрования называется гомоморфной относительно операции умножения (мультипликативный гомоморфизм), если выполняется следующее условие:

$$m_1 * m_2 = D(E(m_1) \otimes E(m_2)) \quad (1.3)$$

где $\oplus, \otimes$ – операции сложения и умножения над зашифрованными данными, которые соответствуют операциям сложения и умножения над открытыми данными; D – функция расшифрования; E – функция шифрования.

Также к гомоморфным схемам шифрования выдвигаются требования корректности и компактности. Гомоморфная схема шифрования называется корректной, если расшифрование возможно после любого числа гомоморфных операций. Гомоморфная схема шифрования называется компактной, если размер шифртекста не зависит от числа выполненных над ним гомоморфных операций, а определяется только выбранными параметрами схемы шифрования.

На основе вышеописанных свойств и критериев гомоморфные схемы шифрования подразделяются на частично (англ. «partially»), почти (англ. «somewhat») и полностью (англ. «fully») гомоморфные:

- Частично гомоморфные схемы шифрования проявляют гомоморфизм только относительно одной операции и, в свою очередь, подразделяются на аддитивно и мультипликативно гомоморфные.
- Почти гомоморфные схемы шифрования проявляют гомоморфные свойства относительно обеих операций, однако не отвечают критериям корректности и компактности. Также такие схемы называются уровневыми или гибкими (англ. «leveled»), так как после некоторого числа операций «шум» превышает допустимый порог, и корректная расшифровка становится невозможной.
- Полностью гомоморфные схемы шифрования проявляют гомоморфные свойства относительно обеих операций, а также отвечают критериям корректности и компактности.

1.1.2. Схема шифрования RSA

RSA – ассиметричная схема шифрования с открытым ключом, получивший широкое применение в мире. Данная схема шифрования является частично гомоморфной относительно умножения.

Алгоритм генерации ключей.

1. Выбираются два простых числа p и q. Вычисляется модуль схемы $n = p * q$. Вычисляется значение функции Эйлера от модуля: $\varphi(n) = (p - 1) * (q - 1)$
2. Выбирается целое число $e \in (1 \dots \varphi(n))$, взаимно простое с $\varphi(n)$
3. Вычисляется d, такое что $d * e \equiv 1 \ (mod\ \varphi(n))$
4. Пара (e, n) – открытый ключ шифрования, пара (d, n) – закрытый ключ шифрования

Алгоритм шифрования задается формулой $E(m) = m^e \ mod\ n$, где m – открытый текст.

Алгоритм расшифрования задается формулой $D(\mathrm{c}) = \mathrm{c}^d \ mod\ n$, где c – шифртекст.

Гомоморфизм по умножению:

$$E(m_1) * E(m_2) = m_1^k * m_2^k \ mod\ n = (m_1 * m_2)^k \ mod\ n = E(m_1 * m_2)$$

Продемонстрируем гомоморфную операцию умножения на численном примере. Пусть p = 5, q = 11, n = 55, $\varphi(n) = 40$, e = 3, d = 27, $m_1 = 7$, $m_2 = 6$, $E(m_1) = 7^3 \ mod\ 55 = 13$, $E(m_2) = 6^3 \ mod\ 55 = 51$. Тогда:

$$D\big(E(m_1) * E(m_2)\big) = (13 * 51 \ mod\ 55)^{27} \ mod\ 55 = 3^{27} \ \ mod\ 55 = 42$$

$$= 7 * 6 = m_1 * m_2$$

Следует отметить, что гомоморфная операция умножения в данном случае определена в кольце Z_n, следовательно, если результат операции не превышает модуль системы $m_1 * m_2 \in [0,\ n)$, то результат операции будет эквивалентен результату умножения во множестве R, в ином случае результат будет отличен.

1.1.3. Схема шифрования Пэйе

Схема шифрования Пэйе[1] – это криптосистема с открытым ключом, основанная на сложности решения задачи факторизации составного числа, являющегося произведением двух простых чисел. Данный алгоритм является частично гомоморфным и проявляет аддитивные гомоморфные свойства.

Генерация ключей.

1. Выбираются два больших простых числа p, q, такие что НОД(pq, (p-1)(q-1) = 1
2. Вычисляются n = pq, $\lambda = \text{НОК}(p-1, q-1)$
3. Выбирается случайное целое число g из $Z_{n^2}^*$
4. Вычисляется $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$, где L = (u-1)/u
5. Пара (n, g) – открытый ключ, (λ, μ) – секретный ключ

Для шифрования выбирается случайное число r из $Z_{n^2}^*$ и вычисляется значение $c = g^m r^n \bmod n^2$

Расшифрование выполняется по формуле: $m = L\left(c^\lambda \bmod n^2\right) * \mu \bmod n$

Гомоморфизм по сложению: $E(m_1) * E(m_2) = g^{m_1} r_1^n g^{m_1} r_2^n \bmod n^2 = g^{m_1+m_2}(r_1 r_2)^n \bmod n^2 = E(m_1 + m_2)$

Кроме вышеописанного аддитивного гомоморфизма, криптосистема Пэйе обладает рядом интересных свойств:

- можно гомоморфно прибавить не только другое зашифрованное число, но и открытый текст (константу) m_2, представленный в виде g^{m_2}. В результате получим следующее выражение: $g^{m_1} r_1^n g^{m_2} \bmod n^2 = E(m_1 + m_2)$
- можно гомоморфно умножить открытый текст на константу k, для этого необходимо возвести шифртекст в степень, равную этой константе. При расшифровке мы получим: $D(E(m)^k \bmod n^2) = km \bmod n$

---

[1] Paillier P. Public-key cryptosystem based on composite degree residuosity classes, EUROCRYPT'99, Lecture Notes in Computer Science, 1592, 223–238

Благодаря вышеописанным свойствам криптосистема Пэйе может быть эффективно использована для гомоморфной реализации алгоритмов, в которых нет гомоморфного перемножения шифртекстов. Это позволит получить лучших показателей быстродействия по сравнению с полностью гомоморфными схемами шифрования.

1.1.4. Схема шифрования Джентри

Крейг Джентри считается основоположником направления гомоморфной криптографии. Именно им в 2009 году был предложен первый алгоритм шифрования [54], гомоморфный относительно двух операций, доказав тем самым возможность существования полностью гомоморфного шифрования. Схема шифрования Джентри первоначально имела ограничения на число последовательных операций из-за нарастания уровня «шума», из-за чего в определенный момент корректная расшифровка становилась невозможной.

Хотя криптосистема Джентри является «почти» гомоморфной, автором был предложен метод создания полностью гомоморфной схемы на основе почти гомоморфной посредством введения метода «обновления» шифртекстов – бутстрэппинга (англ. «bootstrapping»). Данный метод выполняет перезашифрование шифртекста на новом ключе без необходимости его предварительной расшифровки, в результате чего получается новый шифртекст со сниженным уровнем «шума».

Первоначальный алгоритм был построен на идеальных решетках, из-за этого он был очень сложен в реализации, а шифрование и операции были очень трудоемкими. На основе оригинальной схемы была разработана схема DGHV над полем целых чисел Z. Схема DGHV проще в реализации, также для нее были разработаны оптимизации, позволяющие уменьшить размерность ключей шифрования на порядок, а вместо операции бутстрэппинга была предложена операция смены модуля.

Схема шифрования над полем Z_2.

Ключом является целое нечетное число из интервала $p \in [2^{n-1}, 2^n)$.

Для шифрования бита $m \in \{0,1\}$ нужно задать шифртекст как целое число, остаток которого по модулю p имеет ту же четность, что и текст. А именно, предположим, $c = pq + 2r + m$, где q, r – это случайные числа из некоторого интервала, такие, что $|2r|$ меньше $|p/2|$ и r является шумом.
Для расшифрования сообщения необходимо вычислить $(c \bmod p) \bmod 2$.

Когда шум r значительно меньше секретного ключа p, данная схема является «почти» гомоморфной в терминах работы Джентри. Она может быть использована для вычисления полиномов низких степеней над зашифрованными сообщениями. Кроме того, при разумном разборе параметров эта схема является безопасной.

Кроме всего прочего, данную симметричную схему достаточно легко преобразовать в ассиметричную. В этом случае открытый ключ представляет собой множество зашифрованных нулевых битов $x_i = q_i * p + 2r$, где *q, r* – это случайные числа из некоторого интервала. Шифрование заключается в сложении бита $m \in \{0,1\}$ с суммой подмножеств x_i, то есть $D = m + x_i$.

Гомоморфизм относительно сложения: $c_1 + c_2 = m_1 + 2r_1 + p_1 q + m_2 + 2r_2 + p_2 q = m_1 + m_2 + 2(r_1 + r_2) + (p_2 + p_1)q = E(m_1 + m_2)$

Гомоморфизм относительно умножения: $c_1 * c_2 = (m_1 + 2r_1 + p_1 q) * (m_2 + 2r_2 + p_2 q) = m_1 m_2 + 2(2r_1 r_2 + r_1 m_2 + r_2 m_1) + p(m_1 q_2 + 2r_1 q_2 + m_2 q_1 + 2r_2 q_1 + p q_1 q_2) = E(m_1 m_2)$

1.1.5. Понятие «шума» в гомоморфном шифровании

Большинство гомоморфных алгоритмов построено на маскировании открытого текста некоторыми данными, которые называются «шумом». По мере выполнения операций над шифртекстами шум растет. Как только значение шума превысит некоторый допустимый порог, который определяется прочими параметрами схемы шифрования, корректное

расшифрование шифртекста становится невозможным. Максимальное количество операций, которые можно выполнить до достижения порога называют глубиной схемы. Как правило рассматривается глубина относительно умножения, так как умножение – более ресурсоемкая операция и сильнее увеличивает уровень шума после выполнения. Поэтому любые гомоморфные криптосистемы с шумом изначально являются почти гомоморфными, а для преобразования их в полностью гомоморфные используются дополнительные операции и алгоритмы, такие как бутстрэппинг и смена модуля шифрования.

Рассмотрим в качестве примера алгоритм Джентри над полем Z_n. Шифртекст имеет вид: $c = m + nr + pq$, где p – шифртекст, n – модуль открытого текста, r, q – некоторые случайные целые числа, m – шифруемый открытый текст. В данном случае шумом будет являться выражение $m + nr$, а его пороговым значением – величина ключа шифрования p, иначе:

$m + nr + pq \bmod p \bmod n = m + nr - p \bmod n = (m - p) \bmod n$

Численный пример. Пусть $n = 10$, $p = 101$, $m_1 = 2$, $c_1 = 2 + 10*3 + 101*2 = 234$, $m_2 = 3$, $c_2 = 3 + 10*4 + 101*3 = 346$. Выполним вычисления:

$c_3 = c_1 + c_2 = 234 + 346 = 580$

$D(c_3) = 580 \bmod 101 \bmod 10 = 5 = 2 + 3$ (верно)

$c_4 = c_3 + c_2 = 580 + 346 = 926$

$D(c_4) = 926 \bmod 101 \bmod 10 = 7 = 2 + 3 + 3$ (неверно), так как

$c_4 = (2 + 3 + 3) + (10*3 + 10*4 + 10*4) + (101*2 + 101*3 + 101*3)$;

$m_4 + nr = 118 > 101$ – шум превысил допустимый порог

1.2. Области применения

Благодаря возможности выполнения операций над зашифрованными данными гомоморфное шифрование может эффективно использоваться в следующих сферах [59-62]:

- безопасные облачные вычисления (в том числе обработка цифровых изображений);
- электронное голосование (выборы);
- защищенный поиск информации;
- алгоритмы машинного обучения.

1.2.1. Безопасные облачные вычисления

Облачные вычисления – это модель обеспечения доступа по сети к некоторым общим вычислительным ресурсам. Выделяется три основных модели предоставления услуг:

- IaaS (англ. Infrastructure as a service) – инфраструктура как услуга. Пользователю предоставляется компьютерная инфраструктура, обычно виртуальные платформы (компьютеры), связанные в сеть, которые он самостоятельно настраивает под собственные цели.
- PaaS (англ. Platform as a service) – платформа как услуга. Пользователю предоставляется компьютерная платформа с установленной операционной системой, а также, возможно, некоторым программным обеспечением.
- SaaS (англ. Software as a service) – программное обеспечение как услуга. В данном случае пользователь получает доступ к некоторому программному обеспечению, развернутом на удаленных серверах. Как правило доступ обеспечивается через Интернет с использованием некоторого клиентского программного обеспечения (браузера, почтового клиента и так далее).

Модель SaaS наиболее подходит для развертывания сервисов обработки и хранения данных с использованием гомоморфной криптографии. Проанализируем, какие преимущества дает использование гомоморфной криптографии в облачных сервисах. В классической схеме клиент-серверного взаимодействия пользователь передает на сервер некоторые зашифрованные

данные, на сервере они расшифровываются, обрабатываются, результат обработки снова зашифровывается и отправляется пользователю (рисунок 1.1).

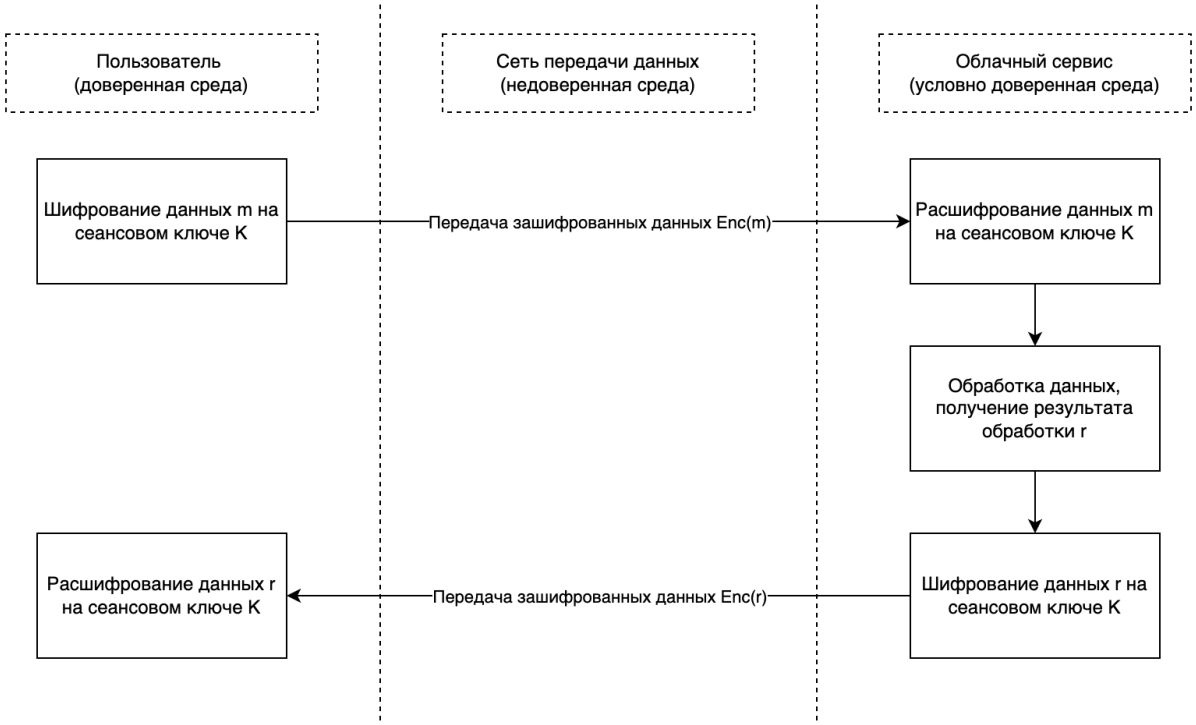

Рисунок 1.1. Схема облачного сервиса с использованием классического шифрования

Для того, чтобы сервер мог расшифровать и зашифровать данные, он должен иметь доступ к секретному ключу шифрования, следовательно ключ либо должен быть передан на сервер, либо сгенерирован с помощью некоторого алгоритма создания общего сеансового ключа. Как следствие, возможны следующие сценарии:

- перехват секретного ключа во время его передачи или генерации;

- подмена сервера; в данном случае злоумышленник может выдать себя за некоторый сервис и получить пользовательские данные;

- недобросовестный поставщик услуг; поставщик услуг может получить доступ к любым данным, так как данные расшифровываются перед обработкой.

Как следствие, уровень защиты информации определяется не только средствами криптографической защиты, но и уровнем доверия к поставщику услуг (сервису). Использование гомоморфной криптографии решает все вышеописанные проблемы. Для этого на серверной стороне реализуется некоторый алгоритм обработки данных с использованием гомоморфной криптографии, на клиентской стороне выполняется гомоморфное шифрование данных для обработки, а далее данные передаются на сервер и обрабатываются там в зашифрованном виде. Пример схемы клиент-серверного взаимодействия с использованием гомоморфной криптографии представлен на рисунке 1.2:

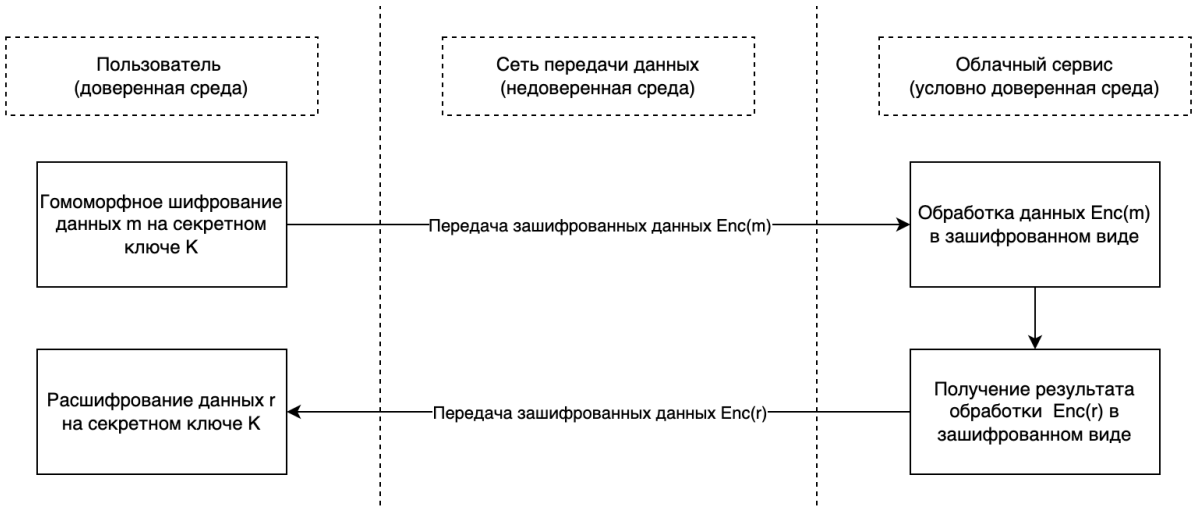

Рисунок 1.2. Схема облачного сервиса с использованием гомоморфного шифрования

В отличие от классической модели клиент-серверного взаимодействия, когда данные передаются в зашифрованном виде только по незащищенным каналам (Интернету), а на сервере расшифровываются и обрабатываются в открытом виде, гомоморфное шифрование позволяет сохранять данные в зашифрованном виде на протяжении всего процесса передачи и обработки. Таким образом, сохранность данных будет зависеть только от стойкости используемого криптографического алгоритма. Гомоморфные алгоритмы

шифрования относятся, как правило, к постквантовым криптосистемам, из чего можно сделать вывод о высоком уровне криптостойкости при условии правильного выбора параметров безопасности. Однако некоторые схемы имеют уязвимости к определенным атакам, а так как направление гомоморфной криптографии еще довольно молодое, требуется выполнение дополнительных исследований криптостойкости гомоморфных алгоритмов шифрования. Но даже с учетом этих недостатков, схема клиент-серверного взаимодействия, в которой уровень защиты данных определяется исключительно криптографической стойкостью используемого алгоритма шифрования, является очень многообещающей.

1.2.3. Обработка цифровых изображений

Обработка цифровых изображений – частная задача облачной обработки данных, когда данными являются цифровые изображения. Гомоморфная обработка цифровых изображений выполняется так же, как и других данных. Изображение попиксельно шифруется на стороне клиента, передается на сервер и обрабатывается в зашифрованном виде, а зашифрованный результат обработки возвращается пользователю. Однако задача обработки изображений имеет свою специфику, обусловленную тем, что изображение описывается большим объемом информации. К примеру, в классическом формате представления цветных цифровых изображений RGB с диапазоном значений каждого цветового канала от 0 до 255, каждый пиксель будет занимать 24 бита (3 байта). А изображение размерностью 100 на 100 пикселей будет занимать 30000 байт (~30 Кб).

При гомоморфном шифровании данные в среднем увеличиваются в размере в 100-1000 раз, в зависимости от используемого алгоритма и выбранных параметров безопасности. Поэтому передача и обработка зашифрованных гомоморфно цифровых изображений малоэффективна. Размер изображения 100 на 100 пикселей, описанного выше, после

шифрования станет ~3-30 Мб. А если первоначальный размер изображения измерялся в мегабайтах, то после шифрования он будет измеряться в гигабайтах. Конечно, современные каналы передачи информации могут обеспечивать скорость передачи, измеряемую в гигабайтах в секунду. Также частично снизить нагрузку на сеть можно с помощью применения гибридного шифрования [14]. Гибридное шифрование позволяет зашифровать данные симметрично, а ключ расшифрования зашифровать гомоморфно и передать на сервер, где сервер зашифрует данные гомоморфно, а после этого выполнит расшифровку симметричного алгоритма, используя гомоморфные операции. В результате на сервере будут получены данные, зашифрованные гомоморфно, а, благодаря использованию симметричного шифрования, объем передаваемых данных снизится. Однако, после обработки цифрового изображения результат будет зашифрован гомоморфно и будет необходимо передать его по сети. Следовательно, гомоморфное шифрование может в теории использоваться для обработки цифровых изображений, однако на практике это будет крайне неэффективно, пока не будут разработаны более производительные гомоморфные алгоритмы шифрования, либо скорость передачи и обработки информации не увеличится на порядок.

1.2.4. Электронное голосование

Благодаря своим свойствам гомоморфная криптография может быть использована для построения схем электронного голосования, к примеру в работе [57] рассматривается возможность построения системы электронного голосования на основе схемы шифрования Пэйе. В общем виде электронное голосование на основе некоторого аддитивного гомоморфного алгоритма шифрования можно описать следующим образом:

Пусть дан некоторый ассиметричный гомоморфный алгоритм шифрования, проявляющий аддитивные гомоморфные свойства; Enc, Dec – функции гомоморфного шифрования и расшифрования соответственно; Ks –

секретный ключ, Ко – открытый ключ, вектор ($a_1 \ldots a_n$) – бюллетень, список кандидатов.

Каждый избиратель получает бюллетень и открытый ключ шифрования.

Голос избирателя представляется в виде вектора предпочтений P ($p_1 \ldots p_n$), где $p_i \in \{0, 1\}$. Избиратель шифрует свой вектор предпочтений с помощью гомоморфного шифрования и отправляет инициатору голосования.

Инициатор голосования суммирует все векторы предпочтений и расшифровывает результат. Индекс наибольшего коэффициента в векторе соответствует индексу победителя голосования.

Описанная система отображает только возможность использования гомоморфной криптографии в системах электронного голосования и для использования в реальных задачах должна быть модифицирована.

1.2.5. Защищенный поиск информации

Также на базе гомоморфной криптографии возможно построение сервиса защищенного (приватного) поиска информации, то есть поиска, при котором поисковый сервер не знает, какие запросы выполняют пользователи. Простейшим примером защищенного поиска будет получение i-го элемента вектора без раскрытия i и самих данных. Пусть дан вектор A $a_1..a_n$, зашифрованный поэлементно с помощью алгоритма полностью гомоморфного шифрования на секретном ключе клиента и хранящийся на сервере. Тогда для получения i-го элемента вектора:

- клиент формирует вектор $b_1 \ldots b_n$, все значения которого равны 0, кроме i-го, шифрует его гомоморфно на своем секретном ключе и передает на сервер;
- на сервере гомоморфно вычисляется скалярное произведение зашифрованных векторов A и B, а результат отправляется клиенту;
- клиент расшифровывает полученный результат на своем секретном ключе.

В реальных же задачах рассматривается работа с зашифрованными базами данных. Алгоритм получения данных из базы данных с использованием гомоморфной криптографии рассмотрен в работе [59]. Авторы рассматривают задачу, когда пользователь хранит свои данные в реляционной базе данных на удаленном недоверенном сервере. Реляционная база данных является набором прямоугольных таблиц (матриц), а каждая таблица соответствует некоторой модели данных. Пусть таблица имеет m атрибутов $a_1 \ldots a_m$ (столбцов) и n записей (строк), E – алгоритм гомоморфного шифрования, D – алгоритм гомоморфного расшифрования, SK – секретный ключ клиента, данные в таблице зашифрованы и имеют вид $db = \{E_{sk}(v_{ij})\}_{i=1,j=1}^{m,n}$. Тогда извлечение данных из базы на основе значений атрибутов выполняется в два этапа:

1. Сначала клиент передает в базу данных зашифрованные значения атрибутов. На сервере эти значения обрабатываются, выполняется операция гомоморфного сравнения запрашиваемых атрибутов с теми, что хранятся в базе данных и формируется зашифрованный результат, который возвращается клиенту. Расшифровав результат, пользователь получает индексы записей в базе данных, которые соответствуют его запросу.
2. Далее пользователь запрашивает данные по индексам из базы данных, воспользовавшись одним из протоколов секретного получения информации [63].

1.2.6. Алгоритмы машинного обучения

Машинное обучение активно используется во многих сферах и показывает отличные результаты. Оно применяется для распознавания изображений, обработки текста, генерации речи, диагностики заболеваний и многих других задач. Машинное обучение, открывает перспективы для новых технологических прорывов в будущем, однако для обучения моделей ему требуются данные. Большая часть данных в существующих моделях

поступают из открытых источников, однако есть сферы, где передача данных третьим лицам в открытом виде может быть невозможна из-за соображений приватности (медицинские данные, финансовые данные и так далее). Данную проблему можешь решить применение гомоморфной криптографии при обработке данных и обучении моделей.

