Федеральное государственное автономное образовательное учреждение
высшего образования
**«Московский физико-технический институт
(национальный исследовательский университет)»**

Физтех-школа прикладной математики и информатики

Кафедра системного программирования

Исследование FHE-based методов распознавания шаблонов

Выпускная квалификационная работа

Научный руководитель _______________ А.В. Шокуров

Обучающийся _______________ И.В. Тунёв

Научный консультант _______________ С.А. Фомин

Москва 2022

Аннотация

Исследование FHE-based методов распознавания шаблонов

Тунёв Иван Валерьевич

Работа посвящена исследованию и реализации схем полностью гомоморфного шифрования в разрезе поиска шаблонов в тексте и распознавания регулярных языков при помощи регулярных выражений и конечных автоматов. В [HAO16] приведена первая полностью гомоморфная схема шифрования матриц и векторов. Конечный автомат можно представить в виде набора матриц, а его работу — в виде последовательного перемножения матриц на вектор, на основании чего в [Gen+19] предъявлен алгоритм гомоморфного по матрично-векторному произведению зашифрованной работы НКА и поиска по регулярному выражению. Последняя статья выбрана в качестве основной для изучения и реализации представленного в ней алгоритма.

Оглавление

Введение **5**
Актуальность темы . . . 5
Степень разработанности темы . . . 8
Цель и научно-практическая ценность работы . . . 8
Структура и объём диплома . . . 9

1 Обзор существующих работ по теме **11**
1.1 Обзор статей . . . 11

2 Разбор ключевого алгоритма **13**
2.1 От регулярных выражений к матричной арифметике недетерминированных конечных автоматов . . . 13
2.2 Основная схема шифрования . . . 17
2.2.1 Основные параметры . . . 17
2.2.2 Шифрование векторов . . . 19
2.2.3 Шифрование матриц . . . 20
2.2.4 Гомоморфность . . . 21
2.3 Криптостойкость схемы . . . 22
2.4 Реализация схемы GGHLM . . . 25
2.5 Работа зашифрованного НКА-автомата . . . 29

3 Приложения **39**
3.1 Схема гомоморфного шифрования Gentry-Sahai-Waters . . . 39

Заключение **47**

Списки иллюстраций **47**

Список алгоритмов **49**

Введение

В дипломной работе мы рассматриваем нетривиальные модели гомоморфного шифрования некоторых классов алгоритмов — регулярных выражений и конечных автоматов, которых в результате шифрования можно применять на «недоверенной» стороне, без риска раскрыть секретный алгоритм.

Актуальность темы

Наступила эпоха коммуникации и обмена данными, требующая безопасных и конфиденциальных операций.

Криптография — именно та наука, которая занимается обеспечением и анализом безопасности санкционированного доступа к информации.

А облачные вычисления являются одним из новых приложений, где криптография, должна раскрыть свои возможности обеспечения информационной безопасности.

В частности, полностью гомоморфическое шифрование является одной из важных основ для использования всех преимуществ облачных вычислений.

Облачные вычисления — это современная перспективная инновация, обеспечивающая публичные платформы для хранения больших объемов данных.

Несмотря на удобство использования общих ресурсов, потенциальную угрозу таят в себе различные проблемы безопасности, возникающие из-за хранения критически важных данных в облаке.

Проблемы конфиденциальности хранения данных в облаке обычно ре-

шают путем шифрования непосредственно перед загрузкой. Однако для каждой отдельной обработки зашифрованных данных, данные должны быть загружены и расшифрованы на стороне клиента, а после обработки данные должны быть дополнительно зашифрованы и загружены в облако.

Эта очевидная необходимость в повторной дешифровке-расшифровке увеличивает сложность обработки и перевешивает преимущество использования облачных ресурсов.

При этом масштабируемая вычислительная мощность облака не используется, а информация, хотя и зашифрованная, постоянно подвергается воздействию противника на клиенте.

Концепция доверенных платформ, таких как «ARM TrustZone» или «Intel SGX», должны обеспечить изолированную безопасную среду выполнения, однако по факту реализуют небольшой спект доверенных приложений, и могут защитить только от очень ограниченного подмножества возможных атак.

И только схема гомоморфного шифрования предоставляет механизм для выполнения операции над зашифрованными данными без их расшифровки. Хотя идея шифрования сохраняющего операции естественна, и как идея, озвучивалась еще в 1978 году (см. [RAD78]), только спустя более 30 лет, в работе [Dij+10] была опровергнута гипотеза о невозможности такого шифрования, а сейчас, есть даже программные библиотеки, реализующие некоторые схемы ГШ (См. алг. 1).

Более того, *полностью* гомоморфное шифрование позволяет выполнять произвольные операции над зашифрованными данными, поэтому оно считается святым Граалем криптографии. Обычно, в качестве таких операций приводят какой-нибудь статистический анализ, получение агрегированных данных над зашифрованными медицинскими данными.

Да, медицинская информация очень конфиденциальна, для каждого отдельного пациента, но различные агрегации полезны для обмена между между страховыми компаниями, больницами, исследовательскими организациями. Особенно для новых, прорывных разработок — например, геномных исследований. Напомним, что ДНК и РНК человека являются биометрическими идентификаторами, такими как отпечатки пальцев, и могут раскрыть важную информацию, такую как риск заболевания (наличие аллеля болезни Альцгеймера) или такие вещи, как факт наличия или отсут-

ствия отцовства.

И конечно единый репозиторий с открытой геномной информацией приведет к множеству рисков, а если научится использовать ПГШ для анализа на зашифрованных геномных данных, то можно всего этого избежать.

Но ПГШ анализ полезен и вне медицины — в промышленности и государственном регулировании различного рода. Например, получение агрегированных данных для государственного мониторинга по зашифрованным данным из множества возможно конкурирующих энергосетей. Или мониторинг любой другой активности в промышленности, с кучей датчиков-контроллеров, данные из которых нельзя разглашать из-за опасности таргетированной атаки хакеров. Или маркетинговый анализ на базе пользовательского поведения или истории покупок.

Да и в целом, во всех областях сейчас активно используется различная прогнозная аналитика — прогнозирования поведения и будущих тенденций с использованием новых и исторических данных, а исходные данные, с одной стороны, желательно интегрировать в одном хранилище, с другой — из-за полной или частичной конфиденциальности, надо держать их зашифрованными

Об этом направлении ПГШ мы делали статьи и доклады (см. [Дан21], [Ири20]).

Другое направление ПГШ — запуск некоторых алгоритмов с встроенными секретными данными, на уязвимой для утечек, территории.

Например, это может быть алгоритм поиска вирусов в данных, который отгружается многим потребителям, где он должен искать вирусы на их территории, но при этом, если он попадет в руки хакеров, они не смогли понять, по какой «сигнатуре» вычисляется их творение.

Или алгоритм проверки разрешенных или запрещенных интернет-ресурсов, который отгружается интернет-провайдерам в виде «черного ящика», из которого нельзя заранее извлечь этот список сигнатур-алгоритмов, по которым определяется белый или черный список.

В большинстве случаев, сигнатуры вирусов или запрещенных интернет ресурсов можно представить регулярными выражениями.

Таким образом, эти классы задач часто можно свести к задаче поиска в некотором открытом тексте зашифрованных регулярных выражений.

В целом, в более широком контексте, эта тема — сделать выполняе-

мую программу, которую можно запускать, но из которой нельзя извлечь никакой информации, называется «обфускацией программ».

И после статьи [Bar+01] о невозможности обфускации в общем случае для любых произвольных программ, возможность обфускации определенных частных случаев (регулярных выражений, конечных автоматов), безусловно представляет большой интерес.

Степень разработанности темы

Начиная с 2010 года, с работы [Dij+10] появилась относительно практичная схема гомоморфного шифрования целых чисел, которую уже можно пытаться использовать на практике, и стали появлятся программные библиотеки для вычислительных экспериментов (См. алг. 1).

Однако, лишь в последние годы, с 2019 года (см. [Gen+19]), сделан теоретический подход к гомоморфному шифрованию конечных автоматов, и эта тема еще далека от наличия доступных реализаций на языках программирования и исследований сложности, надежности и других вычислительных свойств.

Цель и научно-практическая ценность работы

Цель настоящей работы — проанализировать последние достижения по этой тематике и реализовать работающие алгоритмы, решающие эту задачу.

Именно реализовать, ведь часто при решении некоторых прикладных задач встречаются статьи по алгоритмам только с псевдокодом, в котором, после настоящей реализации находят ошибки, или просто не получается воспроизвести заявленные свойства.