Общем виде обработка пользовательских данных будет выглядеть следующим образом. Пусть дан ассиметричный гомоморфный алгоритм шифрования, Enc, Dec – функции шифрования и дешифрования соответственно, pk, sk – открытый и закрытый ключи шифрования соответственно. Тогда обработка данных с помощью машинного обучения может быть выполнена следующим образом:

- клиент шифрует данные на открытом ключе c = Enc(m, pk) и передает на сервер зашифрованные данные и открытый ключ (c, pk)
- сервер шифрует модель на открытом ключе pk и выполняет гомоморфную обработку данных и возвращает клиенту зашифрованный результат Enc(r, pk)
- клиент расшифровывает результат обработки на своем закрытом ключе r = Dec(Enc(r), sk)

Таким образом гомоморфная криптография позволяет передать все вычисления на сервер, а данные передаются и обрабатываются в зашифрованном виде, что обеспечивает сохранение в секрете конфиденциальных данных. Для обучения моделей гомоморфная криптография также может применяться, однако при большом числе данных для обучения его использование будет малоэффективным.

1.3. Анализ исследований и разработок в области гомоморфной криптографии

Определение «гомоморфное шифрование»[2] было впервые сформулировано в 1978 году Рональдом Ривестом, Леонардом Адлеманом и Майклом Дертузосом - авторами алгоритма шифрования RSA. Ими было выдвинуто предположение о возможности выполнения произвольных операций над зашифрованными данными без и расшифрования. Алгоритм RSA проявляет гомоморфные свойства относительно операции умножения. Первые гомоморфные алгоритмы проявляли гомоморфизм только относительно одной операции, либо обладали серьезными ограничениями. Например, в 1982 году Шафи Гольдвассер и Сильвио Микали предложили систему шифрования над битами, в которой умножение шифртекстов было эквивалентно операции умножения по модулю 2 над исходными битами. Ещё одна криптосистема, гомоморфная относительно операции сложения, была предложена в 1999 году Паскалем Пэйе. В 2005 году Дэн Боне, Ю Чжин Го и Коби Ниссим предложили криптосистему, которая позволяла выполнять неограниченное количество операций сложения и одну операцию умножения. Однако проблема разработки полностью гомоморфной схемы шифрования, которая бы проявляла гомоморфные свойства относительно сложения и умножения, оставалась нерешенной более 30 лет. Пока в 2009 году аспирант Стэнфордского университета и стажёр фирмы «IBM» Крейг Джентри теоретически обосновал принципиальную возможность создания полностью гомоморфной криптосистемы шифрования и предложил одну такую систему. Предложенная схема была выполнена над идеальными решетками, была очень сложна в реализации и требовала высоких вычислительных затрат. Первоначальная схема была почти гомоморфной, так как в результате

[2] R.L. Rivest, L. Adleman, M.L. Dertouzos. On data banks and privacy homomorphisms // Foundations of secure computation. – 1978. – Vol. 32, no. 4. – P. 169–178.

выполнения гомоморфных операций в шифртексте накапливался шум и после достижения некоторого порогового числа корректное расшифрование становилось невозможным. Впоследствии Джентри показал, как преобразовать ее в гибкую схему и с помощью перезашифрования убирать накопившийся шум, чтобы выполнить еще минимум одну операцию. После этого Джентри доказал, что любая гибкая схема шифрования может быть конвертирована в полностью гомоморфную, с помощью рекурсивного встраивания в себя саму [54]. Но схема Джентри была непрактична из-за резкого роста размера шифртекста и затрат на вычисление в зависимости от уровня защиты. Дамьен Шехли и Рон Штейнфелд представили ряд оптимизаций и улучшений [50], и впоследствии Найджел Смарт с Фредериком Веркаутереном [32, 51] и Крейг Джентри с Шайем Халеви [46, 47] представили первые рабочие реализации полностью гомоморфной схемы шифрования Джентри.

В 2010 году Мартин ван Дейк, Крейг Джентри, Шай Халеви и Видон Вайкунтанахан представили вторую полностью гомоморфную систему [52]. Она использовала много принципов криптосистемы Джентри, но не требовала идеальных решёток, а использовала целые числа. Эта схема была концептуально проще схемы Джентри, но обладает схожими параметрами в плане гомоморфизма и эффективности. Многие улучшения и оптимизации схем были представлены в ряде работ Джен-Себастиан Корона, Танкрида Лепойнта, Аврадипа Мандалы, Дэвида Наккхи и Мехди Тибухи [39, 48].

Начиная с 2011 года выделяется второе поколение гомоморфных криптосистем. Разработкой новых, более эффективных, гомоморфных криптосистем занимались Цвик Бракерски, Крейг Джентри, Видон Вайкунтанахан и другие. В результате были разработаны следующие схемы шифрования:

- криптосистема Бракерски-Джентри-Вайкунтанахана (BGV) [45], построенная на технике Бракерски-Вайкунтанахана [32];

- криптосистема Бракерски [42];
- криптосистема на NTRU созданная Лопез-Альтом, Тромером и Вайкунтанаханом (LTV);
- криптосистема Джентри-Сахай-Вотерс (GSW) [38].

Защищенность большинства схем основана на сложности решения проблемы обучения с ошибками (LWE). Только у LVT схемы защита реализована на варианте вычислительной NTRU задачи – трудности поиска кратчайшего вектора решётки. У всех криптосистем второго поколения, в отличие от ранних схем, более медленное нарастание шума в процессе гомоморфных вычислений. В результате дополнительной оптимизации сделанной Крейгом Джентри, Шаем Хавели и Найджелом Смартом, была получена криптосистема с практически оптимальной асимптотической сложностью [43, 44, 49]. Эти оптимизации построены на технике Смарта-Веркаутерена, которая позволяет сжимать набор текстовых переменных в один шифртекст и работать над данными переменными одним потоком. Многие достижения из второго поколения полностью гомоморфных систем так же были использованы в криптосистеме на целых числах [34, 39].

Цвика Бракерски и Видон Вайкунтанахан заметили, что для ряда схем, криптосистема GSW показывает слабый рост уровня шума, и, следовательно, большую эффективность и большую защищенность [36]. Якоб Алперин-Шерифф и Крис Пейкерт позднее описали эффективную технику преобразования шифртекста в гибкий, которая как раз и использует такой тип схем [37]. Но такой тип преобразования не совместим с методами сжатия шифртекста, и таким образом оптимизации Джентри-Сахай-Вотерс не могут быть применены к ней [38].

Все криптосистемы второго поколения до сих пор следуют основам конструкции схемы Джентри, а именно используют почти гомоморфную криптосистему, с большим уровнем роста шума, и далее преобразуют её к

полностью гомоморфной криптосистеме с помощью модификации в гибкую схему.

Для большинства предложенных криптосистем были выполнены реализации в виде криптографических библиотек. Самая первая разработка была выполнена Джентри-Халеви на основе первоначальной схемы Джентри, однако данная реализация была очень медленной и устарела после появления схем второго поколения.

В публичном доступе находятся следующие библиотеки гомоморфного шифрования:

- Библиотека HElib [68], разработанная Шаем Хавели и Виктором Шоуп, которая реализует криптосистему BGV с GHS оптимизацией
- Библиотека TFHE [69], реализует схему, описанную в статье [28]
- Библиотека Microsoft SEAL [70], разработанная группой исследования криптографии и конфиденциальности Microsoft, поддерживает схемы BFV, BGV и CKKS.
- Библиотека FHEW [71], разработанная Лео Дуглас и Даниэль Миккианакио, которая является реализацией комбинации криптосистемы обучения с ошибками Регева и техники создания гибкой схемы Алперин-Шериффа и Пейкерта [37].

Подробный обзор и сравнение существующих библиотек гомоморфной криптографии был выполнен в работах [72-75]. В работе [74] рассматриваются SEAL, HElib, TFHE – популярные библиотеки полностью гомоморфного шифрования, находящиеся в публичном доступе, а также частично гомоморфные алгоритмы шифрования: Paillier, ELGamal, RSA. Сравнение библиотек полностью гомоморфного шифрования по критерию поддерживаемых гомоморфных операций приведено в таблице 1.1.

Таблица 1.1. Сравнение библиотек ПГШ

Операции	SEAL	HElib	TFHE
Сумма, разность	Да	Да	Да
Умножение	Да	Да	Да
Деление	Нет	Нет	Нет
Сравнение	Нет	Нет	Нет
Условные операции	Нет	Нет	Да
Побитовые операции	Да	Да	Да
Матричные операции	Да	Да	Нет
Возведение в степень	Да	Да	Нет
Возведение в квадрат	Да	Да	Да
Отрицание	Да	Да	Нет

Как видно из таблицы, существующие программные комплексы не поддерживают операции деления и сравнения шифртекстов, а данные операции необходимы во многих алгоритмах обработки данных. К примеру, для решения СЛАУ методом Гаусса необходима поддержка операций деления и сравнения шифртекстов. А для простейшей операции нахождения среднего арифметического нужна поддержка операции деления.

Также стоит отметить, что наличие поддержки операций над целыми числами и над битами в приведенной выше таблице не означает, что данные операции поддерживаются одновременно в рамках одной системы шифрования. К примеру, библиотека HElib на основе схемы BGV может работать и с битами, и с целыми числами в кольце в зависимости от того, какая

размерность пространства открытого текста была выбрана в начальных параметрах системы. При этом схема над битами будет поддерживать гомоморфные операции логического “И” и “исключающее ИЛИ”, а схема над целыми числами - сложение и умножение, но одновременно все эти операции не поддерживаются.

1.4. Выводы

Гомоморфная криптография выглядит очень многообещающим направлением, которое позволит повысить конфиденциальность данных при их передаче третьим лицам для обработки и хранения, так как уровень защищенности данных будет определяться исключительно криптографической стойкостью алгоритма шифрования, а не уровнем доверия к поставщику услуг. Однако на данный момент оно имеет существенные недостатки, а также многие аспекты возможности применения его для решения реальных задач еще не проработаны.

Текущие работы в области гомоморфной криптографии в основном направлены на улучшение существующих решений, а также разработку новых алгоритмов полностью гомоморфного шифрования. Программные комплексы, находящиеся в публичном доступе и выполненные в виде библиотек полностью гомоморфного шифрования, реализуют, как правило, только криптографические примитивы и базовые операции и не подходят для решения прикладных задач. В программных комплексах отсутствует поддержка операций деления и сравнения шифртекстов, а также нет возможности выполнять логические операции. Шифрование выполняется либо над целыми числами с возможностью выполнять арифметические операции, либо над битами с возможностью выполнять побитовые операции, однако нет возможности выполнять и побитовые, и арифметические операции в рамках одной схемы шифрования.

На основании проведенного анализа можно сделать вывод о нехватке решений для прикладного применения гомоморфной криптографии. Необходимо решить следующие проблемы:

- отсутствие поддержки гомоморфного деления;
- отсутствие поддержки гомоморфного сравнения;
- отсутствие поддержки условных операций;
- отсутствие возможности выполнять и арифметические, и логические операции над шифртекстами в рамках одной криптосистемы.

ГЛАВА 2. ПРОБЛЕМА ГОМОМОРФНОГО ДЕЛЕНИЯ

Гомоморфное шифрование позволяет обрабатывать данные без предварительной расшифровки, что позволяет эффективно использовать его для защищенных облачных вычислений. В процессе может понадобиться выполнить гомоморфную реализацию некоторого прикладного алгоритма обработки данных, что в свою очередь требует поддержку различных операций над гомоморфно зашифрованными данными, однако на криптографическом уровне полностью гомоморфные алгоритмы шифрования поддерживают ограниченное число операций, как правило, сложение, умножение и смена знака.

Одной из простейших задач, в которой необходима поддержка операции деления, является операция нахождения среднего арифметического. Данная операция используется довольно часто в более сложных задачах, а также может использоваться в чистом виде для решения финансовых и статистических задач.

Другим примером прикладной задачи, в которой необходима поддержка операции деления, является решение систем линейных алгебраических уравнений (СЛАУ). Решение СЛАУ требуется во многих задачах и алгоритмах, например, используется для нахождение некоторых неизвестных коэффициентов на основе ряда экспериментов. При решении подобной задачи для каждого эксперимента строится линейное алгебраическое уравнение, а неизвестные коэффициенты вычисляются на основе решения СЛАУ порядка n, где n – число поставленных экспериментов. Одним из наиболее популярных методов решения СЛАУ является алгоритм Гаусса. Для выполнения его гомоморфной реализации необходима поддержка следующих гомоморфных операций: сложение, разность, умножение, деление, сравнение (для исключения нулевых элементов с главной диагонали). Также операция гомоморфного деления необходима для гомоморфной реализации других

алгоритмов. Однако полностью гомоморфные алгоритмы, как правило, поддерживают только операции сложения и умножения, а в некоторых случаях операцию разности (инверсии). Следовательно, для возможности решения прикладных задач, необходимо разработать метод или алгоритм гомоморфного деления.

Операция деления является обратной к операции умножения. Следовательно, чтобы реализовать гомоморфное деление гомоморфный алгоритм шифрования должен проявлять мультипликативные свойства. Деление может быть выполнено как над целыми, так и над рациональными числами. Конечно, при выполнении деления над целыми числами точность вычислений намного ниже, однако для решения некоторых задач этого может быть достаточно. Рассмотрим несколько алгоритмов гомоморфного шифрования и проанализируем возможность выполнения операции деления на их базе.

2.1. Алгоритмы в кольце вычетов

Одним из вариантов построения гомоморфных алгоритмов является отображение в кольце вычетов. Примером такого алгоритма является RSA. Алгоритм RSA проявляет мультипликативный гомоморфизм, а в кольце вычетов можно найти обратный элемент. Следовательно, возможно реализовать операцию деления как умножение на обратное. Однако, операция деления во множестве целых чисел Z и в кольце вычетов Z_n не эквивалентна.

Рассмотрим кольцо Z_5 в качестве примера. Обратный элемент можно найти, воспользовавшись малой теоремой Ферма (2.1):

$$m^{-1}\ mod\ p \equiv m^{p-2}\ mod\ p \qquad (2.1)$$

Продемонстрируем проблему на численном примере. Пусть $m_1 = 2$, $m_2 = 4$, тогда: $\frac{m_1}{m_2}\ mod\ 5\ \equiv \frac{2}{4}\ mod\ 5\ \equiv 2 *\ 4^{-1}\ mod\ 5\ \equiv 2 * 4\ mod\ 5\ \equiv 3\ mod\ 5$

Как видно из примера, 2 / 4 = 3 в кольце Z_5, в то время как мы ожидали получить 0 или 1, в зависимости от стратегии округления результата. Данное решение может подойти для решения некоторых задач, однако не применимо для реализации, к примеру, алгоритма Гаусса.

2.2. Алгоритмы на основе полиномов

Еще один вариант реализации гомоморфного алгоритма шифрования – соотнесение открытому тексту некоторого полинома. К примеру, Ф. Буртыка предложил алгоритм шифрования на основе матричных полиномов (полиномов, каждый коэффициент которых представлен матрицей) [35]. Также в выпускной квалификационной работе Яковлева [76] предлагается алгоритм шифрования посредством соотнесения целого числа полиному с целочисленными коэффициентами. Между двумя полиномами можно выполнить операцию деления, результатом которой будут частное и остаток от деления. Как было указано в начале главы, для решения некоторых задач может хватить точности деления, при которой остаток полностью отбрасывается. Однако в случае с шифртекстами на основе полиномов возникает следующая проблема. Величина открытого текста никак не связана с порядком полинома, следовательно большему числу может соответствовать полином меньшего порядка и наоборот, из-за этого остаток от деления может быть больше частного. Из-за этого процедура отбрасывания остатка от деления может быть некорректной, однако не отбросить остаток мы тоже не можем, так как шифртекст должен быть представлен одним полиномом.

Рассмотрим численный пример на основе алгоритма Яковлева. Пусть даны целые числа $m_1 = 4$, $m_2 = 1$, $p = 4$, $q = 2$, $x_0 = p / q = 2$ – секретный ключ. Выполним шифрование:

$f_1(x) = 5x + 2$; $f_1(x_0) = 12$

$g_1(x) = 22 * f_1(x) – 22 * 12 + m_1 = 20x + 8 – 48 + 4 = 20x - 36$

$f_2(x) = 3x – 5$; $f_2(x_0) = 1$

$g_2(x) = 22 * f_2(x) – 22 * 1 + m_2 = 12x – 20 – 4 + 1 = 12x - 23$

В результате деления $20x – 36$ на $12x – 23$ получим $g_3(x)) = 1$, остаток $g_4(x) = 8x-13$. После расшифрования получим:

$D(g_3(x)) = g_3(x_0) = 1$

$D(g_4(x)) = g_4(x_0) = 8 * 2 – 13 = 3$

Таким образом получается, что остаток в 3 раза больше, чем делитель, что явно некорректно. Поэтому реализовать операцию деления в гомоморфном алгоритме над полиномами подобным образом не представляется возможным.

2.3. Программная реализация

Данный подход сводится к тому, что деление реализуется за счет введения дополнительного уровня абстракции для представления шифртекста в виде простой дроби [77-83]. За основу берется любой полностью гомоморфный алгоритм шифрования над целыми числами, поддерживающий операции суммы, разности и умножения. Шифруемые данные (целое или рациональное число) представляется в виде простой дроби, делимое и делитель которого шифруются по отдельности с помощью данного полностью гомоморфного алгоритма шифрования над целыми числами. Полученная зашифрованная дробь является шифртекстом. Операции над шифртекстами реализуются как операции над простыми дробями, а при расшифровке делимое и делитель расшифровываются раздельно и делятся друг на друга. Алгоритм можно представить в следующем виде:

Пусть дан некоторый полностью гомоморфный алгоритм шифрования над целыми, для которого определены E(m) – алгоритм шифрования, D(c) – алгоритм расшифрования, обратный к E(m), $\otimes$,$\oplus$ – операторы гомоморфного умножения и сложения над зашифрованными данными соответственно, где m

– открытый текст, c – шифртекст. Тогда схема шифрования целого числа m с поддержкой операции деления может быть построена следующим образом.

Алгоритм шифрования:

1. Представляем открытый текст m в виде простой дроби, где m_1 – делимое, m_2 – делитель.

2. Шифруем делимое и делитель с помощью полностью гомоморфного алгоритма шифрования над целыми числами: $a = E(m_1)$, $b = E(m_2)$

3. Шифртекст в предлагаемой схеме шифрования будет представлен в виде пары зашифрованных гомоморфно чисел: c = (a; b)

Алгоритм расшифрования:

1. Расшифруем гомоморфно зашифрованные делимое и делитель, в виде которых представлен шифртекст: $r_1 = D(a)$; $r_2 = D(b)$

2. Выполняем деление, чтобы получить результат в виде десятичной дроби: $r = r_1 / r_2$

Реализация математических операций:

1. Сложение. $C_1 + C_2 = (a_1 \otimes b_2 \oplus\ a_2 \otimes b_1;\ b_1 \otimes b_2)$

2. Умножение. $C_1 * C_2 = (a_1 \otimes a_2;\ b_1 \otimes b_2)$

3. Деление. ${}^{C_1}/_{C_2} = (a_1 \otimes b_2;\ b_1 \otimes a_2)$

Данный подход прост в реализации, универсален, с его помощью можно добавить операцию деления в любой полностью гомоморфный алгоритм шифрования над целыми. Метод хорошо подходит для программной реализации: для этого используется некоторая абстракция над шифртекстом, расширяющая возможности по выполнению математических операций над ним. Предлагается выделить два уровня представления данных – математический и криптографический, пример построения и взаимодействия которых между собой и пользователем приведен на рисунке 2.1.

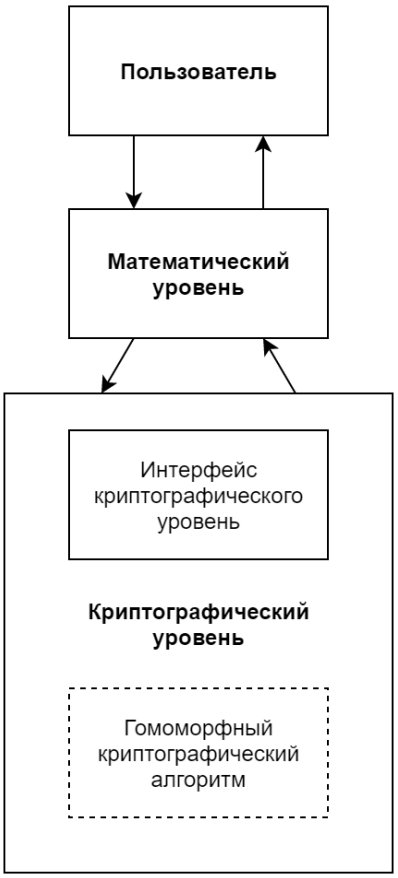

Рисунок 2.1 – Уровни представления данных и их взаимосвязь

Пунктиром выделена часть схемы, которая может иметь различные варианты реализации. Возможность замены реализации криптографического уровня может быть достигнута за счет единства интерфейса. Криптографическое ядро предложенной выше схемы может быть представлено абсолютно любым полностью гомоморфным алгоритмом шифрования. Конечный же пользователь данной схемы будет взаимодействовать только с математическим уровнем, интерфейс которого будет неизменен, какой бы гомоморфный алгоритм шифрования не был бы выбран.

Разработанный метод полезен в том случае, когда реализация гомоморфного деления другим способом требует больших вычислительных мощностей. К минусам можно отнести увеличение размерности шифртекста приблизительно в два раза, так он представлен двумя гомоморфно зашифрованными числами. Также увеличивается сложность выполнения

других операций: умножение усложняется примерно в два раза (из-за необходимости выполнять ее дважды - для делимого и делителя), а сложность операций сложения и разности увеличивается приблизительно в 4 раза (при условии того, что вычислительная сложность операций гомоморфного сложения и умножения эквивалентна) из-за необходимости приведения числа к общему знаменателю.

Рассмотрим подробнее каждый уровень представления данных.

2.3.1. Криптографический уровень

Криптографический уровень представляет собой модуль, основным типом данных которого является зашифрованное гомоморфно число. На данном уровне должны быть реализованы возможности по созданию новых шифртекстов на базе открытых текстов и ключей шифрования (операция шифрования данных), получению открытых данных из шифртекста на основании ключа расшифрования (операция расшифрования данных), а также основные гомоморфные математические операции над шифртекстами – сложение, разность и умножение (рисунок 2.2).

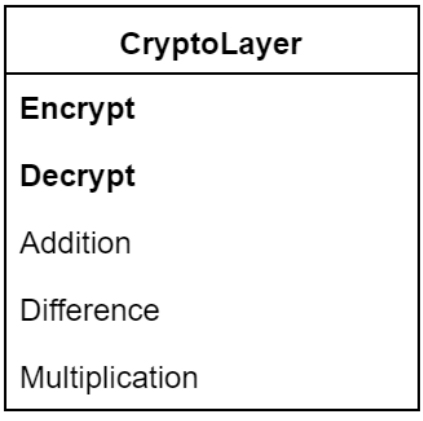

Рисунок 2.2 – Интерфейс класса криптографического уровня

Подобный функционал позволит реализовать любой полностью гомоморфный алгоритм, работающий с целыми числами. Криптографический уровень принимает запросы от математического, гомоморфно обрабатывает

данные и возвращает результат обратно (на математический уровень схемы). Пользователь не имеет доступа к криптографическому уровню напрямую, а взаимодействует только с математическим уровнем.

2.3.2. Математический уровень

Математический уровень является надстройкой над криптографическим. Данный уровень необходим, чтобы обеспечить поддержку операции деления. Для этого на данном уровне данные будут содержаться в виде простой дроби, числителем и знаменателем которой являются экземпляры объектов криптографического уровня. Также на математическом уровне реализуются все арифметические операции, как операции над простыми дробями, – сложение, разность, умножение и деление. Пример интерфейса класса, реализующего математический уровень, приведен на рисунке 2.3.

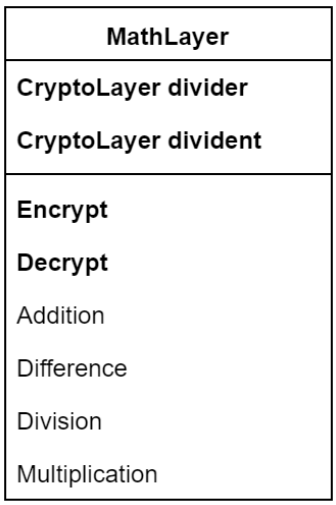

Рисунок 2.3 - Интерфейс класса математического уровня

Математический уровень является прослойкой между пользователем и криптографическим ядром. Все действия на математическом уровне выполняются как операции над простыми дробями, которые были рассмотрены выше, числитель и знаменатель которых поддерживают

операции сложения, разности и умножения так, как представлены объектами криптографического уровня.

2.4. Метод гомоморфного деления на основе битовых операций

Гомоморфное деление возможно реализовать через битовые операции на основе некоторого полностью гомоморфного алгоритма шифрования над битами [77]. Алгоритм гомоморфного битового деления будет аналогичен алгоритму над открытыми данными, с той лишь разницей, что обрабатываются гомоморфно зашифрованные биты, из-за чего будут некоторые особенности реализации. Также может возникнуть необходимость выразить недостающие логические операции через те, которые поддерживаются гомоморфной схемой шифрования. Данный подход будет подробнее рассмотрен в параграфе 3.4. Гомоморфную реализацию деления со сдвигом делимого влево можно представить следующим образом.

Пусть дан некоторый полностью гомоморфный алгоритм шифрования над битами, для которого определены E(m) – алгоритм шифрования, D(c) – алгоритм расшифрования, обратный к E(m), $\otimes$,$\oplus$ – гомоморфные логические операции, определённые в данном алгоритме шифрования, где m – открытый текст, c – шифртекст. В общем виде алгоритм шифрования можно представить следующим образом:

1. Целое число m раскладываем побитно: $m = m_1 m_2 \ldots m_n$
2. Каждый бит шифруется отдельно: $C = E(m_1)E(m_2)\ldots E(m_n)$

Для облегчения выполнения операции разности в данной криптосистемы числа представляются в дополнительном коде. Алгоритм гомоморфного деления целых чисел, представленных в виде массива гомоморфно зашифрованных битов, можно представить следующим образом:

1. Нормализация делимого и делителя. В случае, если делитель и делимое имеют разную разрядность, нормализуем их разрядность к одной

величине – наибольшей разрядности среди них. В шифртекст с меньшей разрядностью для нормализации необходимо продублировать самый левый разряд (знаковый) необходимое число раз.

2. Определяем разрядность частного. Минимально допустимое значение модуля делителя равно 1, поэтому числитель по модулю не может быть больше делимого. Следовательно, разрядность частного будет равна разрядности делимого.

3. Формируем начальный остаток, заполняя его знаковым битом делителя.

4. Сдвигаем содержимое делимого влево, игнорируя знаковый бит. Выдвинутый бит вдвигаем справа в промежуточный остаток.

5. Прибавляем модуль делителя к остатку, взяв его с противоположным знаком. Фактически, это вычитание из старших разрядов делимого, которые мы постепенно перемещаем на место остатка. Если знак полученного остатка совпадает со знаком делителя, то значение следующего бита частного устанавливаем равным 1, иначе 0. Если знак полученного остатка совпадает со знаком делимого, то используем этот остаток на следующей итерации, иначе выполняем операцию «восстановления» остатка – используем предыдущее значение остатка.

6. Если сформированы n+1 битов частного, где n – разрядность частного, то переходим к следующему шагу. Иначе возвращаемся к шагу 4.

7. Прибавляем к частному единицу округления. В случае, если мы получаем отрицательный результат, начальное заполнение частного будет в обратном коде. Поэтому прибавление единицы автоматически округляет результат и переводит его в дополнительный код, в случае отрицательного результата.

8. Отбрасываем (n+1)-й бит частного. Полученное частное является результатом деления.

Данный подход универсален, а кроме гомоморфного деления поддерживает еще логические операции, что позволяет реализовать

практически любой алгоритм или метод вычислений гомоморфно. Однако сложность вычислений возрастает многократно, поэтому применять данный алгоритм нужно только в том случае, если другие подходы не предоставляют достаточный функционал.

2.5. Выводы

В главе рассмотрена проблема реализации операции гомоморфного деления. Проанализирована возможность реализации данной операции в некоторых алгоритмах гомоморфного шифрования.

Разработан метод гомоморфного деления на основе представления шифртекстов в виде простой дроби. Данный метод позволяет реализовать гомоморфное деление на основе любого полностью гомоморфного алгоритма шифрования над целыми числами. Метод хорошо подходит для программной реализации: для этого используется некоторая абстракция над шифртекстом, расширяющая возможности по выполнению математических операций над ним.

Разработан алгоритм гомоморфного побитового деления. Данный алгоритм позволяет реализовать операцию гомоморфного деления над шифртекстом, представленном и зашифрованном побитно с помощью полностью гомоморфного алгоритма шифрования над битами. Данный подход, конечно, увеличивает сложность вычислений, однако позволяет выполнять и логические, и арифметические операции в рамках одной схемы шифрования.

ГЛАВА 3. ГОМОМОРФНАЯ МАТЕМАТИКА НАД ЦЕЛЫМИ ЧИСЛАМИ НА ОСНОВЕ БИНАРНЫХ ОПЕРАЦИЙ

В гомоморфных алгоритмах над целыми обрабатываются исключительно целые числа, из-за чего реализация логических операций не представляется возможной или является ограниченной (пример логических операций в схеме шифрования над целыми числами приведен в параграфе 3.6). С другой стороны, на основе битовых операций возможно реализовать арифметические. Поэтому для решения вышеизложенной проблемы предлагается выполнить реализацию арифметических операций через логические операции на основе некоторого полностью гомоморфного алгоритма над битами [77, 84]. В качестве примера рассмотрим криптосистему Джентри над битами [54]. Она проявляет гомоморфизм относительно конъюнкции и исключающего «или», а в качестве единичного элемента можно использовать единицу или шифртекст единицы. Данный набор функций является полным и образует базис Жегалкина, следовательно, с его помощью можно выразить любую булеву функцию. Основные булевы функции выражаются следующим образом:

- Логическое отрицание реализуется посредством сложения по модулю 2 с единичным элементом: $\overline{x} = x \otimes 1$
- Дизъюнкция: $x \vee y = (x \wedge y) \otimes x \otimes y$

Следовательно алгоритм Джентри над битами поддерживает любые логические операции и можно воспользоваться им для реализации арифметических операций через битовые операции. Стоит отметить, что данная схема выбрана в только качестве примера, для реализации предлагаемых алгоритмов подойдет любая другая полностью гомоморфная схема над битами, поддерживающая все битовые операции над гомоморфно зашифрованными данными.

3.1. Схема шифрования

В предлагаемом подходе в качестве открытых данных будут выступать целые числа, а в процессе шифрования они будут представляться в виде массива (вектора) гомоморфно зашифрованных битов. Алгоритм шифрования можно представить следующим образом.

Пусть дана некоторая полностью гомоморфная схема шифрования над битами, E(x) – функция шифрования, D(x) – функция расшифрования. Тогда алгоритм шифрования целого числа M будет иметь следующий вид:

1. Представляем число в виде массива бит. Число бит должно быть кратным двум, при необходимости добавляем слева нулевой бит. Это необходимо, чтобы злоумышленник не мог сделать предположения о значении исходного бита в шифртексте на основе его позиции.

2. Переводим биты в дополнительный код. Это необходимо для упрощения вычислений. Дополнительный код положительного числа равен его прямому коду, а для получения дополнительного кода отрицательного числа инвертируем все его биты и прибавляем к нему 1. К полученному результату слева добавляем знаковый бит, для положительных чисел он равен 0, а для отрицательных 1. Получаем массив бит следующего вида: $M = m_n \dots m_1 m_0$.

3. Шифруем каждый бит отдельно: $C = E(m_n) \dots E(m_1) E(m_0)$, полученный массив зашифрованных битов является шифртекстом.

Алгоритм расшифрования:

1. Расшифровываем каждый элемент шифртекста. $D(C) = D(c_n) \dots D(c_1) D(c_0) = m'_n \dots m'_1 m'_0$

2. В результате получим массив битов открытого теста. Старший бит – знаковый. Проанализировав его значение, приведем биты к прямому коду:

   2.1. если знаковый бит равен 0, то число положительное, его прямой код равен дополнительному.

2.2. если знаковый бит равен 1, то число отрицательное. Для получения его значения инвертируем все его биты кроме знакового и прибавим 1.

3. Преобразуем полученную двоичную запись к десятичному виду.

Таким образом мы получили схему, в которой шифртекст представлен в двоичном виде в дополнительном коде, каждый бит при этом зашифрован гомоморфно и между двумя любыми битами можно выполнить гомоморфно любую логическую операцию. Если необходимо будет увеличить число бит числа, например, для предотвращения переполнения при выполнении операции сложения, необходимо продублировать знаковый бит необходимое число раз. Таким образом для положительного числа слева будут вставлены зашифрованные гомоморфно нули, а для отрицательного – зашифрованные гомоморфно единицы.

3.2. Операции сложения и вычитания

В вычислительных устройствах операции сложения и вычитания реализуются при помощи многоразрядных сумматоров и вычитателей соответствующе. Однако схемы данных устройств очень схожи и для упрощения реализации применяются универсальные устройства. В такой схеме выполняется сложение в системе обратного и дополнительного кода, а числа представляются в дополнительном коде, что позволяет заменить разность сложением. Сам процесс суммирования в данной схеме выполняется на классическом одноразрядном сумматоре, а количество одноразрядных сумматоров в схеме зависит от числа обрабатываем битов. Принципиальная схема одноразрядного двоичного сумматора приведена ниже:

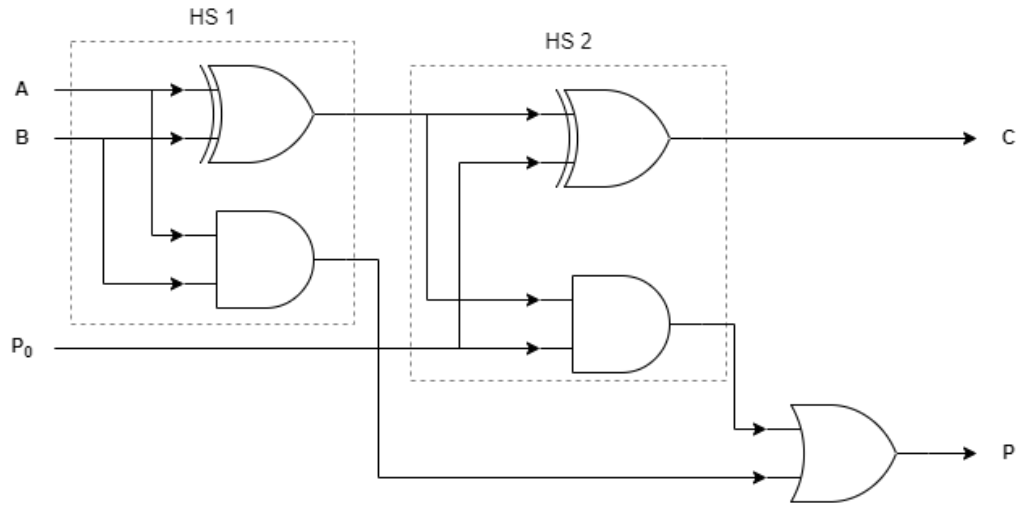

Рисунок 3.1 – Классический одноразрядный двоичный сумматор

В данной схеме A, B - соответствующие биты слагаемых, P_0 - входной бит переноса, C - результирующий бит, P - результирующий бит переноса. Пунктиром выделены два полусумматора HS 1 и HS 2, соединенных каскадно. Для получения бита переноса в схеме используются элемент “ИЛИ”.

Немного адаптируем схему, учитывая специфику схемы Джентри, а именно то, что операции конъюнкции и исключающего «или» доступны изначально и не ресурсоемкие, в то время как операция дизъюнкции реализуется через первые две и минимум в три раза более ресурсоемка. Поэтому упростим схему для нашего случая, заменив операцию дизъюнкции на исключающее «или».

Гипотеза: так как таблица истинности дизъюнкции и исключающего «или» эквивалентны кроме случая, когда ода входных операнда равны единице, а в вышеприведенной схеме оба входа никогда не могут быть равны единице, то мы можем выполнить замену дизъюнкции на исключающее «или».

Доказательство. Предположим обратное, следовательно оба входа приняли единичное значение. Перепишем значения на входах в виде формул (3.1) от входов схемы:

$$A \wedge B = 1;\ (A \otimes B) \wedge P_0 = 1 \qquad (3.1)$$

Результат конъюнкции равен единице только в том случае, если оба операнда равны единице. Получим (3.2):

$$A = 1; B = 1; (A \otimes B) = 1;\ P_0 = 1 \quad (3.2)$$

Выполнив подстановку, получим выражение (3.3):

$$(1 \otimes 1) = 1; 0 = 1 \quad (3.3)$$

Пришли к противоречию, следовательно оба выражения не могут быть равны единице ни при каких входных данных, и мы можем заменить операцию дизъюнкции на операцию исключающее «или».

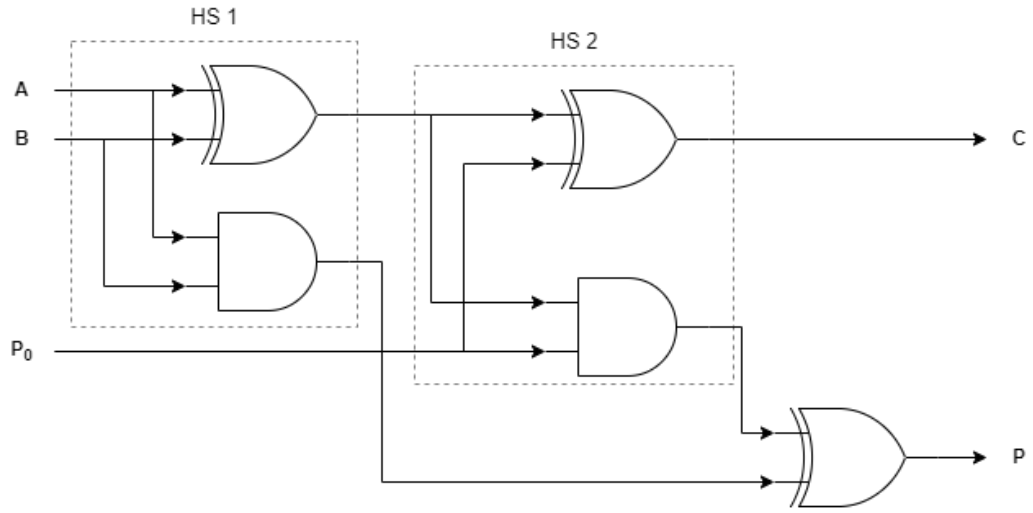

Рисунок 3.2 – Одноразрядный двоичный сумматор XOR-AND

Многоразрядное универсальное устройство для суммирования и вычитания имеет следующий вид (рисунок 3.3).

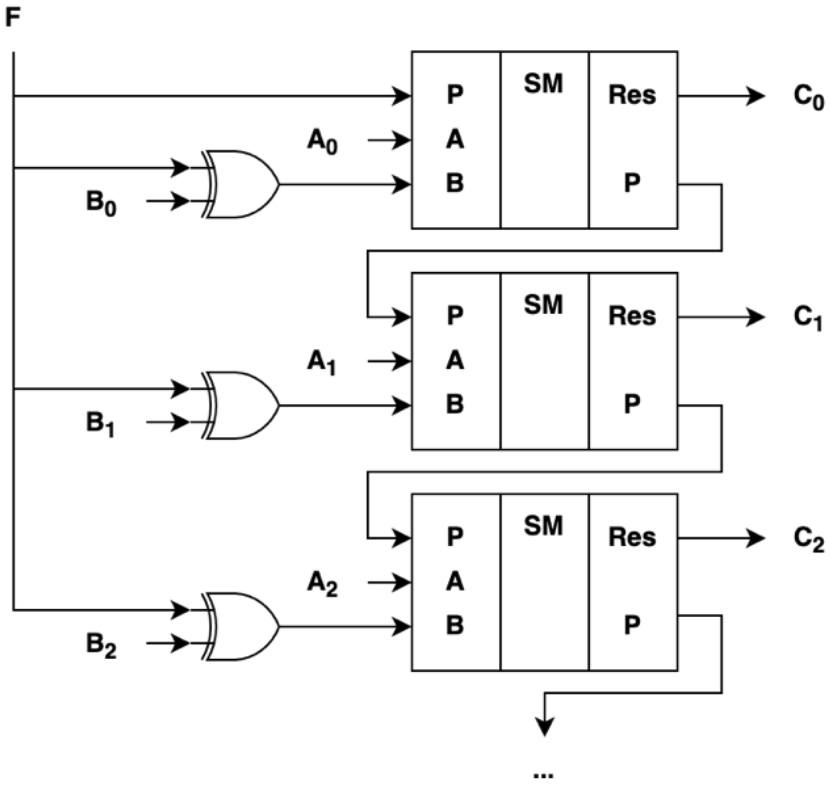

Рисунок 3.3 – Многоразрядное универсальное устройство

На вход F подается управляющий сигнал, который управляет режимом работы устройства. Начальный бит переноса приравнивается к F (для сложения 0, для вычитания 1). Если подается 0, то биты B не меняются и происходит обычное суммирование. Если подается сигнал 1, то биты входа B инвертируются и выполняется операция вычитания.

3.2.1. Алгоритм гомоморфной реализации

Гомоморфная реализация данной схемы может быть выполнена без каких-либо адаптаций, так как все операции в схеме имеют гомоморфную реализацию, а все операции выполняются линейно и результат может быть получен за фиксированное число шагов, основанном на числе двоичных разрядов шифртекста.

3.3. Операция умножения

Ввиду того, что было решено использовать числа в дополнительном коде для упрощения реализации операции деления, имеет смысл и все остальные операции также реализовать над числами в дополнительном коде, чтобы избежать операций перевода чисел из прямого кода в дополнительный и обратно.

Можно отметить три специфических варианта умножения чисел, представленных в дополнительном коде[3]:

- умножение с коррекцией результата в случае отрицательного множителя
- умножение с предварительным изменением знаков сомножителей в случае отрицательного множителя
- умножение путем последовательного преобразования множителя

Первые два способа требуют коррекции на различных этапах вычисления (в первом способе коррекция применяется к результату, а во втором – к сомножителям). Третий способ автоматически корректирует результат на каждом этапе, однако он более сложен в реализации. Рассмотрим каждый подход подробней.

3.3.1. Умножение с коррекцией результата

Основная идея данного метода умножения состоит в том, что в процессе умножения используются только разряды, представляющие цифровую часть множителя (без разряда алгебраического знака), но во всех суммированиях, выполняемых по ходу умножения, множимое А будет участвовать полностью с учетом знака.

---

[3] Ковригин Б. Н. Алгоритмы умножения //Москва: МГИФИ. – 2007.

Так как цифровые разряды в представлении множителя B представляют либо величину $|B|$ (если B > 0), либо величину 1-$|B|$ (если B < 0), то результат (псевдопроизведение) этого варианта умножения будет равен либо

A×|B| = A×B (если B ≥ 0) либо

A×(1-|B|)=A - A×|B|=A + A×B (если B < 0).

Так как в нашем случае биты зашифрованы, мы должны будем всегда выполнять коррекцию. В случае, если B отрицательное, мы должны вычитать из результата A, если B положительное, то будем вычитать 0. Следовательно, после умножение нам нужно будет вычесть $A \wedge b_n$, где b_n – знаковый бит числа B.

3.3.2. Умножение с предварительным изменением знаков сомножителей в случае отрицательного множителя

Коррекцию можно вообще исключить, если при отрицательном множителе провести изменением знаков сомножителей на противоположные. В этом случае следует проводить анализ знаков сомножителей и дополнительные операции преобразования кодов сомножителей. Если хоть один из сомножителей отрицательный, необходимо инвертировать знак сомножителей, а от их численной части взять дополнительный код. После коррекции сомножителей они умножаются как обычно, а знаковый разряд участвует в процессе умножения.

Для гомоморфной реализации данный вариант достаточно сложен, так как начальная коррекция включает в себя условие и дополнительные операции только в случае выполнения этого условия. Мы не можем знать значение зашифрованного бита, поэтому нам придется в любом случае выполнять корректировку сомножителей, а потом выбирать между скорректированным и исходным сомножителем на основе условия. То есть, требуется выполнить следующее преобразование для каждого сомножителя:

$$A' = ((a_n \vee b_n) \wedge A_{\text{дк}}) \vee (\overline{(a_n \vee b_n)} \wedge [-A_{\text{дк}}]_{\text{дк}}) \qquad (3.4)$$

$$B' = ((a_n \vee b_n) \wedge B_{\text{дк}}) \vee (\overline{(a_n \vee b_n)} \wedge [-B_{\text{дк}}]_{\text{дк}}) \quad (3.5)$$

3.3.3. Умножение путем последовательного преобразования множителя

Данный алгоритм умножения обеспечивает автоматическое введение поправок при любых знаках перемножаемых чисел. Множитель участвует в операции целиком, включая разряд алгебраического знака. В ходе умножения для каждого разряда рассчитывается частное произведение и его вес в произведение на основе сравнения его с разрядом справа от него. Для крайнего правого бита считается, что разряд справа равен нулю. Расчет веса частного произведения выполняется по следующему правилу:

- если разряд справа равен текущему, то вес равен 0 и этот разряд не участвует в умножении

- если текущий разряд 1, а разряд справа равен 0, то вес равен -1, то есть частичное произведение необходимо вычесть из промежуточного результата

- если текущий разряд равен 0, а разряд справа равен 1, то вес равен 1, то есть частичное произведение суммируем с промежуточным результатом.

3.3.4. Алгоритм гомоморфной реализация

Рассмотрим гомоморфную реализацию алгоритма побитового умножения на основе алгоритма умножения путем последовательного преобразования множителя. Выбор этого алгоритма обусловлен тем, что он прост в реализации и выполняется линейно за фиксированное число шагов, что важно при построении гомоморфной реализации.

Для гомоморфной реализации данный алгоритм можно представить следующим образом:

- на каждом этапе рассчитываем частичное произведение и суммируем его с промежуточным результатом

- частичное произведение равно $A \wedge (b_i \otimes b_{i-1}) \ll i-1$, нумерацию битов начинаем с младших разрядов, b_0 разряд принимаем равным 0

- полученное частичное произведение подаем на универсальное суммирующее устройство, сигнал, управляющий режимом работы, рассчитываем по формуле $F = (b_i \otimes b_{i-1}) \wedge b_i$. Если биты различны и текущий бит равен 1, то результат выражения будет равен 1 и будет активен режим разности, во всех иных случаях устройство будет работать в режиме суммирования.

3.4. Операция деления

Гомоморфное деление возможно реализовать через битовые операции на основе некоторого полностью гомоморфного алгоритма шифрования над битами. Для этого рассмотрим возможные подходы к выполнению операции деления над числами, представленными в двоичном коде. Как правило операция деления строится на последовательных операциях разности и сдвигов и возможны два варианта:

- деление со сдвигом делителя вправо
- деление со сдвигом делимого влево

Рассмотрим каждый из подходов и проанализируем, какой из них лучше подходит для гомоморфной реализации.

3.4.1. Деление со сдвигом делителя вправо

Алгоритм деления со сдвигом делителя вправо:

1. Проверка делителя на ноль. Это необходимо для корректной работы алгоритма, иначе выполнить шаг 2 будет невозможно. В случае, если делитель равен нулю, алгоритм завершается ошибкой.

2. Нормализация делителя. Для выполнения деления необходимо, чтобы старший разряд делителя был значащим, то есть содержал единицу. Для

достижения этого мы предварительно сдвигаем делитель влево необходимое число раз, а количество сдвигов k запоминаем. Шаги 3-5 мы будем повторять k+1 число раз.

3. Вычитаем делитель из делимого и получаем текущий остаток. На основе комбинации знаков делимого, делителя и текущего остатка формируем очередной разряд частного. Также на основе комбинации знаков либо восстанавливается предыдущий остаток, либо используется текущий.

4. Сдвиг делителя на один разряд вправо

5. Уменьшаем значение счетчика итераций на 1 и проверяем его значение. Если он стал равен нулю, то завершаем процесс деления, иначе возвращаемся к шагу 3.

Плюсом данного подхода является оптимизация процесса деления за счет нормализации делителя и уменьшения числа итераций. Но это является огромным недостатком при построении гомоморфной реализации данного алгоритма, так как итерации выполняются внутри внешней системы, а информация о количестве сдвигов зашифрована гомоморфно. Из-за этого мы не можем получить информацию о числе шагов и придется выполнять максимально возможное число шагов гомоморфно, что значительно усложнит реализацию алгоритма.

3.4.2. Деление со сдвигом делимого влево

Алгоритм деления со сдвигом делимого влево:

1. Формируем начальный остаток

2. Сдвигаем содержимое делимого влево, вдвигая выдвигаемый старший разряд в остаток.

3. Вычитаем делитель из остатка. Фактически, это вычитание из старших разрядов делимого, которые мы постепенно перемещаем на место остатка. Как и в первом варианте алгоритма, в зависимости от знака результата

формируются биты частного и при необходимости выполняется восстановление остатка.

4. Повторяем шаги 2-3, пока не будут сформированы n+1 битов частного (где n – разрядность частного)

5. Прибавляем к частному единицу для округления результата

Данный подход лучше подходит для гомоморфной реализации, так как не требует нормализации и выполняется всегда за фиксированное число шагов. Также алгоритм может обработать ноль в качестве делителя, однако в этом случае в частном будет некоторое ошибочное значение. Поэтому проверка на ноль и генерация ошибки необходима в любом случае. На основе данного подхода предложим метод гомоморфного деления на основе гомоморфных бинарных операций.

3.4.3. Алгоритм гомоморфной реализация

Алгоритм битового деления будет аналогичен алгоритму машинного деления, с той лишь разницей, что обрабатываются гомоморфно зашифрованные биты, из-за чего возникают некоторые особенности реализации.

Пусть дан некоторый полностью гомоморфный алгоритм шифрования над битами, для которого определены E(m) – алгоритм шифрования, D(c) – алгоритм расшифрования, обратный к E(m), $\otimes$,$\oplus$ – гомоморфные логические операции, определённые в данном алгоритме шифрования, где m – открытый текст, c – шифртекст. В общем виде алгоритм шифрования можно представить следующим образом:

1. Целое число m раскладываем побитно: $m = m_1 m_2 \ldots m_n$

2. Каждый бит шифруется отдельно: $C = E(m_1)E(m_2)\ldots E(m_n)$

Для облегчения выполнения операции разности в данной криптосистемы числа представляются в дополнительном коде. Алгоритм

гомоморфного деления целых чисел, представленных в виде массива гомоморфно зашифрованных битов, можно представить следующим образом:

3. Нормализация делимого и делителя. В случае, если делитель и делимое имеют разную разрядность, нормализуем их разрядность к одной величине – наибольшей разрядности среди них. В шифртекст с меньшей разрядностью для нормализации необходимо продублировать самый левый разряд (знаковый) необходимое число раз.