С другой стороны, целью является максимально компактная реализация алгоритмов, и сопровождающих материалов (слайдов, визуализации структур и потоков), чтобы можно было получить обучающие материалы для близких по теме курсов, читаемых в ИСПРАН (см. [Шок07])

Для этого, мы использовали

- Python, как язык максимально компактного представления алгоритмов;
- перенесенные в Python библиотеки из системы компьютерной алгебры Sage (См. [SJ05]);
- ряд других Python-библиотек вычислительной математики, работы с регулярными выражениями, конечными автоматами, гомоморфным шифрованием, некоторые из которых потребовали доработки;
- разработанную в отделе систему наглядного представления алгоритмов в LaTeX материалах (слайдах, статьях и книгах).

Структура и объём диплома

Этот диплом состоит из введения, трех глав, заключения и списка использованных источников, алгоритмов, и рисунков.

Общий объем 52 страниц.

Содержит рисунков — 6, алгоритмов — 19.

Алгоритм 1 Использование гомоморфного шифрования с целочисленной арифметикой на Python

```
def demo_homomorphic_encription():
    #  Инициализируем HE-объект Pyfhel
    HE = Pyfhel()
    #  Контекст для заданного p
    HE.contextGen(p=65537)
    HE.keyGen()

    integer1 = 47
    integer2 = -2
    ctxt1 = HE.encryptInt(integer1)
    ctxt2 = HE.encryptInt(integer2)

    print('ctxt1=' + str(ctxt1))
    print('ctxt2=' + str(ctxt2))

    # Можно выполнять на недоверенной территории
    ctxtSum = ctxt1 + ctxt2
    ctxtSub = ctxt1 - ctxt2
    ctxtMul = ctxt1 * ctxt2

    print('Без расшифровки результаты непонятны:')
    print('ctxtSum=' + str(ctxtSum))
    print('ctxtSub=' + str(ctxtSub))
    print('ctxtMul=' + str(ctxtMul))


    resSum = HE.decryptInt(ctxtSum)
    resSub = HE.decryptInt(ctxtSub)
    resMul = HE.decryptInt(ctxtMul)

    print('И только мы, расшифровав, сможем узнать:')
    print('resSum=' + str(resSum))
    print('resSub=' + str(resSub))
    print('resMul=' + str(resMul))
```

```
ctxt1=<Pyfhel Ciphertext at 0x7fc05ee0e840, encoding=INTEGER, size=2/2, noiseBudget=27>
ctxt2=<Pyfhel Ciphertext at 0x7fc0547d97c0, encoding=INTEGER, size=2/2, noiseBudget=27>

Без расшифровки результаты непонятны:
ctxtSum=<Pyfhel Ciphertext at 0x7fc05386d880, encoding=INTEGER, size=2/2, noiseBudget=27>
ctxtSub=<Pyfhel Ciphertext at 0x7fc05386d740, encoding=INTEGER, size=2/2, noiseBudget=27>
ctxtMul=<Pyfhel Ciphertext at 0x7fc05f1ee440, encoding=INTEGER, size=3/3, noiseBudget=1>

И только мы, расшифровав, сможем узнать:
resSum=45
resSub=49
resMul=-94
```

Глава 1

Обзор существующих работ по теме

1.1 Обзор статей

В статье [HAO16] приводится *первая* полностью гомоморфная схема, реализующая матрично-векторное умножение и сложение. Предъявленная схема является естественным расширением packed FHE[1].

Идеи этой работы используются в том числе в [GZ20].

В статье [Gen+19] приводится описание GSW-образной (см. раздел 3.1) схемы, реализующей гомоморфное по матрично-векторному произведению шифрование недетерминированных конечных автоматов, построенных по заданному регулярному выражению. По утверждению самих авторов схема демонстрирует более высокую эффективность по сравнению с аналогами (например, [HAO16] и модификации).

Автоматы рассматриваются как набор матриц переходов по каждому символу данного алфавита, состояния автомата — векторами, работа же автомата на поданном входе представляет из себя последовательное перемножение матриц перехода на вектор, кодирующий состояния автомата. Стойкость криптосистемы выводится из задач обучения с ошибками и

[1]Packed ciphertext — шифротекст вектора, образованного элементами пространства возможных открытых текстов

некоторых задач на решётках («iNTRU», см. [Шок07]).

Помимо процесса шифрования НКА авторами разбиралась задача оптимизации построения автомата по заданному регулярному выражению, т.е. задача минимизации числа состояний. Данную задачу предлагалось решать при помощи производной Антимирова (см. [Ant96]), которая в свою очередь является расширением производной Брозовского[2] (см. [ORT]), однако в данной работе мы использовали более простой алгоритм из классического учебника [HMU07].