4. Определяем разрядность частного. Минимально допустимое значение модуля делителя равно 1, поэтому числитель по модулю не может быть больше делимого. Следовательно, разрядность частного будет равна разрядности делимого.

5. Формируем начальный остаток, заполняя его знаковым битом делителя.

6. Сдвигаем содержимое делимого влево, игнорируя знаковый бит. Выдвинутый бит вдвигаем справа в промежуточный остаток.

7. Прибавляем модуль делителя к остатку, взяв его с противоположным знаком. Фактически, это вычитание из старших разрядов делимого, которые мы постепенно перемещаем на место остатка. Если знак полученного остатка совпадает со знаком делителя, то значение следующего бита частного устанавливаем равным 1, иначе 0. Если знак полученного остатка совпадает со знаком делимого, то используем этот остаток на следующей итерации, иначе выполняем операцию «восстановления» остатка – используем предыдущее значение остатка.

8. Если сформированы n+1 битов частного, где n – разрядность частного, то переходим к следующему шагу. Иначе возвращаемся к шагу 6.

9. Прибавляем к частному единицу округления. В случае, если мы получаем отрицательный результат, начальное заполнение частного будет в обратном коде. Поэтому прибавление единицы автоматически округляет результат и переводит его в дополнительный код, в случае отрицательного результата.

10. Отбрасываем (n+1)-й бит частного. Полученное частное является результатом деления.

3.5. Условный оператор

Некоторые алгоритмы содержат в себе условные операторы, которые в общем виде можно представить как:

$$f(x) = \begin{cases} a, & \text{условие 1} \\ b, & \text{условие 2} \end{cases}$$

Условия могут быть любыми, но, как правило, сравниваются некоторые значения. Для построения гомоморфной реализации условного алгоритма необходимо разработать гомоморфную реализацию для каждого оператора неравенства и оператора равенства в виде некоторого метода, который в качестве входных параметров будет принимать два шифртекста и будет возвращать в качестве результата гомоморфно зашифрованный бит, который при расшифровании равен 1, если условие верно и 0 в обратном случае. Рассмотрим реализацию данного метода в рамках рассмотренной в 3.1 схемы шифрования.

Пусть c – шифртекст, представленный в виде массива зашифрованных гомоморфно битов, $f(c_1;c_2)$ – некоторый гомоморфный алгоритм, возвращающий шифртекст от 1 в случае, если сравнение чисел выполняется, $\wedge, \vee, \oplus$ – гомоморфные операции конъюнкции, дизъюнкции и исключающего «или» над шифртекстами. Тогда условие вида «если числа верны, результата равен первому числу, иначе результат равен второму числу» можно в общем виде выразить как:

$$c_3 = c_1 \oplus f(c_1; c_2) \vee c_2 \oplus \overline{f(c_1; c_2)} \qquad (3.6)$$

Приведенное выше выражение можно адаптировать при необходимости под условия решаемой задачи. Для реализации сравнения двух целых чисел,

представленных в двоичном виде в дополнительном коде и зашифрованных побитно предлагается следующий алгоритм.

Пусть A, B – два шифртекста, представленных в дополнительном коде и зашифрованных с использованием алгоритма 3.1, для которых определена функция разности, а для каждой пары битов (a_i, b_j) определены гомоморфные операторы исключающего «или», конъюнкции и дизъюнкции, тогда:

Для проверки равенства чисел сравниваем их побитно и суммируем результаты сравнений: $r = \overline{(a_0 \oplus b_0) \wedge (a_1 \oplus b_1) \wedge \ldots \wedge (a_n \oplus b_n)}$. Если все биты попарно равны, то числа равны и мы получим в результате 1, иначе 0.

Строгие сравнения чисел можно выполнить следующим образом:

1. Вычисляем разность C = A – B
2. Вычисляем r – результат проверки чисел на равенство
3. Результат сравнения определяется на основе комбинации знакового бита c_0 и результата сравнения r. Напомним, что знаковый бит равен 1, если число отрицательное, и 0, если положительное.

3.1. $A < B;\ c_0$ – результат отрицательный, только если A строго меньше B

3.2. $A \leq B;\ c_0 \vee r$ – результат отрицательный или числа равны

3.3. $A > B;\ \overline{c_0 \vee r}$ – инверсия выражения $A \leq B$

3.4. $A \geq B;\ \overline{c_0}$ – инверсия выражения A < B.

На основе приведенных выше формул можно получить гомоморфно зашифрованный результат любого сравнения чисел, который можно использовать в разработке гомоморфной реализации алгоритмов. например, при реализации алгоритма Гаусса на основе предложенного метода можно определять нулевые элементы на главной диагонали и выполнять перестановку строк при необходимости.

3.6. Поддержка логических операций в схемах над целыми числами

Рассмотренные в параграфах 3.1 – 3.4 алгоритмы реализации гомоморфных побитовых операций над целыми числами позволяют использовать и логические, и арифметические операции в рамках одной системы шифрования. Это необходимо для реализации многих прикладных алгоритмов. Один из подобных алгоритмов, алгоритм Гаусса, уже был рассмотрен в работе в качестве примера. Для его реализации нужны операции суммы, разности, умножения и деления, а также операция сравнения чисел с нулем. Однако, побитовые операции очень сильно увеличивают размер шифртекста и усложняют вычисления. Например, в алгоритме Джентри шифртекст бита и целого числа будут иметь примерно равный размер при условии выбора одинаковых параметров безопасности схемы шифрования, поэтому представление 64 битного целого числа в виде массива зашифрованных битов увеличит его размер примерно в 64 раза по сравнению с шифрованием целого числа. Проблемы увеличения размера шифртекста и высокой вычислительной сложности являются общими для всей гомоморфной криптографии, поэтому основной задачей при выполнении гомоморфной реализации некоторого алгоритма является оптимизация вычислений. Важно использовать минимально возможные средства для решения задач, к примеру, если планируется выполнять только сложение шифртекстов, имеет смысл использовать частично гомоморфный аддитивный алгоритм, если планируется выполнять небольшое число операций – уровневое гомоморфное шифрование и так далее. Как было сказано в начале главы, выполнение логических операций в гомоморфной схеме шифрования над целыми числами чаще всего невозможно, либо очень ограничено. Однако, при наличии такой возможности необходимо ее использовать. Так, в схеме BGV над целыми числами можно реализовать операцию сравнения с нулем, а также выбор одного из двух значений на основании результата сравнения. Это возможно благодаря тому, что схема BGV строится на модулярной арифметике.

Чтобы сравнить число с нулем, необходимо воспользоваться малой теоремой Ферма (3.1):

$$m^{p-1}\ mod\ p \equiv 1\ mod\ p \tag{3.1}$$

Результат выражения будет равен 1, если m отлично от нуля, либо равно 0, если m равно 0. Для выполнения операции выбора из двух значений необходимо получить инверсию результат сравнения. Для этого вычислим следующие выражения:

$$\mathrm{c} = m^{p-1}\ mod\ p \tag{3.2}$$

$$\bar{\mathrm{c}} = (\mathrm{c} * -1) + 1\ mod\ p \tag{3.2}$$

Алгоритм гомоморфного выбора из двух гомоморфно зашифрованных чисел a, b на основе условия c будет иметь вид:

$$d = (\mathrm{a} * c + b * \bar{c}) mod\ p \tag{3.3}$$

Данный подход не подойдет для реализации алгоритма Гаусса, потому что модуль должен быть очень большим, чтобы в результате выполнения операций число не превышало данный модуль, иначе будет получен некорректный результат. Однако это может пригодиться для решения других задач, например, при обработке цветов, где каждый цветовой канал имеет фиксированное максимальное значение.

Также аналогичным образом можно реализовать сравнение целых чисел на равенство. Пусть даны два целых числа a, b, зашифрованные гомоморфно в системе с модулярной арифметикой с модулем p, тогда для сравнения чисел необходимо вычислить следующее выражение:

$$eq = (a - b)^{p-1}\ mod\ p \tag{3.4}$$

В случае, если числа равны, их разность даст 0 и в результате также будет 0, во всех других случаях в результате будет 1. Для инверсии результата можно также воспользоваться формулой 3.2.

Таким образом, в некоторых гомоморфных схемах шифрования над целыми числами также возможно реализовать логические операции. Несмотря на то, что их число ограничено, это может быть полезно для решения некоторых задач, так как алгоритм над целыми числами даст гораздо лучшее быстродействие по сравнению с алгоритмами побитной арифметики.

3.7. Поддержка рациональных чисел

При необходимости алгоритм обработки целых чисел можно преобразовать для поддержки обработки рациональных чисел с низкой точностью вычислений. Для этого преобразуем исходный алгоритм следующим образом. Пусть дано целое число *m*, алгоритм шифрования *Enc*, алгоритм расшифрования *Dec*, гомоморфная реализация операций сложения, разности, умножения и деления. Тогда для поддержки рациональных чисел можно ввести дополнительный коэффициент *k*, равный 10^n, где *n* - число знаков после запятой, определяющее точность вычислений. Перед шифрованием число умножается на этот коэффициент, а также после каждой операции результат корректируется с учетом этого коэффициента.

Умножение:

$$c_1 \otimes c_2 = \frac{Enc(m_1 * k) \otimes Enc(m_2 * k)}{Enc(k)} = Enc(m_1 * m_2 * k) \quad (3.7)$$

Сумма и разность:

$$c_1 \oplus c_2 = Enc(m_1 * k) \oplus Enc(m_2 * k) = Enc((m_1 \pm m_2) * k) \quad (3.8)$$

Деление:

$$c_1 / c_2 = \frac{Enc(m_1 * k)}{Enc(m_2 * k)} * Enc(k) = Enc(\frac{m_1}{m_2} * k) \quad (3.9)$$

После расшифровки необходимо разделить полученный результат на коэффициент и будет получен результат выполнения операций в виде рационального числа. Это достаточно простой в реализации и гибкий подход,

подходящий для любого полностью гомоморфного алгоритма над целыми числами, однако точность вычислений намного ниже, чем в случае использования представления чисел с плавающей точкой.

3.8. Пример программной реализации

Рассмотрим, как можно выполнить программную реализацию предложенных алгоритмов. Так как алгоритмы строятся на основе битовых операций, имеет смысл выделить в программной реализации два уровня представления данных – на одном уровне будут выполняться гомоморфные операции над битами, а на другом, более высоком уровне, на основе битовых операций будет реализована гомоморфная арифметика. Пример программной реализации предложенных алгоритмов приведен на рисунке 3.4.

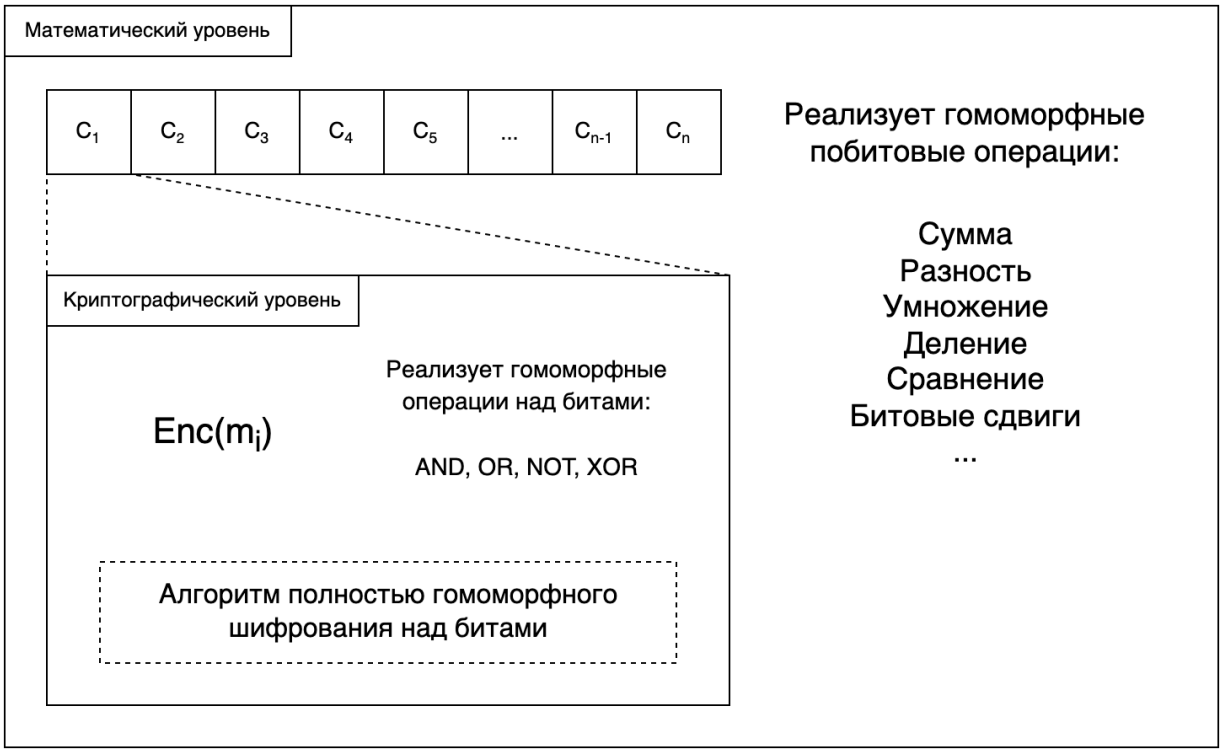

Рисунок 3.4 – Программная реализация побитовых гомоморфных операций

В основе криптографического уровня лежит некоторый алгоритм полностью гомоморфного шифрования над битами, а шифртекст представлен

гомоморфно зашифрованным битом. Криптографический уровень поддерживает выполнение гомоморфных операций между двумя гомоморфно зашифрованными битами – конъюнкция, дизъюнкция, исключающее ИЛИ. Благодаря тому, что операции более высокого уровня строятся на основе операций над битами, алгоритм гомоморфного шифрования над битами может быть любым, единственное требование к алгоритму – он должен быть полностью гомоморфным, чтобы позволять выполнять неограниченное число гомоморфных операций. Как правило криптографический уровень представляется некоторой криптографической библиотекой, реализующей гомоморфное шифрование.

Над криптографическим уровнем строится надстройка – математический уровень. Шифртекст на математическом уровне представляется в виде массива гомоморфно зашифрованных битов – объектов криптографического уровня. Математический уровень реализует алгоритм шифрования целых чисел, посредством их преобразования в массив битов и поэлементного шифрования битов с помощью криптографического уровня, а также алгоритм расшифрования на основе обратного преобразования. Также на математическом уровне реализуются арифметические и логические операции на основе алгоритмов, рассмотренных в данной главе.

Рассмотренная схема программной реализации может быть использована для расширения функционала существующих криптографических библиотек полностью гомоморфного шифрования и последующей реализации прикладных алгоритмов обработки данных.

3.9. Оценка сложности побитовых операций

Минусами гомоморфной криптографии являются увеличение объема данных при зашифровании, а также высокая ресурсоемкость и временная

емкость выполнения операций над гомоморфно зашифрованными данными. Поэтому важно оценить разработанные алгоритмы по этим критериям.

Так как шифртексты в предлагаемой схеме будут представлены в виде массива гомоморфно зашифрованных битов, то увеличение объема данных будет зависеть от числа битов шифртекста. Если предположить, что размерность шифртекста целого числа и бита приблизительно равны, то при использовании побитовой арифметики объем шифртекста увеличится в n раз по сравнению с алгоритмом шифрования над целым числом, где n – число гомоморфно зашифрованных битов.

Так как время выполнения операций может очень сильно отличаться в зависимости от используемой схемы гомоморфного шифрования, от выбранных параметров безопасности и от аппаратных характеристик устройства, на котором выполняются гомоморфные операции, то оценка будет дана относительная, основанная на подсчете числа вложенных операций над битами. Также оценка числа сложенных операций может быть полезна для оценки возможности применения пороговых схем гомоморфного шифрования, которые позволяют выполнять ограниченное число последовательных операций над гомоморфно зашифрованными данными.

Рассмотрим увеличение сложности вычислений на примере схемы Джентри, в которой гомоморфная операция XOR соответствует сумме шифртекстов, а операция AND – перемножению шифртекстов.

При оценке глубины по умножению для упрощения примем следующее допущение: так как операция гомоморфной суммы увеличивает зашумленность шифртекста во много раз медленнее, чем операция гомоморфного умножения, а также операция гомоморфной суммы выполняется небольшое число раз, по сравнению с числом операций гомоморфного умножения, то при оценке можем пренебречь числом операций гомоморфного сложения шифртекстов.

3.9.1. Сложение и вычитание

На рисунке 3.2 рассмотрен одноразрядный сумматор на основе XOR-AND элементов. На рисунке 3.3 рассмотрено многоразрядное суммирующее/вычитающее устройство, которое строится на основе одноразрядных сумматоров, подключенных последовательно, а над одним из входных битов выполняется операция XOR для управления режимом работы. Таким образом, над каждым разрядом шифртекста, представленного в виде массива гомоморфно зашифрованных битов, выполняется 4 гомоморфных операции XOR и 2 гомоморфных операции AND. Следовательно, на обработку каждого бита будет приходиться 4 операции суммы и 2 операции умножения.

Для оценки глубины алгоритма по умножению необходимо рассчитать максимальную степень, возникающую в процессе вычислений. Значение этой степени минус 1 и будет являться необходимой минимальной глубиной по умножению. Изучив схему одноразрядного сумматора (рисунок 3.2), а также алгоритм гомоморфного сложения/вычитания (пункт 3.2.1), выводим следующую формулу (3.10):

$$PW = \max\left(PW_a + PW_b; \max(PW_a; PW_b) + PW_f\right) + (n-1) * \max(PW_a; PW_b) \quad (3.10)$$

где PW – максимальная степень шифртекста, max(a,b) – функция выбора максимального значения, PW_a, PW_b, PW_f – степени шифртекстов от битов слагаемых A и B, а также бита режима суммирования F соответственно. К примеру, если в алгоритм суммирования подаются «свежие» шифртексты, на которыми не выполнялись операции, то их степени будут равны 1, а максимальная степень PW = n+1, а требуемая глубина по умножению будет равна n.

Число гомоморфных операций для разного числа битов шифртекста, а также требуемая глубина по умножению приведены в таблице 3.1.

Таблица 3.1 – Сложность выполнения операций сложения и вычитания

Число битов	Гомоморфное сложение	Гомоморфное умножение	Глубина по умножению
16	64	32	16
32	128	64	32
64	256	128	64

На основе результатов анализа можно сделать вывод, что для реализации побитового гомоморфного сложения и вычитания возможно использование пороговых схем ГШ, однако для минимального числа вложенных операций, так как глубина по умножению будет увеличиваться приближенно на величину $\approx n^m$, где n – число битов шифртекста, m – число вложенных операций.

3.9.2. Умножение

Алгоритм гомоморфной реализации побитового умножения приведен в пункте 3.3.4. Как видно из алгоритма, он состоит из n итераций, где n – число битов шифртекста.

Оценим число вложенных битовых операций. Каждая итерация состоит из одной гомоморфной суммы битов, одного гомоморфного умножения битов, одного поэлементного умножения вектора на бит и одной операции сложения/вычитания. Таким образом число сложений будет равно $n * (1 + n * 4)$, а число умножений равно $n * (1 + 3 * n)$.

Для оценки глубины по умножению рассмотрим алгоритм детальнее. На каждой итерации вычисляется частичное произведение, путем поэлементного гомоморфного умножения, его максимальная степень равна 2. Далее формируется гомоморфно зашифрованный бит, отвечающий за режим работы алгоритма суммирования/вычитания, равный гомоморфному произведению бита веса и текущего бита второго множителя, максимальная степень которого

также равна 2. Далее на каждой итерации выполняется суммирования промежуточного результата и частичного произведения. Степень промежуточного результата возрастает после каждой итерации, воспользуемся формулой (3.10): на первой итерации она равна n, на второй $n + 2 + (n - 1) * n \approx n^2$, на n-й итерации глубина будет приближенно равна n^n.

Число гомоморфных операций над битами для алгоритма побитового умножения для разного числа битов шифртекста, а также требуемая глубина по умножению приведены в таблице 3.2.

Таблица 3.2 – Сложность выполнения операции умножения

Число битов	Гомоморфное сложение	Гомоморфное умножение	Глубина по умножению
16	1040	784	$\approx 16^{16}$
32	4128	3104	$\approx 32^{32}$
64	16448	12352	$\approx 64^{64}$

На основе результатов анализа можно сделать вывод, что для реализации побитового гомоморфного умножения можно использовать только схемы ПГШ, потому что пороговые схемы не смогут обеспечить такую глубину по умножению.

3.9.3. Деление

Алгоритм гомоморфной реализации побитового деления приведен в пункте 3.4.3. Как видно из алгоритма, он состоит из n+1 итераций, где n – число битов шифртекста. На каждой итерации выполняется гомоморфная сумма/разность между промежуточным остатком и делителем. После этого выполняется операция восстановления остатка, по время которой берется предыдущее значение промежуточного остатка, если текущий знак остатка

отличается от знака делимого. После формирования n+1 битов результата необходимо прибавить к результату единицу округления.

Оценим число гомоморфных операций сложения и умножения над гомоморфно зашифрованными битами. Одна сумма выполняется для вычисления знакового бита результата. Каждая из n+1 итераций формирования результирующих битов включает в себя следующие операции. Две суммы для формирования бита режима суммирования. Побитовое сложение промежуточного остатка и делителя, размерность слагаемых n битов. Две суммы для вычисления очередного бита результата. Две суммы для формирования битов сравнения знаков промежуточного остатка и делимого. Процедура восстановления остатка требует n итераций, на каждой из которых выполняется 3 умножения и 2 суммы. После формирования n+1 битов результат выполняется побитовая сумма размерностью n+1. Таким образом, для выполнения побитового деления требуется $(n+1)*(2+4n+2+2+2n)+(n+1)*4=(n+1)*(10+6n)$ операций гомоморфной суммы битов и $(n+1)*(2n+3n)+(n+1)*2=(n+1)*(2+5n)$ операций гомоморфного умножения битов.

Оценим глубину по умножению. Как было рассмотрено ранее, операция гомоморфной побитовой суммы увеличивает глубину по умножению на $\approx n^m$, где n – число битов шифртекста, m – число последовательных операций гомоморфной побитовой суммы, в нашем случае m = n + 1. Также на каждой итерации выполняется операция восстановления остатка, состоящая из 3 последовательных умножений, следовательно глубина по умножению будет равна $\approx (4n)^{n+1}$, также после формирования n+1 битов частного выполняется сумма с единицей для округления результата, следовательно результирующая сложность будет равна $\approx (4n)^{n+2}$.

Число гомоморфных операций над битами для алгоритма побитового деления для разного числа битов шифртекста, а также требуемая глубина по умножению приведены в таблице 3.3.

Таблица 3.3 – Сложность выполнения операции деления

Число битов	Гомоморфное сложение	Гомоморфное умножение	Глубина по умножению
16	1802	1394	$\approx 64^{18}$
32	6666	5346	$\approx 128^{34}$
64	25610	20930	$\approx 256^{66}$

На основе результатов анализа можно сделать вывод, что для реализации побитового гомоморфного деления можно использовать только схемы ПГШ, потому что пороговые схемы не смогут обеспечить такую глубину по умножению.

3.10. Выводы

В главе рассматривается возможность построения гомоморфных операций над целыми числами на основе гомоморфных операций над битами. Данный подход позволяет выполнять и логические, и арифметические операции в рамках одной гомоморфной схемы шифрования.

Разработаны алгоритмы шифрования и расшифрования. В результате шифрования целое число представляется в виде массива битов, а каждый бит шифруется с использованием полностью гомоморфного алгоритма шифрования над битами. Алгоритм шифрования выполняет обратное преобразование.

Разработаны алгоритмы гомоморфной реализации побитовых целочисленных операций сложения, разности, умножения и деления, которые могут быть выполнены на основе любого полностью гомоморфного алгоритма шифрования над битами. Разработанные алгоритмы позволяют выполнять полный перечень арифметических операций. Благодаря побитному представлению возможна реализация сложных логических выражений над

гомоморфно зашифрованными данными. Дана оценка сложности разработанных алгоритмов, выраженная в числе вложенных гомоморфных операций над битами.

ГЛАВА 4. ГОМОМОРФНАЯ МАТЕМАТИКА НАД РАЦИОНАЛЬНЫМИ ЧИСЛАМИ НА ОСНОВЕ БИНАРНЫХ ОПЕРАЦИЙ

В главе 3 были рассмотрены алгоритмы побитовой арифметики над целыми числами, представленными побитно в формате с фиксированной точкой. Данный подход позволяет реализовать любые арифметические и логические операции в рамках одной схемы шифрования, что позволяет выполнить гомоморфную реализацию любого алгоритма, имеющего линейную структуру и выполняющегося за фиксированное число шагов. Однако, как и в операциях над незашифрованными данными, представление чисел в формате с фиксированной запятой имеет ряд недостатков по сравнению с форматом с плавающей точкой – небольшая размерность числа и низкая точность при выполнении округления после операции деления. Конечно, можно ввести поддержку нескольких разрядов после запятой, как рассмотрено в параграфе 3.7, однако точность вычислений все равно будет достаточно невысокой. Для решения этой проблемы предлагается рассмотреть возможность гомоморфной реализации побитовых операций над рациональными числами в формате с плавающей точкой [85]. Это должно решить проблему с низкой точностью вычислений, а также позволит хранить числа большей размерности, используя то же число гомоморфно зашифрованных битов.

4.1. Представление рациональных чисел в ЭВМ

В современных ЭВМ числа с плавающей точкой, как правило, представляются в прямом коде в виде мантиссы и порядка. Один из наиболее распространенных форматов представления чисел с плавающей точкой – IEEE 754 [86]. Кроме представления самих чисел стандарт имеет несколько особых

значений, отражающих ошибки при выполнении вычислений. Как правило в программных комплексах используются форматы представления чисел одинарной точности (32 бита) и двойной точности (64 бита). Однако также встречаются кратные реализации – половинная точность (16 бит), четверная точность (128 бит) и так далее. Рассмотрим в качестве примера представление числа в формате половинной точности. Другие форматы представляются абсолютно аналогично, однако имеют другую размерность составляющих. Стандарт определяет для числа с плавающей точкой половинной точности формат, приведенный в таблице 4.1.

Таблица 4.1 – Представление числа с плавающей точкой половинной точности

Диапазон	15	10…14	0…9
Число бит	1	5	10
Компонента	Знаковый бит	Смещенный порядок	Мантисса

Знаковый бит определяет знак числа. 0 – положительное число, 1 – отрицательное.

Смещенный порядок – это значение порядка минус смещение. Смещение вводится специально, чтобы не вводить еще один бит знака. Смещение порядка равно половине максимального значения, например для числа половинной точности оно равно 01111_2. При этом значения смещенного порядка 00000_2 и 11111_2 – это специальные значения, используемые для представления специальных чисел. Все специальные значения приведены в таблице 4.2.

Таблица 4.2 – Значения чисел в формате половинной точности в стандарте IEEE 754.

Порядок	Мантисса = 0	Мантисса ≠ 0
00000_2	+0 и -0	Денормализованные числа
00001_2 ... 11110_2	Нормализованные числа	
11111_2	$\pm\infty$	NaN («не число»)

Перед записью в память значение мантиссы нормализуется так, чтобы старшим значащим знаком была единица. При этом сама единица в память не записывается, это называется неявным старшим разрядом. Это позволяет дополнительно увеличить точность, так как размерность мантиссы получается равной 11 битам, в то время как явно в памяти хранятся только 10 бит.

В качестве примера представим числа 1.99 и -2.5 в данном формате:

$1.99_{10} = 1.1111110101_2 = 1.1111110101_2 * 10^0 = 0\ 01111\ 1111110101_2$

$-2.5_{10} = -10.1_2 = -1.01 * 10^1 = 1\ 10000\ 0100000000_2$

Представление чисел в формате с плавающей точкой позволяет хранить намного большие числа, используя то же число бит, что и числа в формате с фиксированной точкой. Сравнение форматов приведено в таблице 4.3.

Таблица 4.3 – сравнение форматов представления числа.

Критерий	Число с фиксированной точкой (64 бита)	Число с плавающей точкой (64 бита)
Минимальное число	-2^{63}	$-1.7 * 10^{308}$
Максимальное число	$2^{63} - 1$	$1.7 * 10^{308}$
Минимальное число (>0)	1	$2.3 * 10^{-308}$

Как видно из приведенной таблицы, число в формате с плавающей точкой поддерживает куда больший диапазон чисел, что позволит избежать переполнения, а также имеет более высокую точность. Стоит также отметить, что числа в формате с плавающей точкой в отличие от чисел с фиксированной точкой распределены неравномерно, сетка чисел более густая для чисел с малыми порядками и более редкая для чисел с большими порядками. И хотя минимальное положительное число равно $2.3 * 10^{-308}$, это не означает, что числа отличаются на эту величину. Точность вычислений в формате с плавающей точкой описывается машинным эпсилоном. Машинный эпсилон – это числовое значение, меньше которого нельзя задать относительную точность для любого формата с плавающей точкой. Машинный эпсилон можно описать как минимальное число ε, для которого выполняется условие 4.1:

$$1 + \varepsilon > 1 \qquad (4.1)$$

Для чисел с плавающей точкой одинарной точности (32 бита) абсолютное значение машинного эпсилона приблизительно равно $5.96 * 10^{-8}$, что обеспечивает точность около 7 знаков после запятой, а для чисел двойной точности (64 бита) $\varepsilon = 1.11 * 10^{-16}$, что обеспечивает 15 значащих цифр после запятой. Что обеспечивает достаточно высокую точность вычислений, в особенности по сравнению с представлением чисел в формате с фиксированной точкой.

Рассмотрим основные арифметические операции.

4.1.1. Сложение и вычитание

Пусть m – мантисса, e – смещенный порядок, тогда сложение и вычитание чисел в формате с плавающей точкой выполняется по следующему алгоритму:

1. Вычисляем разницу порядков $p = p_1 – p_2$

2. Если порядки равны и разность равна 0, переходим к следующему шагу, иначе выполняем выравнивание порядков. Для выравнивания порядков мантиссу числа с меньшим порядком сдвигаем вправо на абсолютное значение разницы порядков |p|. Сдвинутые младшие разряды при этом теряются.

3. После выравнивания порядков выполняем требуемую операцию (сложение или вычитание) над мантиссами с учетом знака числа.

4. Нормализуем полученное значение мантиссы при необходимости.

5. Порядок результата берется равным большему порядку чисел.

4.1.2. Умножение

Пусть m – мантисса, e – смещенный порядок, тогда сложение и вычитание чисел в формате с плавающей точкой выполняется по следующему алгоритму:

1. Порядок результата равен сумме порядков $p = p_1 + p_2$.

2. Мантиссы необходимо перемножить.

3. Нормализуем результат при необходимости.

4.1.3. Деление

Пусть m – мантисса, e – смещенный порядок, тогда сложение и вычитание чисел в формате с плавающей точкой выполняется по следующему алгоритму:

1. Порядок результата равен разности порядков $p = p_1 - p_2$.

2. Мантиссы необходимо разделить.

3. Нормализуем результат при необходимости.

4.2. Гомоморфная реализация

Ввиду того, что числа с плавающей точкой представляются в виде целочисленных мантиссы и порядка, их можно реализовать гомоморфно, зашифровав побитно с помощью полностью гомоморфного алгоритма шифрования над битами.

Все операции над числами в формате с плавающей точкой состоят из операций над целочисленными порядком и мантиссой, следовательно, легко могут быть реализованы через битовые операции. Однако сложности возникают с операциями нормализации и приведения чисел к одной степени. Сложность заключается в том, что данные обрабатываются в зашифрованном виде, а управляющий алгоритм не имеет к ним доступа. Из-за этого он должен обрабатывать данные “вслепую” и должен корректно обработать любой возможный результат. Можно получить, к примеру, разницу степеней двух чисел, но эта разница будет в зашифрованном виде и воспользоваться ей для определения величины сдвига или числа итераций не получится.

Приведение чисел к одной степени. Данная операция осложнена тем, что и мантиссы, и порядки зашифрованы и управляющий алгоритм не знает, сколько раз ему нужно повторить сдвиг или в какой момент нужно остановиться. Поэтому понадобится найти модуль разницы степеней, а после этого циклически сдвигать меньшее из чисел вправо на один разряд и уменьшать разницу степеней на 1, пока степень не достигнет нуля. Так как значение разницы степеней неизвестно, необходимо рассматривать худший вариант - сдвиг мантиссы на максимальное число ее разрядов. Чем больше бит выделено на мантиссу, тем выше сложность данной операции.

Нормализация результата вычислений. В результате выполнения операций над мантиссами может получиться ненормализованный результат. Так как данные зашифрованы, мы не можем определить факт возникновения ненормализованного результата, поэтому нужно выполнять нормализацию после каждой операции. Однако можно предугадать диапазон отклонения

числа после выполнения операции и упростить нормализацию для некоторых арифметических операций. Известно, что нормализованное число имеет формат 1.xxx в двоичном виде, а также может быть нулевым, то есть находится в диапазоне 0 ИЛИ [1…2) в десятичном виде.

4.2.1. Схема шифрования

Шифрование. Для шифрования представим шифруемое число в двоичном виде в формате с плавающей точкой. Каждый бит исходного числа шифруется с помощью полностью гомоморфного алгоритма шифрования над битами. Полученный массив зашифрованных битов является шифртекстом.

Расшифрование. Чтобы расшифровать шифртекст, необходимо применить к нему обратное преобразование. Каждый зашифрованный бит расшифровать с помощью полностью гомоморфного алгоритма шифрования над битами, полученный массив битов и будет открытым текстом в двоичном виде в формате с плавающей точкой. При необходимости полученный результат можно перевести в десятичную дробь.

4.2.2. Сложение и разность

Как было рассмотрено в пункте 4.1.1, сложение и разность чисел в формате с плавающей точкой сводятся к выполнению операций приведения к одному порядку, побитовой суммы или разности и нормализации. Операция побитовой суммы и разности была подробно рассмотрена в параграфе 3.2. Для операций приведения к одному порядку и нормализации необходимо разработать алгоритмы гомоморфной реализации с учетом специфики работы с зашифрованными данными.

Для приведения чисел к одному порядку разработаем гомоморфную реализацию алгоритма, рассмотренного в пункте 4.1.1. Работая с гомоморфно зашифрованными данными, мы не можем узнать, на какое число разрядов необходимо выполнить сдвиг. Поэтому необходимо поэтапно предпринимать

попытку сдвига мантиссы, а чтобы быть уверенным в результате, необходимо рассматривать худший случай – необходимость сдвига на максимальное число разрядов. Алгоритм приведения двух чисел A, B в формате с плавающей точкой, зашифрованных побитно с использованием схемы ПГШ над битами, можно представить следующим образом:

1. Вычисляем разность порядков C. Порядки в числе в формате с плавающей точкой – целые положительные числа. Поэтому для выполнения операции побитовой разности добавляем к каждому числу слева шифртекст от 0 (знаковый знак положительного числа) и выполняем разность, согласно алгоритму, рассмотренному в пункте 3.2.1. Знаковый бит c_0 будет шифртекстом от 1, если порядок числа B строго больше порядка числа A. Поэтому результирующий порядок вычислим по формуле 4.2:

$$E_c = (c_0 \wedge E_b) \vee (\overline{c_0} \wedge E_a) \tag{4.2}$$

2. Устанавливаем счетчик i = 0

3. Выполняем попытку сдвига. Для этого вычисляем условие необходимости сдвига каждого из чисел. Сдвигать необходимо число с наименьшей степенью. Следовательно, сдвигать число A нужно, если e_0 = Enc(1), сдвигать число B нужно, если e_0 = Enc(0) и e_0 не равно нулю. Обозначим соответствующие условия как r_a, r_b. Сдвиг мантиссы каждого из чисел гомоморфно вычисляем по формуле:

$$M = (\mathrm{r} \wedge (\gg \mathrm{M})) \vee (\overline{\mathrm{r}} \wedge \mathrm{M}) \tag{4.3}$$

4. Прибавляем к разности порядков единицу, взятую с противоположным знаком.

5. Если i = n – 1, где n число двоичных разрядов мантиссы, то завершаем алгоритм. Иначе возвращаемся к шагу 3.

Чтобы выполнить нормализацию чисел после выполнения операции проанализируем возможное значения результата. Обе данные операции в

результате сводятся к операции суммы чисел, которые могут иметь одну из четырех комбинаций знаков. В случае, если суммируются два числа с одним знаком, результат не может уменьшиться, а может только увеличиться на один разряд за счет переноса в старший разряд, а модуль результата будет в диапазоне 0 ИЛИ [1…4). Если же складываются числа с разными знаками, то модуль результата может быть в диапазоне [0…2). Таким образом, при выполнении операции необходимо учитывать возможное переполнение на 1 бит. Для нормализации необходимо проверить, что она требуется, сравнив старший бит с единицей – если старший бит равен единице, а бит переполнения равен нулю, то число уже нормализовано. Иначе его необходимо либо сдвинуть вправо на 1 разряд, либо влево на n разрядов, пока в старший разряд не будет смещена единица. Над зашифрованными данными алгоритм будет иметь следующий вид. Пусть мантисса имеет n бит в памяти, старший единичный бит не записывается явно, бит переноса p. Тогда для нормализации числа необходимо:

1. Вычисляем бит, указывающий на необходимость нормализации e = p.
2. Сдвигаем число вправо на 1 разряд. Записываем в результат $(\gg m \wedge e) \vee (m \wedge \overline{e})$. Прибавляем к степени e. Обнуляем бит переноса.
3. Устанавливаем индекс цикла i = 0
4. Вычисляем бит, указывающий на необходимость нормализации e = $\overline{m_n}$
5. Сдвигаем число влево на 1 разряд. Записываем в число $(\ll m \wedge e) \vee (m \wedge \overline{e})$. Вычитаем из степени e.
6. Если i < n, возвращаемся к шагу 3, иначе завершаем алгоритм.

4.2.3. Умножение

Операция произведения сводится к сумме порядков, перемножению мантисс и нормализации результатов. Алгоритм умножения над гомоморфно зашифрованными данными можно представить в виде:

1. Выполнить операцию побитовой гомоморфной суммы порядков. Порядок числа – гомоморфно зашифрованное побитно целое число. Для выполнения операции гомоморфной суммы необходимо к порядкам чисел прибавить слева знаковый бит (шифртекст от нуля) и выполнить над ними операцию суммы, используя алгоритм, рассмотренный в параграфе 3.2.

2. Выполнить операцию гомоморфного побитового перемножения мантисс, используя алгоритм из параграфа 3.3. Так как операция будет выполняться над рациональными числами, а также возможно переполнение разрядной сетки, необходимо это учесть, увеличив число битов результата.

3. Выполнить нормализацию результата.

Для выполнения нормализации необходимо оценить, какой результат может быть получен. Модуль произведения будет находиться в диапазоне 0 ИЛИ [1, 4). Поэтому при нормализации достаточно учесть бит переноса. Алгоритм нормализации будет иметь вид:

1. Вычисляем бит, указывающий на необходимость нормализации e = p.

2. Записываем в мантиссу числа результат выражения $(\gg m \wedge e) \vee (m \wedge \overline{e})$. Прибавляем к степени e. Обнуляем бит переноса.

4.2.4. Деление

Операция деления сводится к разнице порядков, перемножению мантисс и нормализации результатов. Алгоритм умножения над гомоморфно зашифрованными данными можно представить в виде:

1. Выполнить операцию побитовой гомоморфной разности порядков. Порядок числа – гомоморфно зашифрованное побитно целое число.

Для выполнения операции гомоморфной разности необходимо к порядкам чисел прибавить слева знаковый бит (шифртекст от нуля) и выполнить над ними операцию разности, используя алгоритм, рассмотренный в параграфе 3.2.

2. Выполнить операцию гомоморфного побитового деления мантисс, используя алгоритм из параграфа 3.4.

3. Выполнить нормализацию результата.

Модуль частного будет находится в диапазоне 0 или (0,5…2) или бесконечность. Таким образом, если в результате деления не была получена бесконечность или ноль, то результат достаточно сдвинуть 1 раз вправо, чтобы нормализовать. Ноль и бесконечность – особые значения, при которых мантисса равна нулю. Таким образом алгоритм нормализации будет иметь вид:

1. Вычисляем бит, указывающий на необходимость нормализации $e = \overline{m_n} \wedge M \neq 0$.

2. Сдвигаем число влево на 1 разряд. Записываем в число ($\ll m \wedge e$) $\vee$ ($m \wedge \overline{e}$). Вычитаем из степени e.

4.3. Оценка сложности побитовых операций

Побитовые гомоморфные операции над числами в формате с плавающей точкой строятся на базе побитовых гомоморфных операций над целыми числами. Сложность соответствующих операций была рассмотрена в параграфе 3.9. В дополнение к целочисленным побитовым операциям необходимо оценить сложность выполнения операций нормализации и приведения к одному порядку. Наиболее популярные размерности чисел в формате с плавающей точкой приведены в таблице 4.4.

Таблица 4.4 – Размерность чисел в формате с плавающей точкой

Число битов	Мантисса (бит)	Порядок (бит)

16	11	5
32	24	8
64	53	11

Рассмотрим увеличение сложности вычислений на примере схемы Джентри, в которой гомоморфная операция XOR соответствует сумме шифртекстов, а операция AND – перемножению шифртекстов.

При оценке глубины по умножению для упрощения примем следующее допущение: так как операция гомоморфной суммы увеличивает зашумленность шифртекста во много раз медленнее, чем операция гомоморфного умножения, а также операция гомоморфной суммы выполняется небольшое число раз, по сравнению с числом операций гомоморфного умножения, то при оценке можем пренебречь числом операций гомоморфного сложения шифртекстов.

4.3.1. Сложение и вычитание

Гомоморфные побитовые операции сложения и вычитания над числами в формате с плавающей точкой состоят из приведения к одному порядку, побитовой гомоморфной операции сложения или разности над числом в формате с фиксированной точкой, а также последующей нормализации.

Сложность выполнения операции суммы или разности можно рассчитать на основе оценки из пункта 3.9.1.

Операция приведения к одному порядку состоит из ряда других операций. Разность порядков размерности k+1, где k – число битов порядка. Выбор одного из шифртекстов размерности k, на каждой итерации обработки битов выполняется 2 суммы и 3 умножения. Далее выполняется n итераций сдвига, каждая итерация состоит из выбора одного из шифртекстов размерности n, где n – число битов мантиссы, а также из суммы шифртекстов размерности k+1.

При нормализации выполняется n+1 итераций сдвигов мантисс, каждая из которых состоит из суммы/разности размерности k+1, а также выбора из двух шифртекстов размерности n, состоящего из n итераций, на каждой из которых выполняется 2 суммы и 3 умножения. Приблизительная оценка сложности выполнения операций гомоморфной побитовой суммы и разности над числом в формате с плавающей точкой приведена в таблице 4.5.

Таблица 4.6 – Сложность выполнения операции умножения

Число битов	Гомоморфное сложение	Гомоморфное умножение
16	1136	1048
32	4264	4500
64	16760	19744

Как видно из оценки, гомоморфные операции суммы и разности над числами с плавающей точкой имеют намного более высокую сложность, чем аналогичные операции над числами с фиксированной точкой, сложность выполнения которых была рассмотрена в пункте 3.9.2.

4.3.2. Умножение

Операция побитового умножения сводится к побитовой сумме порядков, побитовому умножению мантисс и нормализации результата. Оценка суммы и умножения будет аналогична рассмотренным в параграфе 3.9. Для нормализации выполняется два гомоморфных поэлементных умножения мантиссы на бит, а также операции поэлементного логического ИЛИ между битами результатов, которая стоится на двух операциях суммы и одной операции умножения. Также прибавляем к порядку бит переполнения, для этого выполняем операцию побитной гомоморфной суммы. Сложность выполнения операции умножения приведена в таблице 4.6.

Таблица 4.6 – Сложность выполнения операции умножения

Число битов	Гомоморфное сложение	Гомоморфное умножение
16	565	431
32	2448	1860
64	11491	8687

Таким образом, сложность побитовой гомоморфной операции умножения над числом в формате с плавающей точкой будет ниже, чем сложность аналогичной операции над числом в формате с фиксированной точкой.

4.3.3. Деление

Операция деления аналогична операции умножения, разница только в том, что над мантиссами выполняется операция деления, а не умножения. Разница порядков имеет аналогичную сложность, что и сумма. Нормализация после выполнения операции также аналогична. Сложность выполнения операции деления приведена в таблице 4.7.

Таблица 4.7 – Сложность выполнения операции умножения

Число битов	Гомоморфное сложение	Гомоморфное умножение
16	898	737
32	3970	3158
64	17914	15057

Таким образом, сложность побитовой гомоморфной операции деления над числом в формате с плавающей точкой будет ниже, чем сложность аналогичной операции над числом в формате с фиксированной точкой.

4.4. Выводы

В главе рассматривается возможность построения арифметических гомоморфных побитовых операций над числами в формате с плавающей точкой. Это позволит повысить точность вычислений по сравнению с алгоритмами над целыми числами, разработанными в главе 3.

Разработаны алгоритмы гомоморфной реализации побитовых операций сложения, разности, умножения и деления над числами в формате с плавающей точкой, которые могут быть выполнены на основе любого полностью гомоморфного алгоритма шифрования над битами. Разработанные алгоритмы позволяют выполнять арифметические и логические операции в рамках одного гомоморфного алгоритма шифрования, а также повышают точность вычислений по сравнению с алгоритмами над числами в формате с фиксированной точкой.

Дана оценка сложности разработанных алгоритмов, выраженная в числе вложенных гомоморфных операций над битами. Оценка показала, что операции гомоморфного сложения и вычитания над числами в формате с плавающей точкой намного сложнее аналогичных операций над числами в формате с фиксированной точкой. Однако операции умножения и деления, наоборот, имеют более низкую сложность. Следовательно, разработанные алгоритмы могут эффективней применяться в алгоритмах обработки данных, содержащих большое число операций суммы и умножения, и обеспечивать более высокую точность вычислений, по сравнению с алгоритмами над числами в формате с фиксированной точкой.

ГЛАВА 5. ПРАКТИЧЕСКАЯ АПРОБАЦИЯ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

5.1. Обработка цифровых изображений

Цифровые изображения – это двумерные изображения, представленные в цифровом виде. В зависимости от способа представления принято делить цифровые изображения на растровые и векторные. Растровые изображения выражаются в виде двумерного массива, каждые элемент которого – это минимальный элемент цифрового изображения, пиксель. Векторные изображения описываются формулами геометрических примитивов – точек, линий, окружностей и так далее. Из-за специфики представления, векторные и растровые изображения имеют разные свойства. Растровые изображения проще создавать и обрабатывать, однако они занимают больший объем памяти, а также при их масштабировании ухудшается качество. Векторные изображения обладают противоположными свойствами.

Обработка и редактирование цифровых изображений – это процесс изменения оригинального изображения для достижения определенных целей [87-88]. В основном редактирование изображений требуется для достижения таких целей, как устранение дефектов изображения, структурное редактирование изображения, подготовка изображения к отображению на цифровых носителях или к печати. В работе рассматривается редактирование цифровых изображений, так как алгоритмы их обработки проще, и они чаще применяются на практике. Можно выделить следующие виды обработки изображений:

- изменение размеров изображения;
- редактирование яркости;
- редактирование контрастности;
- устранение нерезкости;

- замена и инверсия цветов;
- кадрирование;
- дорисовка, добавление надписей, символов;
- добавление спецэффектов, фильтров, теней.

В рамках данного исследования был выполнен только анализ алгоритмов масштабирования цифровых изображений [89]. Дальнейшие исследования в этом направлении были приостановлены, так как гомоморфная обработка изображений крайне неэффективна из-за роста размерности данных на несколько порядков при шифровании. А так как для представления цифрового изображения требуется большой объем данных, их обработка и передача по сети будет проблематичной задачей.

5.1.1. Масштабирование

Масштабирование растровых цифровых изображений – это процесс уменьшения или увеличения размеров изображения, которое зачастую дополняется алгоритмами постобработки, чтобы улучшить качество результирующего изображения. Основная проблема при уменьшении изображения – потеря части пикселей и снижение детализации, а при увеличении возникает обратная проблема, которую обычно называют «ступенчатостью» изображения. «Ступенчатость» возникает из-за того, что при увеличении один пиксель исходного изображения преобразуется в n пикселей результирующего изображения, где n – коэффициент увеличения, и визуально каждый пиксель просто становится больше. Для решения этих проблем применяются более сложные алгоритмы масштабирования, которые учитывают не только преобразуемый пиксель, но и соседние пиксели. Скорость и эффективность каждого метода масштабирования можно оценить на основе числа пикселей исходного изображения, которые участвовали в формировании конечного изображения, к общему числу пикселей исходного изображения.

Самым простым и универсальным методом масштабирования изображения является метод ближайшего соседа. Этот метод позволяет изменить размер изображения (уменьшить или увеличить) на любой коэффициент, при этом размеры результирующего изображения будут округлены до целого числа пикселей. Коэффициент увеличения по вертикали и горизонтали можно задавать раздельно, а если задать их равными, то изображение будет увеличено с сохранением пропорций. Алгоритм состоит из двух этапов. Сначала вычисляются размеры нового изображения:

$$w' = w * k_w;\ h' = h * k_h \tag{5.1}$$

где w, h – ширина и высота исходного изображения, w', h' – ширина и высота результирующего выражения, k_w, k_h – коэффициенты масштабирования изображения по горизонтали и вертикали соответственно.

После этого для каждой позиции пикселя результирующего изображения вычисляется позиция ближайшего соответствующего пикселя исходного изображения с учетом коэффициентов масштабирования. Вычисление координат (x; y) нужного исходного пикселя вычисляется по формулам (). В случае получения дробного числа, значение округляется вниз до целого.

$$x = \frac{(x'-1)*(w-1)}{w'-1} + 1 \tag{5.2}$$

$$y = \frac{(y'-1)*(h-1)}{h'-1} + 1 \tag{5.3}$$

Данные вычисления выполняются для каждого пикселя результирующего изображения, и каждая позиция заполняется значением пикселя исходного изображения (рис. 5.1). Очевидно, что в формировании каждого пикселя результирующего пикселя участвует только один пиксель исходного изображения, из чего следует низкое качество результата. Однако данный метод очень быстрый и часто применяется для формирования предварительного изображения, пока более качественный результат

вычисляется более трудоемким методом. Метод ближайшего соседа не требует применения гомоморфного шифрования, так как в перестановках участвуют только индексы, а значения самих пикселей не учитываются.