В статье [GZ20] приводится схема *virtual black box*[3] обфускации детерминированного конечного автомата, использующая матричное представление автомата и ноль-тестирование (*limited zero-one test*[4]).

[2]Производной Брзозовского (*Brzozowski derivative*) некоторого множества строк S по строке u называется множество строк $u^{-1}S$, которое получается из S отсечением префикса, совпадающего со строкой u, у всех элементов S.

[3]Virtual Black Box обфускация — это когда обфускированный код не раскрывает никаких секретных сведений об исходной программе за исключением тех, которые можно получить из обращения к программе через чёрный ящик

[4]Ноль-тестирование — проверка того, что заданный шифротекст есть результат шифрования нуля

Глава 2

Разбор ключевого алгоритма

2.1 От регулярных выражений к матричной арифметике недетерминированных конечных автоматов

Определение 2.1.1. ***Регулярное выражение*** *(regular expression) в алфавите Σ и задаваемое им множество допустимых слов (язык) в этом же алфавите определяются рекурсивно следующим образом:*

- $\emptyset$ *— регулярное выражение, обозначающее пустое множество слов;*
- ε *— регулярное выражение, задающее пустую строку (пустое слово);*
- *Пусть $\sigma \in \Sigma$ — символ из алфавита, тогда σ — регулярное выражение, задающее множество, состоящее из этого символа $\{\sigma\}$;*
- *Пусть p, q — регулярные выражения, задающие языки P и Q соответственно. Тогда*
  - $p|q$ *— регулярное выражение, задающее $P \cup Q$;*
  - pq *— регулярное выражение, задающее конкатенацию языков P и Q;*

   - p^* — *регулярное выражение, задающее* P^* *(звезда Клини);*

- *Других регулярных выражений в алфавите* Σ *нет;*

Т.о. регулярное выражение представляет из себя строку-образец (англ. pattern, по-русски её часто называют «шаблоном», «маской»), состоящую из символов и метасимволов и задающая правило поиска в тексте.

Стандартной реализацией регулярного выражения, как известно, являются недетерминированные конечные автоматы (НКА, см. [HMU07]), представляющие из себя набор состояний (часть из которых помечена и называется стартовыми, часть —финишными, конечными, терминальными, финальными) и правил перехода из состояния в состояние.

НКА принимает на вход слово и посимвольно считывает его и в зависимости от обработанного символа переходит в другое состояние (возможны также переходы по пустой строке, они же ϵ-переходы, т.е. такие, для которых необязательно прочесть какой-либо символ на данном такте).

Наиболее естественным представлением НКА является ориентированный граф, вершинами которого являются состояния автомата, размеченные в соответствии со своей ролью (старт-финиш или отсутствие метки), в котором пара (i, j) соединена ребром, помеченным символом σ (или ϵ) в том и только том случае, если существует переход из состояния i в состояние j по данному символу или пустой строке (см. рис 2.1)).

В силу недетерминированности прочтению поданной на вход строки (слова) может соответствовать множество маршрутов в графе НКА.

Строка (слово) принимается данным НКА (т.е. соответствует заданному регулярному выражению), если во множестве этих маршрутов содержится путь заканчивающийся финальным состоянием.

В решении поставленной задачи мы будем сводить работу НКА к матрично-векторному произведению, что накладывает дополнительное ограничение: использовать НКА без ϵ-переходов. Данное ограничение несущественно для нас, поскольку любой НКА с ϵ-переходами можно преобразовать в эквивалентный ему, таких переходов не содержащий (см. рис 2.1, мы использовали классический алгоритм с 77 страницы учебника [HMU07]).

Перейдём к матрично-векторному представлению НКА без ϵ-переходов.

Алгоритм 2 Получение матриц переходов для каждого символа алфавита НКА

```
def nfa2_zzq_matrices(nfa, N=0, ZZq=ZZ):
    '''
    Генерит матрицы переходов
    * для автомата ‘nfa‘
    * в заданном кольце ‘ZZq‘
    * выровненную до квадрата N
    '''
    N = max(N, len(nfa.states))
    state2index = {}

    for i, state in enumerate(nfa.states):
        state2index[state] = i

    mats = {}
    symbol2index = {}
    for idx, symbol in enumerate(nfa.symbols):
        symbol2index[symbol] = idx
        mats[idx] = zero_matrix(ZZq, N, N)

    for e_state, symbol, next_state in nfa._transition_function.get_edges():
        i = state2index[e_state]
        j = state2index[next_state]
        mats[symbol2index[symbol]][j, i] = 1

    return mats, state2index, symbol2index
```

Пусть дан НКА с r состояниями. Для каждого символа σ из алфавита Σ можно построить матрицу переходов M_σ размера $r \times r$ по этому символу в заданном НКА.

Полагая состояния занумерованными от 1 до r, наделим M_σ следующим свойством: её (i, j) элемент равен 1, если в НКА существует переход из j в i состояние по символу σ, и нулю в противном случае (см. алгоритм 2).

Состояния автомата же будем представлять вектором v из r элементов, i позиция которого равна 1, если состояние стартовое и нулю иначе. Пусть автомату подаётся на вход строка $w = w_1 ... w_k, w_i \in \Sigma$. Тогда работа автомата на поданном входе описывается следующим выражением:

$$(\prod M_{w_i}{}_{i=1}^{k}) \cdot v = (M_{w_k} \dots M_{w_1})v$$

результат которого будет содержать себе накопленную информацию о переходах во время работы, анализ которой и сообщит нам, соответствует ли поданная на вход строка требуемому шаблону (см. алг. 3).

Алгоритм 3 НКА через матрицу переходов

```
@dataclass
class RegexByNFA:
    regex_str: str
    ring = ZZ
    N = 0

    def __post_init__(self):
        self.regex = Regex(self.regex_str)
        self.enfa = self.regex.to_epsilon_nfa()
        self.nfa = self.enfa.remove_epsilon_transitions()

        self.N = max(self.N, len(self.nfa.states))

        (self.mats, self.state2index,
            self.symbol2index) = nfa2_zzq_matrices(self.nfa, self.N, self.ring)

    def set_start_state(self):
        self.state = zero_matrix(self.ring, self.N, 1)
        for state in self.nfa.start_states:
            self.state[self.state2index[state]] = 1

    def eval_symbol_by_nfa(self, idx):
        printv(self.mats[idx])
        self.state = self.mats[idx] * self.state
        pass

    def open_match_text(self, text):
        print(f'Матчинг шаблона «{self.regex_str}» в «{text}»\n\n')
        self.set_start_state()

        for chr_ in text:
            idx = self.symbol2index[chr_]
            self.eval_symbol_by_nfa(idx)

        for state in self.nfa.final_states:
            if self.state[self.state2index[state]] > 0:
                return True
        return False

def demo_nfa_by_matrix():
    r = RegexByNFA('(a*|b)')
    print(r.open_match_text('aab'))
```

Матчинг шаблона «(a*|b)» в «aab»

$$\begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&0&1&0&1&1&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 3 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&0&1&0&1&1&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 3 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 3 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&1&0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 3 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

False

Нашей дальнейшей целью является разработка и реализация схемы шифрования, гомоморфного по матрично-векторному произведению.

2.2 Основная схема шифрования

Ниже приводится схема, осуществляющая шифрование матриц, векторов и демонстрирующая зашифрованную работу НКА.

Наперёд отметим, что данная схема показывает свою эффективность в сравнении с другими (например, предъявленной в [HAO16] и её модификациями). Эта криптосистема похожа на схему из статьи [Gen+19].

Нами же будет реализована модификация данной схемы, использующая методы перешифровки, описанные в [AP14], и опирающаяся на предположение о Circular Security

Замечание 2.2.1. лучше ещё раз разобраться с CIRCULAR SECURITY, как это по-русски

Определение 2.2.1. *Говорят, что криптосистема удовлетворяет circular security, если имея шифротекст самого ключа шифрования мы (как противник) не можем этот же ключ шифрования. Это понятие остылает нас к задаче KDMS (Key-dependent message security, являясь примитивнейшей формой такой безопасности.*

один ключ вместо многих, процедура перешифровки убирают накопленный шум –> в ключе никакого шума не будет

Circular security это циклическое перешифрование одного ключа по другому. (1 -> 2 ->... n –> 1), т.е. стойкость к шифрованию на самом себе (через некоторое время)

Введём ещё одно важнейшее понятие из теории решёток:

Определение 2.2.2. *Гаджет-матрией обычно называют матрицу следующего вида:* $G = I \bigotimes g$, *где* I *— единичная матрица, а вектор* $g = (1, 2, ..., 2^{\log q - 1})$ *можно трактовать как оператор обратный битовой декомпозиции. Секрет (он же потайной вход, лазейка, trapdoor) для матрицы* G *является известным.*