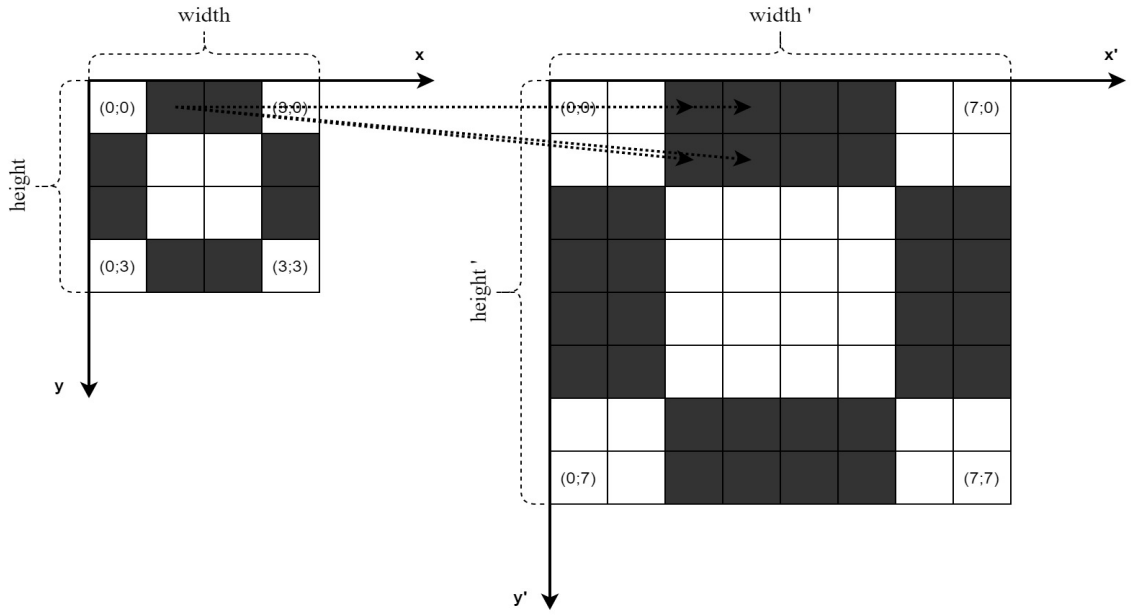

Рисунок 5.1 – Метод ближайшего соседа

Все более сложные алгоритмы масштабирования цифровых изображений учитывают в процессе масштабирования также значения соседних пикселей, а чем большее число соседних пикселей участвуют в вычислении значения результирующего пикселя, тем более качественных получается результат. В качестве примера рассмотрим семейство достаточно простых алгоритмов масштабирования пиксельной графики EPX/Scale2x/AdvMAME2x, чтобы получить представление об общих принципах работы подобных алгоритмов и проанализировать возможности их гомоморфной реализации.

EPX («Eric’s Pixel eXpansion», пиксельное увеличение Эрика) – алгоритм масштабирования пиксельной графики, разработанный Эриком Джонстоном из LucasArts при портировании игрового движка с IBM PC, где разрешение было 320 на 200 пикселей, на первые цветные компьютеры Macintosh, где разрешение экрана было больше примерно вдвое. AdvMAME2x

и Scale2x – это разработанные позднее модификации алгоритма EPX; они имеют более эффективную, но функционально идентичную реализацию. Данные алгоритмы масштабирования цифровых изображений созданы специально для увеличения графических изображений низкого качества. Они дают в качестве результата менее размытую картинку, чем традиционные алгоритмы и позволяют избежать ступенчатости, как при использовании метода ближайшего соседа. Следует отметить, что данные алгоритмы позволяют увеличивать изображение только в кратное двум число раз (2x, 4x) и не позволяют уменьшать изображение. В общем виде вышеописанные алгоритмы состоят из следующих шагов (рис 5.2).

- для каждого пикселя исходного изображения генерируются 4 пикселя результирующего изображения и изначально красятся в цвет исходного пикселя
- цвет пикселей результирующего изображения корректируется на основе значения пикселей, пограничных с исходным
- если у исходного пикселя нет соседних пикселей с одной или нескольких сторон, то значение недостающих пикселей принимается равным значению рассматриваемого пикселя

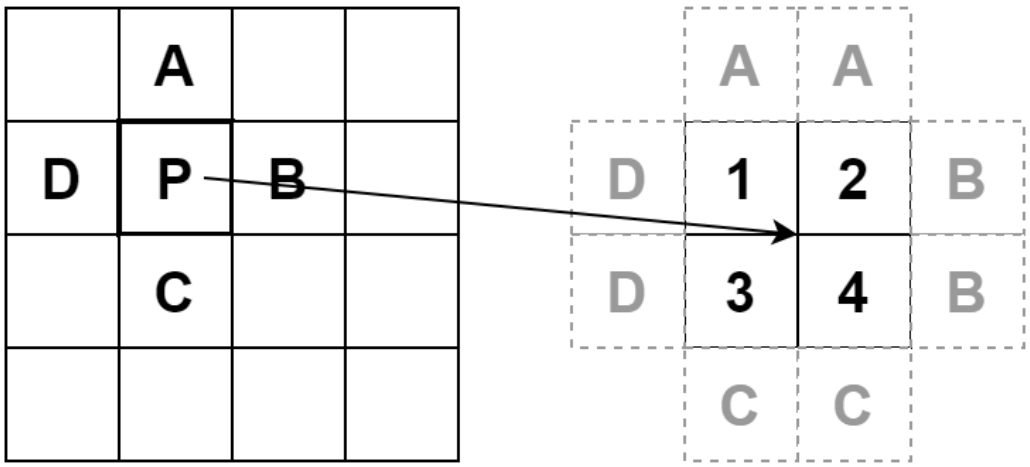

Рисунок 5.2 – Алгоритм EPX

Корректировка цвета результирующих пикселей в алгоритме EPX выполняется следующим образом:

- Если A = D, то 1 = A, иначе 1 = P
- Если A = B, то 2 = A, иначе 2 = P
- Если C = D, то 3 = C, иначе 3 = P
- Если C = B, то 4 = C, иначе 4 = P

Если значения пикселей A и D исходного изображения равны, то пиксель результирующего изображения 1 красится в цвет пикселя A, иначе цвет пикселя 1 остается равен P (начальному заполнению). Аналогичные операции выполняются для остальных пикселей результирующего изображения.

Рассмотрим возможность гомоморфной реализации описанного выше алгоритма. Для корректировки цветов необходимо выполнить сравнение двух пикселей. Следовательно, для выполнения гомоморфной реализации алгоритма EPX нам понадобится метод выбора одного из двух значений на основе результата сравнения, описанный в параграфе 3.5. Предположим, что обрабатывается цветное изображение, представленное в формате RGB – каждый пиксель выражается тремя цветовыми компонентами, каждая из которых равна целому числу в диапазоне [0; 255]. Адаптируем алгоритм сравнения чисел для сравнения цветов пикселей.

Пусть даны два пикселя X, Y, которые представлены цветовыми компонентами (R_x, G_x, B_x) и (R_y, G_y, B_y) соответственно, каждый из компонентов представлен в виде массива гомоморфно зашифрованных бит $R = r_1 r_2 \dots r_n$. Тогда результат сравнения этих пикселей можно получить по формулам (4.4-4.7).

$$e_r = \left(r_1^x \oplus r_1^y\right) \vee \left(r_2^x \oplus r_2^y\right) \vee \dots \vee \left(r_n^x \oplus r_n^y\right) \quad (5.4)$$

$$e_g = \left(g_1^c \oplus g_1^d\right) \vee \left(g_2^c \oplus g_2^d\right) \vee \dots \vee \left(g_n^c \oplus g_n^d\right) \quad (5.5)$$

$$e_b = \left(b_1^c \oplus b_1^d\right) \vee \left(b_2^c \oplus b_2^d\right) \vee \dots \vee \left(b_n^c \oplus b_n^d\right) \quad (5.6)$$

$$e = \overline{e_r \vee e_g \vee e_b} \quad (5.7)$$

где $r_{1...n}$, $g_{1...n}$, $b_{1...n}$ – гомоморфно зашифрованные биты соответствующих цветовых компонент размерности n, e – бит, соответствующий результату сравнения двух пикселей.

Для удобства обозначим предложенный вариант сравнения двух гомоморфно зашифрованных пикселей как метод Eq(X, Y). Используя этот метод, выполним гомоморфную реализацию алгоритма масштабирования EPX. Начальное заполнение пикселей результирующего выражения эквивалентно алгоритму ближайшего соседа, поэтому выполняется одинаково для открытых и зашифрованных данных. Гомоморфную реализацию процесса корректировки цветов на основе соседних пикселей выполним следующим образом:

- $E_1 = A\ eq\ B;\ 1 = (E_1 \wedge A) \vee (\overline{E_1} \wedge P)$
- $E_2 = A\ eq\ C;\ 2 = (E_2 \wedge A) \vee (\overline{E_2} \wedge P)$
- $E_3 = D\ eq\ B;\ 3 = (E_3 \wedge D) \vee (\overline{E_3} \wedge P)$
- $E_4 = D\ eq\ C;\ 4 = (E_4 \wedge D) \vee (\overline{E_4} \wedge P)$

Таким образом возможно выполнить увеличение изображение гомоморфно, однако данное решение является очень ресурсоемким ввиду того, что операции выполняются над гомоморфно зашифрованными битами.

5.2. Метод Гаусса для решения СЛАУ

Метод Гаусса – это классический метод решения СЛАУ, в ходе которого последовательно исключаются переменные при помощи элементарных преобразований, и система приводится к треугольному виду (прямой ход метода Гаусса), а затем из полученной системы последовательно находятся все переменные (обратный ход метода Гаусса).

Метод Гаусса отлично подходит для решения систем линейных алгебраических уравнений, обладающий следующими преимуществами:

- нет необходимости исследовать систему на совместность;
- метод можно применять и к системам, в которых число уравнений не равно числу неизвестных, а основная матрица системы вырожденная;
- метод результативен при сравнительно небольшом числе операций;
- возможность получить результат при фиксированном числе операций

Для выполнения гомоморфной реализации метода Гаусса необходимо адаптировать его алгоритм с учетом того, что будут обрабатываться гомоморфно зашифрованные данные и управляющий алгоритм не будет знать реальные значения обрабатываемых данных [90-91]. Поэтому алгоритм должен учитывать это и давать результат за фиксированное число шагов для любых исходных данных.

5.2.1. Предварительное исследование СЛАУ

Предварительное исследование СЛАУ Так как данные зашифрованы, мы можем получить только целочисленное решение системы. Поэтому приступать к решению системы, которая заведомо не имеет решений, либо имеет бесконечное множество решений, не имеет смысла.

Если число переменных больше числа уравнений, то можно сразу сделать вывод о бесконечном числе решений в случае совместности системы или об отсутствии решений в обратном случае, завершаем алгоритм с ошибкой. Так как шифруются коэффициенты СЛАУ, а не индексы, данную проверку можно выполнить над гомоморфно зашифрованными данными так же, как над открытыми данными.

5.2.2. Прямой ход

Во время прямого хода метода Гаусса матрица приводится к треугольному виду. Для этого уравнения переставляются при необходимости, чтобы в главной диагонали матрицы находился ненулевой элемент. После перестановки уравнений коэффициенты в текущем столбце ниже главной диагонали обнуляются. При выполнении гомоморфной реализации прямого хода метода Гаусса возникают две основные проблемы: сложность проверки, что в главной диагонали находится ненулевой элемент и переполнение размерной сетки.

Основная сложность проверки элементов на главной диагонали заключается в том, что мы можем получить результат сравнения элемента с нулем, однако этот результат также будет зашифрован. Поэтому управляющий алгоритм не знает, необходима ли перестановка уравнений и не может остановиться на первом же уравнении с ненулевым элементом на главной диагонали. Для решения данной проблемы необходимо выполнять все возможные перестановки уравнений, что гарантирует, что если есть хоть одно уравнение с ненулевым элементом в текущем столбце, то оно будет подставлено в текущую строку. Алгоритм перестановки уравнений A, B при обработке гомоморфно зашифрованных данных, применяемый к каждой паре уравнений:

1. Вычисляем зашифрованный бит e, указывающий на равенство нулю текущего элемента на главной диагонали. Если элемент равен нулю, то e = 1, иначе e = 0.

2. Переставляем уравнения на основе значения бита e. Для вычисляем новые значения уравнений:

$$A = A \wedge \overline{e} \vee B \wedge e \tag{5.1}$$

$$B = B \wedge \overline{e} \vee A \wedge e \tag{5.2}$$

Таким образом, если бит e равен 1, то будет выполнена перестановка уравнений. В противном случае уравнения останутся без изменений. Перестановка будет выполнена только в том случае, если бит равен 0, как только будет найдено первое уравнения с ненулевым коэффициентом, перестановки будут завершены. Однако попытки перестановок все равно продолжатся, пока не будут перебраны все уравнения, так как управляющий алгоритм не знает, в какой момент будет найден ненулевой элемент.

Вторая проблема во время выполнения прямого хода над гомоморфно зашифрованными данными связана с потенциальным переполнением разрядной сетки. Числа в ЭВМ имеют фиксированную разрядность. Как только размерность числа превысит эту разрядность, в результате выполнения операции будет ошибочное значение. Во время выполнения прямого хода в результате перемножения уравнений для обнуления коэффициента каждый коэффициент в уравнении ниже текущего уравнения будет увеличен примерно в два раза. Данная операция будет выполняться n-1 раз, где n – число неизвестных в СЛАУ. Таким образом, размерность коэффициентов будет увеличена в 2^{n-1} раз. Предположим, что в криптосистеме на число отводится 64 бита, числа в криптосистеме представлены в целочисленном виде, начальная размерность коэффициентов до 2^{10}, тогда максимально возможная размерность системы, при которой не возникнет переполнения, n = 7. При n = 8 размерность шифртекста выйдет за границы 64 бит: $2^{10} * 2^{8-1} = 2^{70}$. Данную проблему можно решить нормализацией шифртекстов после каждой операции путем деления всех коэффициентов на левый ненулевой коэффициент, если он больше единицы. Также возможным решением будет использование криптосистемы с поддержкой чисел в формате с плавающей точкой. Число в формате с плавающей точкой двойной точности (64 бита) позволяет хранить числа размерности 2^{1023}. Значит, при начальной размерности коэффициентов до 2^{10} можно без переполнения обрабатывать системы с числом переменных до 103. Если же возникнет необходимость обрабатывать систему большей

размерности, то можно добавить операцию нормализации после каждого обнуления коэффициента, как было описано выше.

5.2.3. Обратный ход

Для гомоморфной реализации обратного хода алгоритма Гаусса требуется поддержка гомоморфных арифметических операций разности, умножения и деления. Данные операции возможно выполнить на основе предложенных в диссертационной работе алгоритмов. Никаких адаптаций для гомоморфной реализации алгоритма Гаусса не требуется.

5.2.4. Гомоморфная реализация метода Гаусса

В прошлом пункте были рассмотрены адаптации, необходимые для выполнения гомоморфной реализации алгоритма Гаусса. Просуммируем их и сформулируем алгоритм.

Пусть дано:

- некоторый полностью гомоморфный алгоритм шифрования над битами
- некоторая гомоморфная криптосистема, шифртекст которой представлены в виде массива гомоморфно зашифрованных битов, поддерживающая следующие операции: Enc(x) – шифрование бита x, гомоморфные сумма, разность, умножение, деление, логическое И (между битом и шифртекстом), логическое ИЛИ (между битом и шифротестом), Eq(x, y) – сравнение двух зашифрованных битов или шифртекстов, результатом которого является зашифрованный бит
- СЛАУ, коэффициенты которой являются шифртекстами гомоморфной криптосистемы и представлены в виде двумерного массива A размерности m×n, где m - число уравнений, n – число неизвестных; свободные

члены которой являются шифртекстами гомоморфной криптосистемы и заданы в виде массива B размерности m.

- бит e (бит ошибки), зашифрованный с помощью данного полностью гомоморфного алгоритма и указывающий на несовместность системы / отсутствие численных решений

Тогда алгоритм Гаусса для решения гомоморфно зашифрованной СЛАУ будет иметь следующий вид.

Предварительное исследование матрицы: если число переменных больше числа уравнений, то можно сразу сделать вывод о бесконечном числе решений в случае совместности системы или об отсутствии решений в обратном случае, устанавливаем бит e = Enc(1), завершаем алгоритм.

Прямой ход (приведение матрицы к треугольному виду). Пусть i = 1.

3. Переставить строки [i; m] так, чтобы в позиции a_{ii} был ненулевой элемент. Для каждого j из [i+1; m]:

3.1. Выполняем проверку на ноль:

$$c_i = Eq(a_{ii}, Enc(0)) \tag{5.1}$$

3.2. Для каждого k из [1; n] выполняем

$$a_{ik} = (a_{ik} \wedge c_i) \vee (a_{jk} \wedge \overline{c_i}) \tag{5.2}$$

$$a_{jk} = (a_{jk} \wedge c_i) \vee (a_{ik} \wedge \overline{c_i}) \tag{5.3}$$

То есть, если a_{ii} не равен нулю, то оставляем строки без изменения, иначе меняем их местами. Выполнив перестановку каждой пары уравнений, мы будем уверены, что если в системе есть хоть одно уравнение с ненулевым элементом в столбце i, оно будет поставлено в строку i.

Обнулить коэффициенты столбца i в строках [i+1; m]. Для этого:

3.3. для каждых k из [i; n] и l из [i+1; m] вычисляем

$$a_{lk} = a_{lk} \times a_{ii} - a_{ik} \times a_{il} \tag{5.4}$$

3.4. для каждого l из [i+1; m] вычисляем

$$b_l = b_l \times a_{ii} - b_i \times a_{il} \tag{5.5}$$

4. Увеличить i на 1, если i $\geq n$ - перейти к пункту 6, иначе вернуться к пункту 3 и повторить шаги для нового значения i.

5. Если число уравнений больше числа переменных, обнулить их коэффициенты и проверить, что уравнения [i; m] имеют смысл (имеют вид 0=0), иначе система несовместна и не имеет решений, завершаем алгоритм. Так как левая часть была приведена к 0 на шаге 4, нам достаточно проверить, что правая часть каждого уравнения также равна нулю. Для этого для каждого k из [i; m] вычислим

$$c_k = Eq(a_{kn}, Enc(0)) \tag{5.6}$$

$$e = c_i \vee \ldots \vee c_m \tag{5.7}$$

6. Обратный ход. Пусть i = n

7. Вычислим x_i

$$x_i = \left(b_i - \sum_{j=i+1}^{n} a_{ij} \times x_j \right) / a_{ii} \tag{5.8}$$

8. Уменьшить i на 1, если i < 1 – завершить обратный ход, все коэффициенты найдены, иначе вернуться к шагу 8.

Результатом выполнения алгоритма Гаусса служит массив коэффициентов X размерности n, а также бит ошибки e. После расшифровки сперва необходимо проверить бит ошибки, если он равен 1, значит в процессе решения было обнаружено, что система несовместна и не имеет решений, либо имеет бесконечное число решений, а значит массив коэффициентов содержит случайные или ошибочные числа. В обратном случае массив X содержит решение системы.

5.3. Программная реализация побитовых операций

Для проверки на практике результатов исследования была выполнена программная реализация алгоритмов гомоморфных целочисленных

арифметических операций на основе гомоморфных битовых операций. В качестве основы для реализации гомоморфной криптографии была использована библиотека Helib для MacOS. Для шифрования была использована схема BGV в режиме шифрования битов (пространство открытого текста равно 2). На основе алгоритмов, рассмотренных в главе 3, с использованием битовых гомоморфных операций, были реализованы арифметические гомоморфные операции суммы, разности, умножения и деления. Программный код приведен в приложении Б.

5.3.1. Схема программной реализации

Схема программной реализации представлена на рисунке 5.3.

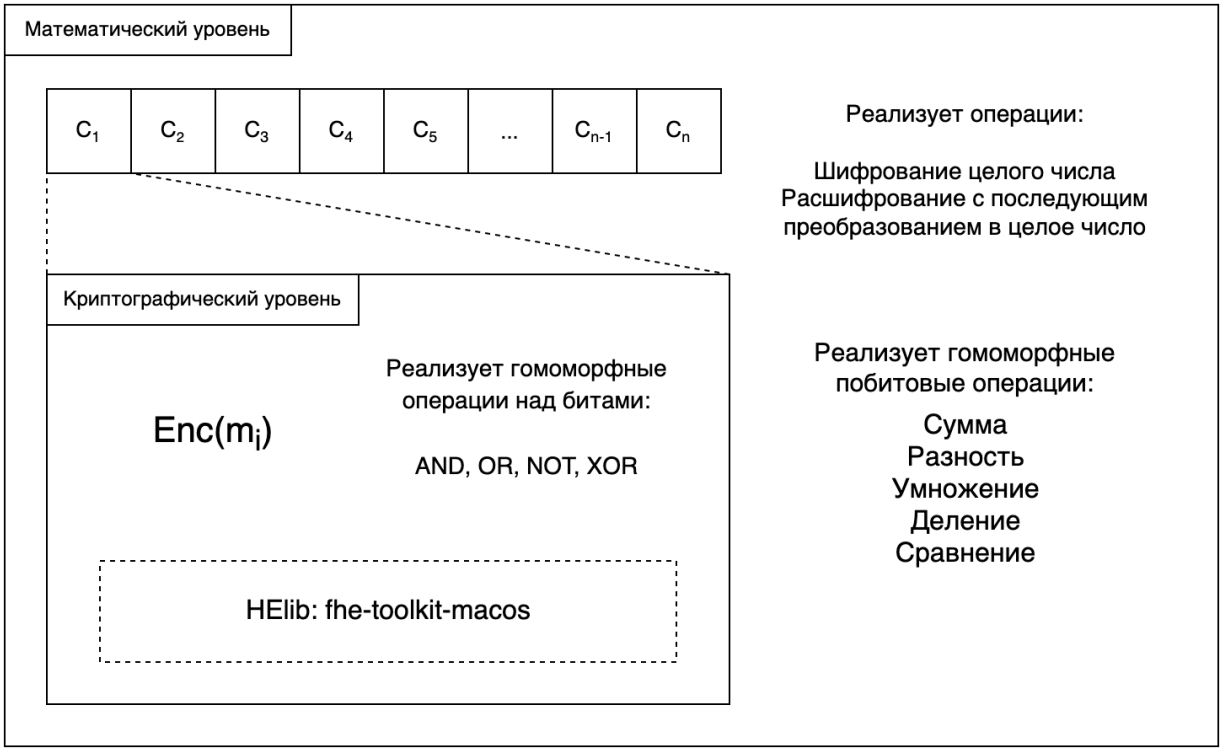

Рисунок 5.3 – Схема программной реализации побитовой арифметики

Как видно из рисунка, схема соответствует той, что была рассмотрена в главе 3. В качестве криптографического ядра (схемы ПГШ) используется криптографическая библиотека “HElib: fhe-toolkit-macos”. Программный код

приведен в диссертации в приложении Б. Рассмотрим подробнее каждый уровень представления данных.

5.3.2. Криптографический уровень

Класс, определяющий основной тип данных криптографического уровня, содержит в себе функционал, позволяющий генерировать ключи шифрования, выполнять операции шифрования и расшифрования а также основные гомоморфные операции над зашифрованными гомоморфно битами. В нашем случае криптографический уровень представлен библиотекой “HElib: fhe-toolkit-macos”. Интерфейс криптографического уровня можно представить в виде псевдокода следующим образом (рисунок 5.4).

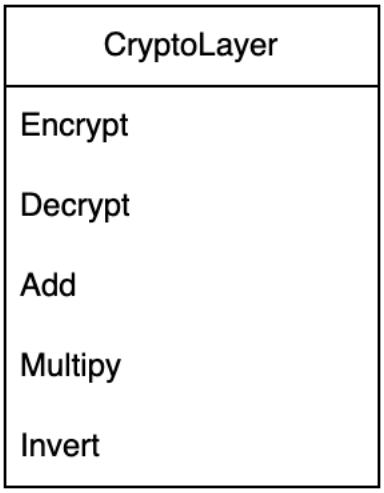

Рисунок 5.4 – Интерфейс класса криптографического уровня

В программной реализации использовался алгоритм BGV, в рамках этого алгоритма сумма шифртекстов эквивалентна операции исключающего ИЛИ над открытыми текстами (битами). Перемножение шифртекстов эквивалентно операции конъюнкции над открытыми текстами (битами). Инверсия зашифрованного бита реализуется через сумму шифртекста с шифртекстом от единицы (a XOR 1).

5.3.3. Математический уровень

Данный уровень является надстройкой над математическим. На этом уровне реализуется побитовая арифметика над гомоморфно зашифрованными битами. На математическом уровне шифруются целые числа, при шифровании числа переводятся в двоичную запись в дополнительный код. Перевод числа в дополнительный код, как было рассмотрено ранее, упрощает реализацию операции разности. Далее число в двоичной записи шифруется побитно с помощью криптографического уровня, а полученный массив гомоморфно зашифрованных битов будет являться шифртекстом математического уровня. Для расшифрования выполняется обратное преобразование: шифртекст расшифровывается побитно, биты переводятся из дополнительного кода в прямой код, число из двоичной записи переводится в десятичную запись. Над шифртекстами математического уровня реализуются арифметические и логические операции. Интерфейс класса математического уровня можно представить в виде псевдокода следующим образом (рисунок 5.5).

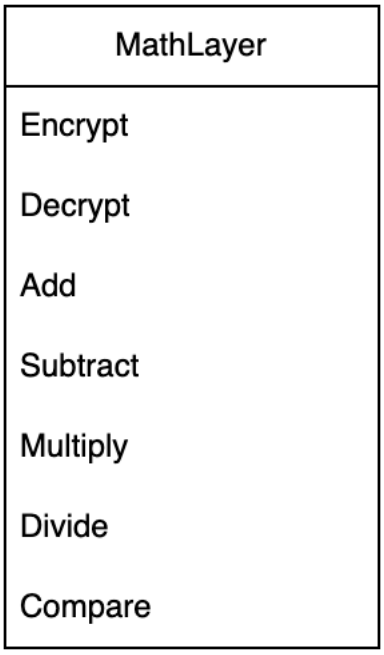

Рисунок 5.5 – Интерфейс класса математического уровня

Кроме арифметических операций на математическом уровне также реализуется сравнение чисел посредством их побитного сравнения. Кроме того для удобства на математическом уровне могут быть реализованы логические операции между математическим и криптографическим уровнем. К примеру, операция поэлементной конъюнкции между шифртекстом математического уровня и шифртекстом криптографического уровня.

5.3.4. Тестирование

Для проверки достоверности разработанных алгоритмов были подготовлены тесты, в рамках которых для каждой арифметической операции генерировались два случайных числа, шифровались побитно, над ними выполнялась гомоморфная арифметическая операция, результат которой расшифровывался и сравнивался с эталоном. Тесты были выполнены для разной размерности шифртекста (16 битов и 32 бита). Результаты тестирования приводятся в таблице 5.1.

Таблица 5.1. Результаты тестирования

Операция	16 битов		32 бита	
	Всего тестов	Успешно пройдено	Всего тестов	Успешно пройдено
Сложение	100	100	100	100
Разность	100	100	100	100
Умножение	100	100	100	100
Деление	100	100	100	100

Как видно из таблицы, все тесты были пройдены успешно. Следовательно, практическая апробация показала, что предложенные алгоритмы работают корректно.

5.4. Выводы

Выполнена практическая апробация результатов исследования. Программная реализация разработанных алгоритмов гомоморфной арифметики над битами, выполненная на основе криптографической библиотеки HElib подтвердила корректность работы разработанных алгоритмов.

Для проверки достоверности разработанных алгоритмов были подготовлены тесты, в рамках которых для каждой арифметической операции генерировались два случайных числа, шифровались побитно, над ними выполнялась гомоморфная арифметическая операция, результат которой расшифровывался и сравнивался с эталоном. Все тесты были пройдены успешно.

ЗАКЛЮЧЕНИЕ

В ходе диссертационного исследования была проделана следующая работа:

1. Проанализированы существующие разработки и исследования в области гомоморфного шифрования, в результате чего были сделаны выводы о недостаточной проработанности темы прикладного применения гомоморфной криптографии и выдвинуто предположение о необходимости разработки методов и алгоритмов, которые смогут расширить возможности прикладного применения существующих разработок.

2. Проанализирована проблема реализации гомоморфного деления, выполнен анализ ряда существующих алгоритмов гомоморфного шифрования и изучена возможность реализации данной операции математически.

3. Предложен метод гомоморфного деления, основанный на представлении шифртекста в виде простой дроби, и позволяющий выполнять операцию деления над гомоморфно зашифрованными данными. При этом сложность гомоморфных вычислений несколько увеличивалась, но данное снижение скорости вычислений можно считать приемлемым.

4. Предложена реализация криптосистемы над целыми числами, в которой целые числа представляются в виде массива гомоморфно зашифрованных битов, а гомоморфные операции над целыми реализуются через битовые операции. В рамках данной криптосистемы возможно выполнение арифметических и логических операций, что позволяет выполнить гомоморфную реализацию любого алгоритма обработки данных. Существенным недостатком данного решения является многократное увеличение сложности вычислений и объема шифртекста, поэтому данное решение необходимо применять только в тех случаях, когда недостаточно гомоморфной криптографии над целыми числами.

5. Предложена реализация криптосистемы над числами в формате с плавающей точкой, в которой рациональные числа в формате с плавающей точкой представляются в виде массива гомоморфно зашифрованных битов, а гомоморфные операции над целыми реализуются через битовые операции. В рамках данной криптосистемы возможно выполнение арифметических и логических операций, что позволяет выполнить гомоморфную реализацию любого алгоритма обработки данных. Данный подход позволяет существенно повысить точность вычислений, хоть и увеличивает сложность вычислений, по сравнению с криптосистемой над целыми числами.

6. Проанализированы возможности применения предложенных методов и алгоритмов и разработаны гомоморфные реализации алгоритмов масштабирования цифровой графики и алгоритма Гаусса.

7. Выполнена программная реализация разработанных алгоритмов побитовых гомоморфных операций над целыми числами. Проведенное тестирование с использованием данной программной реализации подтвердило корректность разработанных алгоритмов.

Основные результаты, полученные в диссертации, заключаются в разработке методов и алгоритмов полностью гомоморфного шифрования, которые расширяют возможности прикладного использования гомоморфной криптографии. Разработанные методы и алгоритмы могут быть использованы для реализации других программных решений, которые могут использоваться в облачных сервисах для реализации безопасных облачных вычислений (в том числе обработки цифровых изображений) и защищенного поиска информации.

СПИСОК ЛИТЕРАТУРЫ

1. Дерябин, М. А. Обзор безопасных методов шифрования для облачных вычислений / М. А. Дерябин, Н. Н. Кучеров // Новости науки в АПК. – 2019. – № 3(12). – С. 298-303.
2. Трубей, А. И. Гомоморфное шифрование: безопасность облачных вычислений и другие приложения (обзор) / А. И. Трубей // Информатика. – 2015. – № 1(45). – С. 90-101.
3. Варновский, Н. П. Методы пороговой криптографии для защиты облачных вычислений / С. А. Мартишин, М. В. Храпченко, А. В. Шокуров // Труды ИСП РАН. – 2014. – № 2.
4. Батура, Т. В. Облачные технологии: основные понятия, задачи и тенденции развития / Т. В. Батура, Ф. А. Мурзин, Д. Ф. Семич // Программные продукты и системы. – 2014. – № 3. – С. 64-72.
5. Жиров, А. О. Безопасные облачные вычисления с помощью гомоморфной криптографии / А. О. Жиров, О. В. Жирова, С. Ф. Кренделев // Безопасность информационных технологий. – 2013. – Т.1. – С. 6-12.
6. Афанасьев, С. В. Облачные сервисы, онтологическое моделирование таксономии / С. В. Афанасьев // Труды СПИИРАН. – 2012. – № 23. – С. 392-399.
7. Беккер, М. Я. Информационная безопасность при облачных вычислениях: проблемы и перспективы / М. Я. Беккер, Ю. А. Гатчин, Н. С. Кармановский, А. О. Терентьев, Д. Ю. Федоров // Научно-технический вестник информационных технологий, механики и оптики. – 2011. – С. 97-102.
8. Ковалев, Д. Информационная безопасность облачных вычислений / Д. Ковалев // T-Comm. – 2011. – № S1. – С. 14-16.

9. Денисов, Д. В. Перспективы развития облачных вычислений / Д. В. Денисов // Прикладная информатика. – 2009. – № 5(23). – С. 52-58.
10. Su, Y. ReMCA: A reconfigurable multi-core architecture for full RNS variant of BFV homomorphic evaluation / Y. Su, B.-L. Yang, C. Yang, S.-Y. Zhao // IEEE Trans. Circuits Syst. I. – 2022. – Vol. 69, no. 7. – P. 2857–2870.
11. Reagen, B. Brooks Cheetah: Optimizing and accelerating homomorphic encryption for private inference / B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, D. // Proc. 27th IEEE Int. Symp. High Perform. Comput. Archit. – 2021. – P. 26–39.
12. Al Badawi, A. Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme / A. Al Badawi, Y. Polyakov, K.M.M. Aung, B. Veeravalli, K. Rohloff, // IEEE Transactions on Emerging Topics in Computing. – 2021. – vol. 9. – P. 941–956.
13. Micciancio, D. Bootstrapping in fhew-like cryptosystems / D. Micciancio, Y. Polyakov // Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography. – 2021. – P. 17–28.
14. Бабенко, Л. К. Гибридное шифрование на основе использования симметричных и гомоморфных шифров / Л. К. Бабенко, Е. А. Толоманенко // Известия ЮФУ. Технические науки. – 2021. – № 2(219). – С. 6-18.
15. Bian, S. APAS: Application-specific accelerators for RLWE-based homomorphic linear transformations / S. Bian, D. E. S. Kundi, K. Hirozawa, W. Liu, T. Sato // IEEE Trans. Inf. Forensics Security. – 2021. – vol. 16. – P. 4663–4678.
16. Feldmann, A. F1: A fast and programmable accelerator for fully homomorphic encryption / A. Feldmann, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, D. Sanchez // Proc. 54rd Annu. IEEE/ACM Int. Symp. Microarchit. – 2021.

17. Wu, W. Secure and efficient outsourced k-means clustering using fully homomorphic encryption with ciphertext packing technique / W. Wu, J. Liu, H. Wang, J. Hao, M. Xian // IEEE Trans. Knowl. Data Eng. – 2020. – vol. 33, no. 10. – P. 3424–3437.
18. Turan, F. HEAWS: An accelerator for homomorphic encryption on the amazon AWS FPGA / F. Turan, S. S. Roy, I. Verbauwhede // IEEE Trans. Comput. – 2020. – Vol. 69, no. 8. – P. 1185–1196.
19. Mert, A. C. An extensive study of flexible design methods for the number theoretic transform / A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, A. Aysu // IEEE Trans. Comput. – 2020. – Vol. 71, no. 11. – P. 2829–2843.
20. Kim, S. FPGA based accelerators of fully pipelined modular multipliers for homomorphic encryption / S. Kim, K. Lee, W. Cho, J. H. Cheon, R. A. Rutenbar // Proc. Int. Conf. ReConFigurable Comput. FPGAs. – 2019. – P. 1–8.
21. Roy, S. S. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data / S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, I. Verbauwhede // Proc. 25th IEEE Int. Symp. High Perform. Comput. Archit. – 2019. – P. 387–398.
22. Halevi, S. An improved RNS variant of the BFV homomorphic encryption scheme / S. Halevi, Y. Polyakov, V. Shoup // Cryptographers' Track RSA Conf. – 2019. – P. 83–105.
23. Mert, A. C. Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme / A. C. Mert, E. Öztürk, E. Savaş // IEEE Trans. VLSI Syst. – 2019. – Vol. 28, no. 2. – P. 353–362.
24. Cheon, J. H. Bootstrapping for approximate homomorphic encryption / J. H. Cheon, K. Han, A. Kim, M. Kim, Y. Song // Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, 29 April–3 May 2018. – Springer: Berlin/Heidelberg, Germany. – 2018. – P. 360–384.

25. Al Badawi, A. Highperformance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA / A. Al Badawi, B. Veeravalli, C. F. Mun, K. M. M. Aung // IACR Trans. Cryptographic Hardware Embedded Syst. – 2018. – P. 70–95.
26. Roy, S. S. HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation / S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, I. Verbauwhede // IEEE Trans. Comput. – 2018. – Vol. 67, no. 11. – P. 1637–1650.
27. Chillotti, I. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE / I. Chillotti, N. Gama, M. Georgieva, M. Izabachène // Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Hong Kong, China, 3–7 December 2017. – Springer: Berlin/Heidelberg, Germany, 2017. - 2017. – P. 377–408.
28. Chillotti, I. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds / I. Chillotti, N. Gama, M. Georgieva, M. Izabachène // Proceedings of the Advances in Cryptology ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, 4–8 December 2016. – Springer: Berlin/Heidelberg, Germany. – 2016. – P. 3–33.
29. Jean-Claude, B. A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes / B. Jean-Claude, J. Eynard, A. Hasan, V. Zucca // IACR Cryptol. ePrint Arch. 2016. – 2016. – P. 423-442.
30. Бабенко, Л. К. Методы полностью гомоморфного шифрования на основе матричных полиномов / Л. К. Бабенко, Ф. Б. Буртыка, О. Б. Макаревич, А. В. Трепачева // Вопросы кибербезопасности. – 2015. – № 1(9).
31. Ducas, L. Fhew: bootstrapping homomorphic encryption in less than a second / L. Ducas, D. Micciancio // Annual International Conference on

the Theory and Applications of Cryptographic Techniques. Springer. - 2015. – P. 617–640.

32. Brakerski, Z. Efficient fully homomorphic encryption from (standard) lwe / Z. Brakerski, V. Vaikuntanathan // SIAM Journal on computing. – 2014. – vol. 43, no. 2. – P. 831–871.
33. Smart N., Vercauteren F. Fully homomorphic SIMD operations // Des. Codes Cryptogr. – Springer US, Springer Science+Business Media, 2014. – Vol. 71, Iss. 1. – P. 57–81.
34. Coron, J. Scale-Invariant Fully Homomorphic Encryption over the Integers / J. Coron, T. Lepoint, M. Tibouchi // Public-Key Cryptography – PKC 2014: 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014, Proceedings / H. Krawczyk – Springer Science+Business Media. - 2014. – P. 311-328.
35. Буртыка, Ф. Б. Пакетное симметричное полностью гомоморфное шифрование на основе матричных полиномов / Ф. Б. Буртыка // Труды Института системного программирования РАН. – 2014. – Т. 26. – № 5. – С. 99-116.
36. Brakerski, Z. Lattice-Based FHE as Secure as PKE / Z. Brakerski, V. Vaikuntanathan // Proceedings of the 5th Conference on Innovations in Theoretical Computer Science - ITCS ’14 January 12–14, 2014, Princeton, New Jersey, USA. – 2014. – P. 1-12.
37. Alperin-Sheriff, J. Faster Bootstrapping with Polynomial Error / J. Alperin-Sheriff, C. Peikert // Garay, J.A., Gennaro, R. (eds) Advances in Cryptology – CRYPTO 2014. – CRYPTO 2014. Lecture Notes in Computer Science, vol 8616. – Springer, Berlin, Heidelberg. – 2014. – P. 297-314.
38. Gentry, C. Homomorphic Encryption from Learning With Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based / C. Gentry, A. Sahai, B. Waters // Advances in cryptology – CRYPTO-2013,

33rd Annual Cryptology Conf. Santa Barbara, CA, USA. - 2013. – Part 1. – P. 73-93.

39. Cheon, J. H. Batch Fully Homomorphic Encryption over the Integers / J. H. Cheon, J. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, A. Yun // Advances in Cryptology – EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings / T. Johansson, P. Q. Nguyen – Springer Berlin Heidelberg, 2013. - 2013. – P. 315-335.
40. Coron, J. Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers / J. Coron, D. Naccache, M. Tibouchi // Advances in Cryptology – EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012, Proceedings / D. Pointcheval, T. Johansson – Springer Science+Business Media, 2012. - 2012. – P. 446-464.
41. Jain, N. Implementation and analysis of homomorphic encryption schemes / N. Jain, S. K. Pal, D. K. Upadhyay // Intern. J. on Cryptography and Information Security (IJCIS). – 2012. – Vol. 2, no. 2. – P. 27-44.
42. Brakerski, Z. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP / Z. Brakerski // CRYPTO 2012. – 2012. – P. 868–886.
43. Gentry, C. Fully Homomorphic Encryption with Polylog Overhead / C. Gentry, S. Halevi, N.P. Smart // Pointcheval, D., Johansson, T. (eds) Advances in Cryptology – EUROCRYPT 2012. - 2012. – P. 465-482.
44. Gentry, C. Homomorphic Evaluation of the AES Circuit / C. Gentry, S. Halevi, N.P. Smart // Safavi-Naini, R., Canetti, R. (eds) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science. – 2012. - Vol. 7417. – P. 850-867.

45. Brakerski, Z. Fully homomorphic encryption without bootstrapping / Z. Brakerski, C. Gentry, V. Vaikuntanathan // Proc. 3rd ITCS. – 2012. – P. 309-325.
46. Gentry, C. Fully Homomorphic Encryption without Squashing Using Depth-3 Arithmetic Circuits / C. Gentry, S. Halevi // Foundations of Computer Science (FOCS), IEEE 52nd Annual Symposium on IEEE - 2011. — P. 107–109.
47. Gentry, C. Implementing Gentry's Fully-Homomorphic Encryption Scheme / C. Gentry, S. Halevi // Advances in Cryptology — EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011, Proceedings / K. G. Paterson — Springer Science+Business Media. - 2011. — P. 129—148.
48. Coron, J. Fully Homomorphic Encryption over the Integers with Shorter Public Keys / J. Coron, A. Mandal, D. Naccache, M. Tibouchi // Advances in Cryptology – CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011, Proceedings / P. Rogaway – Springer Science+Business Media. - 2011. – P. 487-504.
49. Gentry, C. Better Bootstrapping in Fully Homomorphic Encryption / C. Gentry, S. Halevi, N. P. Smart // Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography. – 2011.
50. Stehlé, D. Faster Fully Homomorphic Encryption / D. Stehlé, R. Steinfeld // Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16. – 2010 – P. 377-394.
51. Smart, N. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes / N. Smart, F. Vercauteren // Public Key Cryptography – PKC 2010: 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010, Proceedings /

P. Q. Nguyen, D. Pointcheval – Berlin, Heidelberg, New York, NY, London [etc.]: Springer Science+Business Media. - 2010. – P. 420-443.

52. Dijk, M. Fully Homomorphic Encryption over the Integers / M. Dijk, C. Gentry, S. Halevi, V. Vaikuntanathan // Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings / H. Gilbert — Berlin: Springer Berlin Heidelberg. - 2010. – P. 24-43.
53. Gentry, C. A Fully homomorphic encryption using ideal lattices / C. Gentry // Symposium on the Theory of Computing (STOC). – Bethesda, USA, 2009. – 2009. – P. 169-178.
54. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University, Stanford (2009).
55. Гаража, А. А. Об использовании библиотек полностью гомоморфного шифрования / А. А. Гаража, И. Ю. Герасимов, М. В. Николаев, И.В. Чижов // International Journal of Open Information Technologies. – 2021. – Vol. 9, No. 3. – P. 11-22.
56. Babenko, L. K. Development of algorithms for data transmission in sensor networks based on fully homomorphic encryption using symmetric Kuznyechik algorithm / L. K. Babenko, E. A. Tolomanenko // Journal of Physics: Conference Series. – 2021. – Vol. 1812. – P. 246-251.
57. Салман, В. Д. Гомоморфная криптосистема для подсчета голосов / В. Д. Салман // Новые импульсы развития: вопросы научных исследований: сборник статей Международной научно-практической конференции, Саратов, 18 мая 2020 года. – Саратов: НОО «Цифровая наука». – 2020. – Том 1 – С. 93-99.
58. Щелкунов, А. М. Гомоморфное шифрование в базах данных / А. М. Щелкунов, М. Л. Глухарев // Интеллектуальные технологии на транспорте. – 2018. – № 2(14). – С. 47-52.

59. Martins, P. A survey on fully homomorphic encryption: An engineering perspective / P. Martins, L. Sousa, A. Mariano // ACM Computing Surveys (CSUR). – 2017. – Vol. 50, №. 6. – P. 1-33.
60. Archer, D. Applications of homomorphic encryption / D. Archer, L. Chen, C. Jung Hee, R. Gilad-Bachrach, R. A. Hallman, Z. Huang, X. Jiang // Crypto Standardization Workshop, Microsoft Research. – 2017. – vol. 14.
61. Аракелов, Г. Г. Прикладная гомоморфная криптография: примеры / Г. Г. Аракелов, А. В. Грибов, А. В. Михалев // Фундаментальная и прикладная математика. – 2016. – Т. 21, № 3. – С. 25-38.
62. Бабенко, Л. К. Защищенные вычисления и гомоморфное шифрование / Л. К. Бабенко, Ф. Б. Буртыка, О. Б. Макаревич, А. В. Трепачева // III Национальный суперкомпьютерный форум 25-27 ноября 2014, г.Переславль-Залесский. – ИПС имени А. К. Айламазяна РАН. – 2014.
63. Boneh, D. Private database queries using somewhat homomorphic encryption / D. Boneh, C. Gentry, S. Halevi, D. Wang, D. J. Wu // Applied cryptography and network security Springer. - 2013. - P. 102–118.
64. Peng, Z. Danger of using fully homomorphic encryption: A look at Microsoft SEAL / Z. Peng // ArXiv. – 2019.
65. Bogos, S. Cryptanalysis of a homomorphic encryption scheme / S. Bogos, J. Gaspoz, S. Vaudenay // Cryptogr. Commun. – 2018. – № 10. – P. 27–39.
66. Wang, B. Cryptanalysis of a Symmetric Fully Homomorphic Encryption Scheme / B. Wang, Y. Zhan, Z. Zhang // IEEE Transactions on Information Forensics and Security. – 2018. – Vol. 13, no. 6. – P. 1460-1467.
67. Babenko, L. K. Known plaintexts attack on polynomial based homomorphic encryption / L. K. Babenko, A. V. Trepacheva // Proceedings of the 7nd international conference on security of information and networks ACM. – 2014. – P. 67–70.

68. HElib: Design and implementation of a homomorphic-encryption library [Электронный ресурс]. – URL: https://github.com/homenc/HElib (дата обращения 05.09.2023).
69. TFHE. Fast Fully Homomorphic Encryption Library over the Torus [Электронный ресурс]. – URL: https://github.com/tfhe/tfhe (дата обращения 05.09.2023).
70. Microsoft SEAL [Электронный ресурс]. – URL: https://github.com/microsoft/SEAL (дата обращения 05.09.2023).
71. FHEW. A Fully Homomorphic Encryption library [Электронный ресурс]. – URL: https://github.com/lducas/FHEW (дата обращения 05.09.2023).
72. Kim, A. Revisiting homomorphic encryption schemes for finite fields / A. Kim, Y. Polyakov, V. Zucca // Advances in Cryptology. – ASIACRYPT. – 2021. – P. 608–639.
73. Babenko, M. G. Comparative Analysis of Homomorphic Encryption Algorithms Based on Learning with Errors / M. G. Babenko, E. I. Golimblevskaia, E. M. Shiriaev // Proceedings of the Institute for System Programming of the RAS – 2020. – Vol. 32, No. 2. – P. 37-52.
74. Sathya, S. S. A Review of Homomorphic Encryption Libraries for Secure Computation / S.S. Sathya, P. Vepakomma, R. Raskar, R. Ramachandra, S. Bhattacharya // ArXiv – 2018.
75. Parmar, P. V. Survey of various homomorphic encryption algorithms and schemes / P. V. Parmar, S. B. Padhar, S. N. Patel, N. I. Bhatt, R. H. Jhaveri // International Journal of Computer Applications – 2014. – Vol. 91, no. 8. – P. 26–32.
76. Яковлев М.О. Защищенный калькулятор. Разработка клиентского компонента. // Выпускная квалификационная работа бакалавра [Электронный ресурс]. – URL: http://www.nsu.ru/xmlui/bitstream/handle/nsu/471/Text_YakovlevMO.pdf (дата обращения 05.09.2023).

77. Русаловский, И. Д. Разработка методов гомоморфного деления / И. Д. Русаловский, Л. К. Бабенко, О. Б. Макаревич // Известия ЮФУ. Технические науки. – 2022. – № 4(228). – С. 103-112.
78. Бабенко, Л. К. Метод реализации гомоморфного деления / Л. К. Бабенко, И. Д. Русаловский // Известия ЮФУ. Технические науки. – 2020. – № 4(214). – С. 212-221.
79. Русаловский, И. Д. Библиотека гомоморфного шифрования Helib / И. Д. Русаловский // III Всероссийская научно-техническая конференция "Фундаментальные и прикладные аспекты компьютерных технологий и информационной безопасности" : материалы Всероссийской научно-технической конференции, Таганрог, 03-09 апреля 2017 г. : [в 2 ч.]. Ч. 1. – Ростов-на-Дону : Издательство Южного федерального университета, 2017. – С. 73-76.
80. Бабенко, Л. К. Библиотека полностью гомоморфного шифрования целых чисел / Л. К. Бабенко, И. Д. Русаловский // Известия ЮФУ. Технические науки. – 2020. – № 2(212). – С. 218-227.
81. Русаловский, И. Проблема полностью гомоморфной обработки целых чисел / И. Русаловский, Л. Бабенко // Безопасность информации и компьютерных сетей : материалы 12-й Международной научной конференции (SIN 2019), 12-15 сентября 2019 г., Сочи, Россия. – Сочи : РИЦ ФГБОУ ВО "СГУ", 2019. – С. 41-43.
82. Русаловский, И. Д. Проблема гомоморфного деления / И. Д. Русаловский // Студенческая наука для развития информационного общества : сборник материалов VII Всероссийской научно-технической конференции (г. Ставрополь, 26-28 декабря 2017 года) : [в 2 ч.]. Ч. 1. – Ставрополь : СКФУ, 2018. – С. 434-437.
83. Бабенко, Л. К. Гомоморфное шифрование. Теоретические основы. Области применения / Л. К. Бабенко, И. Д. Русаловский // Современные компьютерные технологии : сборник статей Научно-

методической конференции, Таганрог, 25-29 февраля 2020 г. – Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2020. – С. 58-65. – DOI 10.18522/mod.comp.tech.2020.1.9.

84. Rusalovskiy, I. Homomorphic operations on integers via operations on bits / I. Rusalovskiy, L. Babenko // The 15th IEEE International Conference on Security of Information and Networks (SIN 2022): Proceedings of the 15th International Conference on Security of Information and Networks, Sousse, Tunisia, November 11th. – 2022. – P. 01-04. – DOI 10.1109/SIN56466.2022.9970502.
85. Бабенко, Л. К. Побитовые гомоморфные операции над числами с плавающей точкой / Л. К. Бабенко, И. Д. Русаловский // Известия ЮФУ. Технические науки. – 2023. – № 4(234). – С. 26-34. – DOI 10.18522/2311-3103-2023-4-26-34.
86. IEEE Standard for Floating-Point Arithmetic // IEEE Std 754-2019 (Revision of IEEE 754-2008). – 22 July 2019. – P.1-84.
87. Freedman Gilad Image and Video Upscaling from Local Self-examples // ACM Trans. Graphics. — 2011. — Vol. 30, no. 2. — P. 1-11.
88. Потапов, А. А. Новейшие методы обработки изображений / А. А. Потапов, А. А. Пахомов, С. А. Никитин, Ю. В. Гуляев // — М.: Физматлит - 2008. — С. 496.
89. Бабенко, Л. К. Масштабирование цифровых изображений с применением гомоморфного шифрования / Л. К. Бабенко, И. Д. Русаловский // Вопросы кибербезопасности. – 2021. – № 3(43). – С. 2-10. – DOI 10.21681/2311-3456-2021-3-2-10.
90. Бабенко, Л. К. Гомоморфная реализация метода Гаусса / Л. К. Бабенко, И. Д. Русаловский // Вопросы кибербезопасности. – 2023. – № 4(56). – С. 33-40. – DOI 10.21681/2311-3456-2023-4-33-40.
91. Русаловский, И. Д. Гомоморфная реализация алгоритма Гаусса / И. Д. Русаловский // IV Всероссийская научно-техническая

конференция "Фундаментальные и прикладные аспекты компьютерных технологий и информационной безопасности" : материалы Всероссийской научно-практической конференции. – Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2018. – С. 364-367.

ПРИЛОЖЕНИЕ А

Свидетельство о государственной регистрации программы для ЭВМ

2020

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2020611853

Реализация программы полностью гомоморфной обработки целых чисел

Правообладатель: *федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет» (Южный федеральный университет) (RU)*

Авторы: *Бабенко Людмила Климентьевна (RU), Русаловский Илья Дмитриевич (RU)*

Заявка № 2020610864
Дата поступления 31 января 2020 г.
Дата государственной регистрации
в Реестре программ для ЭВМ *11 февраля 2020 г.*

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

ПРИЛОЖЕНИЕ Б

Программная реализация гомоморфной арифметики над целыми через операции над битами

Файл NSNumber+ConvenienceAdditions.swift

```
import Foundation

extension NSNumber {

    @objc func binaryRepresentation(with size: Int) -> [NSNumber] {
        let value = int64Value
        var string = String(value, radix: 2)
        if value < 0 {
            string = string.replacingOccurrences(of: "-", with: "")
        }
        for _ in 0..<(size - 1 - string.count) {
            string = "0" + string
        }
        var result: [UInt8] = []
        if value < 0 {
            result = string.compactMap({ UInt8(String($0)) })
            result = NSNumber.revertTwosComplement(of: result)
            result.insert(1, at: 0)
        } else {
            string = "0" + string
            result = string.compactMap({ UInt8(String($0)) })
        }
        return result.compactMap({ NSNumber(value: $0) })
    }

    @objc static func buildFromBitsArray(_ bitsArray: [NSNumber]) ->
NSNumber {
        var bits = bitsArray.map({ $0.uint8Value })
        let sign = bits.removeFirst()
        if sign == 1 {
            bits = revertTwosComplement(of: bits)
```

```
            }
            bits.reverse()
            var result: Double = 0
            for (index, bit) in bits.enumerated() {
                if bit == 1 {
                    result += pow(2, Double(index))
                }
            }
            result = sign == 1 ? -result : result
            return NSNumber(value: result)
        }

        // MARK: - Helpers

        private static func revertTwosComplement(of array: [UInt8]) ->
[UInt8] {
            // invert
            var result: [UInt8] = array.map({ $0 == 1 ? 0 : 1 })
            // add 1 to right
            result.reverse()
            var overflow: UInt8 = 1
            for i in 0...result.count-1 {
                let summ = result[i] ^ overflow
                overflow = result[i] & overflow
                result[i] = summ
            }
            result.reverse()
            return result
        }

    }
```

Файл FHEBitwiseObject.h

```
#import <Foundation/Foundation.h>
#include "helayers/hebase/hebase.h"

using namespace helayers;

NS_ASSUME_NONNULL_BEGIN
```

```
@interface FHEBitwiseObject : NSObject

- (instancetype)initWithBits:(NSArray *)bits;
- (NSArray *)decrypt;
- (std::vector<CTile>)getEncryptedBits;

- (void)diffOther:(FHEBitwiseObject *)other;
- (void)summWithOther:(FHEBitwiseObject *)other;
- (void)multiplyWithOther:(FHEBitwiseObject *)other;
- (void)divideByOther:(FHEBitwiseObject *)other;

@end

NS_ASSUME_NONNULL_END
```

Файл FHEBitwiseObject.mm

```
#import "FHEBitwiseObject.h"
#import "FHEManager.h"
#import "Bitwise_operations-Swift.h"
#include <iostream>
#include "helayers/hebase/hebase.h"
#include "helayers/hebase/helib/HelibBgvContext.h"
#include <fstream>
#include <helib/helib.h>
#include <helib/EncryptedArray.h>
#include <helib/ArgMap.h>
#include <NTL/BasicThreadPool.h>

using namespace helayers;
using namespace std;

@interface FHEBitwiseObject()
{
    vector<CTile> encryptedBits;
}

@end

@implementation FHEBitwiseObject
```

```
- (instancetype)initWithBits:(NSArray *)bits {
    self = [super init];
    [[FHEManager sharedObject] encryptNumberBits:bits
result:&encryptedBits];
    return self;
}

- (instancetype)initWithEncriptedBits:(vector<CTile>)bits {
    self = [super init];
    encryptedBits = bits;
    return self;
}

- (NSArray *)decrypt {
    return [[FHEManager sharedObject] decryptNumberBits:&encryptedBits];
}

- (vector<CTile>)getEncryptedBits {
    return encryptedBits;
}

- (void)multiplyWithBit:(CTile)bit {
    for(int i = 0; i < encryptedBits.size(); i++) {
        encryptedBits[i].multiply(bit);
    }
}

- (void)rShiftPositions:(int)positions {
    if (positions < 1) {
        // do nothing
        return;
    }
    CTile signBit = encryptedBits[0];
    for(int i = 0; i < positions; i++) {
        encryptedBits.pop_back();
        encryptedBits.insert(encryptedBits.begin(), signBit);
    }
}

- (void)lShiftPositions:(int)positions {
    if (positions < 1) {
        // do nothing
```

```
            return;
        }
        Encoder encoder(*[self getHe]);
        CTile additionalBit(*[self getHe]);
        encoder.encodeEncrypt(additionalBit, vector<int>{0});
        for(int i = 0; i < positions; i++) {
            encryptedBits.erase(encryptedBits.begin());
            encryptedBits.insert(encryptedBits.end(), additionalBit);
        }
    }

    - (void)multiplyWithOther:(FHEBitwiseObject *)other {
        Encoder encoder(*[self getHe]);
        FHEBitwiseObject *result = [[FHEBitwiseObject alloc]
initWithBits:[@0 binaryRepresentationWith:encryptedBits.size()]];
        CTile additionalBit(*[self getHe]);
        encoder.encodeEncrypt(additionalBit, vector<int>{0});
        vector<CTile> otherBits = other.getEncryptedBits;
        otherBits.insert(otherBits.end(), additionalBit);
        for(int i = int(otherBits.size()) - 2; i >= 0; i--) {
            CTile bCurrent = otherBits[i];
            CTile bNext = otherBits[i+1];
            // bi xor bi-1
            CTile coeff = bCurrent;
            coeff.add(bNext);
            // (A and (bi xor bi-1)) << i-1
            FHEBitwiseObject *partSumm = [[FHEBitwiseObject alloc]
initWithEncriptedBits:encryptedBits];
            [partSumm multiplyWithBit:coeff];
            [partSumm lShiftPositions:int(otherBits.size()) - 2 - i];
            // F = (bi xor bi-1) AND bi
            coeff.multiply(bCurrent);
            [result summOrDiffWithOther:partSumm encMode:coeff];
        }
        encryptedBits = result.getEncryptedBits;
    }

    - (void)divideByOther:(FHEBitwiseObject *)other {
        // remainder = [a0, a0, a0, ....]
        // remainder = [a0, ...., a1]
        // res = reminder - B
        // c0 = not(res0 xor b0)
```

```
            // c = c + 1
            Encoder encoder(*[self getHe]);
            CTile aSignBit = encryptedBits[0];
            vector<CTile> remainderBits(encryptedBits.size(), aSignBit);
            vector<CTile> resultBits;
            // helper bit to inverse encrypted bits
            CTile inverseBit(*[self getHe]);
            encoder.encodeEncrypt(inverseBit, vector<int>{1});

            // result sign = a0 xor b0
            CTile resultSign = aSignBit;
            resultSign.add(other.getEncryptedBits[0]);
            resultBits.push_back(resultSign);

            // result bits
            for(int i = 1; i < encryptedBits.size() + 1; i++) {
                remainderBits.erase(remainderBits.begin());
                if (i < encryptedBits.size()) {
                    remainderBits.push_back(encryptedBits[i]);
                } else {
                    CTile additionalBit(*[self getHe]);
                    encoder.encodeEncrypt(additionalBit, vector<int>{0});
                    remainderBits.push_back(additionalBit);
                }
                FHEBitwiseObject *remainder = [[FHEBitwiseObject alloc]
initWithEncriptedBits:remainderBits];
                // diff if signs are the same, sum otherwise
                // sum mode 0, diff mode 1, so just not(r0 xor bo)
                CTile modeBit = remainder.getEncryptedBits[0];
                modeBit.add(other.getEncryptedBits[0]);
                modeBit.add(inverseBit);

                [remainder summOrDiffWithOther:other encMode:modeBit];
                CTile resultBit = remainder.getEncryptedBits[0];
                resultBit.add(other.getEncryptedBits[0]);
                resultBit.add(inverseBit);
                resultBits.push_back(resultBit);
                // restore remainder if needed
                // a or b = (a and b) xor a xor b
                // (a0 xor remainder0) and old_remainderBits or not(a0 xor
remainder0) and new_remainderBits
                CTile notEqBit = encryptedBits[0];
```

```
            notEqBit.add(remainder.getEncryptedBits[0]);
            CTile eqBit = notEqBit;
            eqBit.add(inverseBit);

            vector<CTile> correctedRemainderBits;
            for(int j = 0; j < remainderBits.size(); j++) {
                CTile left = remainderBits[j];
                left.multiply(notEqBit);

                CTile right = remainder.getEncryptedBits[j];
                right.multiply(eqBit);

                CTile bit = left;
                bit.multiply(right);
                bit.add(left);
                bit.add(right);
                correctedRemainderBits.push_back(bit);
            }
            remainderBits = correctedRemainderBits;
        }
        FHEBitwiseObject *one = [[FHEBitwiseObject alloc] initWithBits:[@1
binaryRepresentationWith:resultBits.size()]];
        FHEBitwiseObject *result = [[FHEBitwiseObject alloc]
initWithEncriptedBits:resultBits];
        [result summWithOther:one];
        resultBits = result.getEncryptedBits;
        resultBits.pop_back();
        encryptedBits = resultBits;
    }

    - (void)diffOther:(FHEBitwiseObject *)other {
        [self summOrDiffWithOther:other mode:1];
    }

    - (void)summWithOther:(FHEBitwiseObject *)other {
        [self summOrDiffWithOther:other mode:0];
    }

    - (void)summOrDiffWithOther:(FHEBitwiseObject *)other mode:(int)mode {
        Encoder encoder(*[self getHe]);
        CTile f(*[self getHe]);
        encoder.encodeEncrypt(f, vector<int>{mode});
```

```
        [self summOrDiffWithOther:other encMode:f];
    }

    - (void)summOrDiffWithOther:(FHEBitwiseObject *)other encMode:(CTile)f {
        Encoder encoder(*[self getHe]);
        CTile a(*[self getHe]);
        CTile b(*[self getHe]);
        CTile p = f;

        CTile overflowBit(*[self getHe]);
        vector<CTile> result;
        for(int i = int(encryptedBits.size()) - 1; i >= 0; i--) {
            // initial setup
            a = encryptedBits[i];
            b = [other getEncryptedBits][i];
            b.add(f);

            // XOR-AND single bit adder
            CTile aXorB = a;
            aXorB.add(b);

            CTile aAndB = a;
            aAndB.multiply(b);

            CTile c = aXorB;
            c.add(p);
            // result bit
            result.push_back(c);

            aXorB.multiply(p);
            aXorB.add(aAndB);
            if (i == 0) {
                // last iteration - iteration over sign bits
                // calculate overflow bit
                // ref: https://www.geeksforgeeks.org/overflow-in-
arithmetic-addition-in-binary-number-system/
                overflowBit = p;
                overflowBit.add(aXorB);
            }
            // carry bit
            p = aXorB;
        }
```

```
        reverse(result.begin(), result.end());
        encryptedBits = result;
    }

    - (HelibBgvContext *)getHe {
        return [[FHEManager sharedObject] getContext];
    }

    @end
```

Файл FHEManager.h

```
    #import <Foundation/Foundation.h>
    #include "helayers/hebase/helib/HelibBgvContext.h"
    #include "helayers/hebase/hebase.h"

    NS_ASSUME_NONNULL_BEGIN

    @interface FHEManager : NSObject
    {
        helayers::HelibBgvContext he;
    }

    + (instancetype)sharedObject;

    - (void)encryptNumberBits:(NSArray *)bits
result:(std::vector<helayers::CTile> *)result;
    - (NSArray *)decryptNumberBits:(std::vector<helayers::CTile> *)bits;
    - (helayers::HelibBgvContext *)getContext;

    @end

    NS_ASSUME_NONNULL_END
```

Файл FHEManager.mm

```
    #import "FHEManager.h"
    #include <iostream>
    #include "helayers/hebase/hebase.h"
    #include "helayers/hebase/helib/HelibBgvContext.h"
```

```
#include <fstream>
#include <helib/helib.h>
#include <helib/EncryptedArray.h>
#include <helib/ArgMap.h>
#include <NTL/BasicThreadPool.h>

using namespace helayers;
using namespace std;

@interface FHEManager()


@end

@implementation FHEManager

// Note: These parameters have been chosen to provide fast running times
as
// opposed to a realistic security level. As well as negligible
security,
// these default parameters result in an algebra with only 10 slots
which limits
// the size of both the “keys” and “values” to 10 chars. If you try to
use
// bigger “keys” or “values” you will need to choose different
parameters
// that give you more slots, otherwise the code will throw an
// "helib::OutOfRangeError" exception.
//
// Commented below there is the parameter "m-130" which will result in
an algebra
// with 48 slots, thus allowing for “keys” and “values” up to 48 chars.

// Plaintext prime modulus
unsigned long p = 2;
// Cyclotomic polynomial - defines phi(m)
unsigned long m = 3; // this will give 1 slot
// Hensel lifting (default = 1)
unsigned long r = 1;
// Number of bits of the modulus chain
unsigned long bits = 1000;
// Number of columns of Key-Switching matrix (default = 2 or 3)
```

```
unsigned long c = 2;
// Size of NTL thread pool (default =1)
unsigned long nthreads = 12;
// debug output (default no debug output)
unsigned long debug = 1;

+ (instancetype)sharedObject {
    static id instance;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[self alloc] init];
    });
    return instance;
}

- (instancetype)init {
    self = [super init];
    if (self) {
        // set NTL Thread pool size
        if (nthreads > 1) {
            NTL::SetNumThreads(nthreads);
        }

        // To setup helib using the hebase layer, let's first
        // copy all configuration params to an HelibConfig object:
        HelibConfig conf;
        conf.p = p;
        conf.m = m;
        conf.r = r;
        conf.L = bits;
        conf.c = c;

        // Next we'll initialize a BGV scheme in helib.
        // The following two lines perform full intializiation
        // Including key generation.
        he.init(conf);

        cout << "\nNumber of slots: " << he.slotCount() << endl;
    }
    return self;
}
```

```
- (void)encryptNumberBits:(NSArray *)bits
result:(std::vector<helayers::CTile> *)result {
    Encoder encoder(he);
    for (NSNumber *bit in bits) {
        CTile encryptedBit(he);
        PTile encodedBit(he);
        encoder.encode(encodedBit, [bit intValue]);
        encoder.encrypt(encryptedBit, encodedBit);
        (*result).push_back(encryptedBit);
    }
}

- (NSArray *)decryptNumberBits:(std::vector<helayers::CTile> *)bits {
    NSMutableArray *result = [[NSMutableArray alloc] init];
    Encoder encoder(he);
    for (const auto& bit : *bits) {
        int resultBit = encoder.decryptDecodeInt(bit)[0];
        [result addObject:@(resultBit)];
    }
    return result;
}

- (helayers::HelibBgvContext *)getContext {
    return &he;
}

@end
```

Файл GaussianElimination.h

```
#ifndef GaussianElimination_h
#define GaussianElimination_h
#endif /* GaussianElimination_h */
#import <Foundation/Foundation.h>

@interface GaussianElimination: NSObject

- (NSArray<NSNumber *> *)findResultOfMatrix:(NSArray<NSArray *>
*)matrix;

@end
```

Файл GaussianElimination.mm

```
#import <Foundation/Foundation.h>
#import "GaussianElimination.h"
#import "FHEManager.h"
#import "FHEBitwiseObject.h"
#import "Bitwise_operations-Swift.h"

@implementation GaussianElimination

- (int)getBitsNumber {
    return 12;
}

- (NSArray<NSNumber *> *)findResultOfMatrix:(NSArray<NSArray *> *)matrix
{
    int bitsNumber = [self getBitsNumber];
    // helpers
    FHEBitwiseObject *encZero = [[FHEBitwiseObject alloc]
initWithBits:[@0 binaryRepresentationWith:bitsNumber]];
    // encrypt data
    NSMutableArray<NSMutableArray<FHEBitwiseObject *> *> *encryptedData
= [[NSMutableArray alloc] init];
    for (NSArray *line in matrix) {
        NSMutableArray *encLine = [[NSMutableArray alloc] init];
        for (NSNumber *element in line) {
            FHEBitwiseObject *encElement = [[FHEBitwiseObject alloc]
initWithBits:[element binaryRepresentationWith:bitsNumber]];
            [encLine addObject:encElement];
        }
        [encryptedData addObject:encLine];
    }
    // straight stroke
    for (int i = 0; i < [matrix[0] count] - 2; i++) {
        // replace lines
        for (int j = i + 1; j < [matrix count]; j++) {
            // check zero bit
            CTile eq = [encZero compareWithOther:encryptedData[i][i]];
            // replace lines if needed
            NSArray<NSArray<FHEBitwiseObject *> *> *correctedLines =
[self shiftElements:@[encryptedData[i], encryptedData[j]] basedOnBit:eq];
```

```
                encryptedData[i] = [correctedLines[0] mutableCopy];
                encryptedData[j] = [correctedLines[1] mutableCopy];
            }

            // zero bytes
            for (int j = i + 1; j < [matrix count]; j++) {
                for (int k = i + 1; k < [matrix[0] count]; k++) {
                    FHEBitwiseObject *aCopy = [[FHEBitwiseObject alloc]
initWithEncriptedBits:encryptedData[i][k].getEncryptedBits];
                    [aCopy multiplyWithOther:encryptedData[j][i]];
                    [encryptedData[j][k]
multiplyWithOther:encryptedData[i][i]];
                    [encryptedData[j][k] diffOther:aCopy];
                }
                encryptedData[j][i] = encZero;
            }
        }

        // reverse stroke
        NSMutableArray<FHEBitwiseObject *> *results = [[NSMutableArray
alloc] init];
        for (int i = int([matrix[0] count] - 2); i >= 0; i--) {
            int lastIndex = int([matrix[0] count] - 1);
            for (int j = i + 1; j < lastIndex; j++) {
                // invert index
                int resultIndex = int([matrix[0] count]) - 2 - j;
                [encryptedData[i][j]
multiplyWithOther:results[resultIndex]];
                [encryptedData[i][lastIndex] diffOther:encryptedData[i][j]];
            }
            [encryptedData[i][lastIndex] divideByOther:encryptedData[i][i]];

            [results addObject:encryptedData[i][lastIndex]];
        }
        results = [[[results reverseObjectEnumerator] allObjects]
mutableCopy];
        // decrypt results
        NSMutableArray<NSNumber *> *decResults = [[NSMutableArray alloc]
init];
        for (FHEBitwiseObject *result in results) {
            NSNumber *decResult = [NSNumber buildFromBitsArray:[result
decrypt]];
```

```
            [decResults addObject:decResult];
        }
        NSLog(@"Result is: %@", decResults);
        return decResults;
    }

    - (NSArray<NSArray<FHEBitwiseObject *> *>
*)shiftElements:(NSArray<NSArray<FHEBitwiseObject *> *> *)array
basedOnBit:(CTile)bit {
        CTile useA = bit;
        CTile useB = bit;
        [[FHEManager sharedObject] invertBit:useB];

        NSMutableArray<FHEBitwiseObject *> *resultA = [[NSMutableArray
alloc] init];
        NSMutableArray<FHEBitwiseObject *> *resultB = [[NSMutableArray
alloc] init];
        for (int i = 0; i < array[0].count; i++) {
            // A AND bit
            FHEBitwiseObject *aAndBit = [[FHEBitwiseObject alloc]
initWithEncriptedBits: array[0][i].getEncryptedBits];
            [aAndBit multiplyWithBit:useA];
            // A AND not_bit
            FHEBitwiseObject *aAndNotBit = [[FHEBitwiseObject alloc]
initWithEncriptedBits: array[0][i].getEncryptedBits];
            [aAndNotBit multiplyWithBit:useB];
            // B AND not_bit
            FHEBitwiseObject *bAndNotBit = [[FHEBitwiseObject alloc]
initWithEncriptedBits: array[0][i].getEncryptedBits];
            [aAndNotBit multiplyWithBit:useB];
            // B AND bit
            FHEBitwiseObject *bAndBit = [[FHEBitwiseObject alloc]
initWithEncriptedBits: array[1][i].getEncryptedBits];
            [aAndBit multiplyWithBit:useA];
            // (A AND bit) OR (B AND not_bit)
            [aAndBit summWithOther:bAndNotBit];
            // (A AND not_bit) OR (B AND bit)
            [aAndNotBit summWithOther:bAndBit];
            [resultA addObject:aAndBit];
            [resultB addObject:aAndNotBit];
        }
        return @[resultA, resultB];
```

```
    }

    - (void)printCurrentMatrix:(NSArray<NSArray<FHEBitwiseObject *> *>
*)encryptedData {
        NSMutableArray *testArray = [[NSMutableArray alloc] init];
        for (NSMutableArray *line in encryptedData) {
            NSMutableArray *lineArray = [[NSMutableArray alloc] init];
            for (FHEBitwiseObject *object in line) {
                NSNumber *decObject = [NSNumber buildFromBitsArray:[object
decrypt]];
                [lineArray addObject:decObject];
            }
            [testArray addObject:lineArray];
        }
        NSLog(@"Straight stroke result: %@", testArray);
    }

    @end
```

ПРИЛОЖЕНИЕ В.
АКТЫ ОБ ИСПОЛЬЗОВАНИИ РЕЗУЛЬТАТОВ ДИССЕРТАЦИИ

«УТВЕРЖДАЮ»
Директор Института
компьютерных технологий и
информационной безопасности
_____ Г.Е. Веселов
«10» _____ 2024 г.

А К Т

об использовании результатов

кандидатской диссертации Русаловского Ильи Дмитриевича на тему: «Разработка методов и средств реализации алгоритмов гомоморфного шифрования» в гранте РФФИ № 20-37-90140/20

Настоящим актом подтверждается использование в гранте РФФИ № 20-37-90140/20 на тему «Разработка методов и средств гомоморфного шифрования для облачных сервисов» результатов диссертационной работы Русаловского Ильи Дмитриевича:

1. метод, позволяющий выполнять гомоморфное деление на базе любого полностью гомоморфного алгоритма над целыми числами
2. алгоритм гомоморфного сравнения чисел при их побитном гомоморфном шифровании
3. гомоморфная реализация алгоритмов побитовых целочисленных операций сложения, разности, умножения и деления, которые могут быть выполнены на основе любого полностью гомоморфного алгоритма шифрования над битами

Руководитель гранта РФФИ № 20-37-90140/20, д.т.н., профессор — Л.К. Бабенко

«УТВЕРЖДАЮ»
Директор Института
компьютерных технологий и
информационной безопасности
_____ Г.Е. Веселов
«__» _____ 2024 г.

А К Т

об использовании результатов

кандидатской диссертации Русаловского Ильи Дмитриевича на тему: «Разработка методов и средств реализации алгоритмов гомоморфного шифрования» в учебном процессе

Настоящим актом подтверждается внедрение в учебный процесс научных и практических результатов диссертационной работы Русаловского Ильи Дмитриевича, полученных в области гомоморфной криптографии:

1. гомоморфная реализация алгоритмов побитовых целочисленных операций сложения, разности, умножения и деления, которые могут быть выполнены на основе любого полностью гомоморфного алгоритма шифрования над битами
2. гомоморфная реализация алгоритмов побитовых операций сложения, разности, умножения и деления над числами в формате с плавающей точкой, которые могут быть выполнены на основе любого полностью гомоморфного алгоритма шифрования над битами
3. программная реализация разработанных алгоритмов на основе криптографической библиотеки HElib

Результаты представленной диссертации использованы при подготовке студентов по специальности 10.05.03 «Информационная безопасность автоматизированных систем» в дисциплине «Криптографические методы защиты информации».

Заведующий кафедрой БИТ им. О.Б. Макаревича, к.т.н., доцент — Е.С. Абрамов

ООО «АйвиАппс»
347924, Ростовская область, г.о. город Таганрог, г Таганрог, ул Москатова, зд. 31-2 , этаж 4, ком. 12
ИНН 6154147318 КПП 615401001 ОГРН 1176196005468
расчетный счет: 40702810102500003734, ООО "Банк Точка"
БИК 044525104 К/с 30101810745374525104

15 января 2024 г.

АКТ

об использовании результатов

кандидатской диссертации Русаловского Ильи Дмитриевича на тему «Разработка методов и средств реализации алгоритмов гомоморфного шифрования» в коммерческом проекте компании ООО «АйвиАппс».

Настоящим актом подтверждается, что результаты кандидатской диссертации Русаловского Ильи Дмитриевича «Разработка методов и средств реализации алгоритмов гомоморфного шифрования» были использованы в процессе работы над одним из коммерческих проектов компании ООО «АйвиАппс».

Русаловским И.Д. была разработана и реализована архитектура программного комплекса, реализующего гомоморфную арифметику, описаны уровни системы. Отличительной особенностью разработанного программного комплекса является возможность выполнять как арифметические, так и логические операции над гомоморфно зашифрованными данными. Это позволяет выполнить гомоморфную реализацию практически любого прикладного алгоритма обработки данных, что можно использовать для построения сервиса защищенных облачных вычислений.

Директор ООО «АйвиАппс»

_____ В.М. Тимофеев