2.2.1 Основные параметры

Дано:

- параметры (n, m, q), которые будут доопределены позднее

- χ — распределение специального шума над $\mathbb{Z}_q$, вероятности которого сконцентрированы на значениях, много меньших чем q (в итоге работаем со специальными 0-1 матрицами);

- Недетерминированный конечный автомат с $r \leq n$ числом состояний, представленный своими матрицами перехода $M_\sigma, \sigma \in \Sigma$. В случае если переходы автомата определены не по всем символам, дополним автомат (проинициализируем матрицы перехода по отсутствующим символам нулевыми);

Шифрование матриц и векторов включает в себя добавление шума для обеспечения семантической стойкости.

Определение 2.2.3. *Криптосистема называется семантически стойкой, если по шифротексту невозможно получить значимую информацию об исходном открытом тексте.*

Генерация ключа Секретный ключ представляет из себя пару матриц $S \sim \chi^{n\times n}, E \sim \chi^{n\times m}$. От S требуется обратимость (необратима с вероятностью $\sim \frac{1}{q}$), E участвует в случайной выборке шума (см. алг. 5).

def/trapdoor

Замечание 2.2.2. гаджет-матрица — матрица с известным секретом (англ. trapdoor), эта матрица не квадратная

Замечание 2.2.3. Блин, у нас же не так! Матрица шума НЕ ЧАСТЬ ключа, а выбирается для каждой операции шифрования (и не используется для дешифрования, не помним о ней нигде).
Семплирование у нас — это рандомная 0-1 матрица, генеримая алгоритмом 4 — используется и для генерации ключа, и в каждом шифровании для спецшума.
Circular стойкость: один ключ вместо многих, процедура перешифровки убирают накопленный шум –> в ключе никакого шума не будет
Circular security это циклическое перешифрование одного ключа по другому. (1 -> 2 ->... n –> 1), т.е. стойкость к шифрованию на самом себе (через некоторое время)

Определение 2.2.4. *Circular security — это стойкость к шифрованию ключа на самом себе.*

Обозначение: $sk := (S, E)$

Сэмплирование шума Осуществляется процедурой $noise_samp(SK, v)$, возвращающей вектор $e = E \times r \mod q$, где $r = G^{-1}v$, здесь v обозначает начальный вектор состояний НКА.

Замечание 2.2.4. Вообще нет такого, нет гаджет-матриц.

Basic transformation Опишем процедуру, которая будет вызываться при шифровании матриц и векторов. При заданном ключе SK генерируется шум $e \leftarrow NoiseSamp(SK, v)$ подаётся на выход

$$c = Enc^*_{sk}(v) = S^{-1}(v + e)$$

откуда видно, что $c \approx S^{-1}v$ в силу малости нормы вектора шума e.

2.2.2 Шифрование векторов

Шифрование векторов базируется на Enc^*_{SK}. Чтобы эта процедура была функцией шифрования, нам, как и в схеме шифрования Регева [O. Regev.

On lattices, learning with errors, random linear codes, and cryptography. J. ACM, 56(6), 2009.] необходимо подавать шифруемый вектор, предварительно умноженный на достаточно большой скаляр β, который в свою очередь будет выступать в роли верхней границы шума (т.е. вероятность $\mathbb{P}(||e|| \leq \beta)$ должна быть достаточно близка к единице).

Замечание 2.2.5. библиография оформить Регева [O. Regev. On lattices, learning with errors, random linear codes, and cryptography. J. ACM, 56(6), 2009.]

Ограничимся ещё сильнее: пусть b будет верхей границей l_∞ нормы векторов, с которыми мы можем столкнуться во время работы, при этом $b << q$, а β положим равным $\lfloor \frac{q}{b} \rfloor$. В реализации же параметр β выбирается достаточно большим и независимым от b, q.

Само же шифрование и дешифрование вектора $v \in \mathbb{Z}_q^n$ выглядит следующим образом:

$$c = VecEnc(v) = Enc^*_{SK}(\beta v) = S^{-1}(\beta v + e)$$

Замечание 2.2.6. Нужен переход с остатками, в явном виде доказать.

$$VecDec(c) = \lfloor \frac{Sc}{\beta} \rceil = \lfloor \frac{bSc}{q} \rceil$$

где $\lfloor \dots \rceil$ обозначает округление в сторону ближайшего целого.

Корректность такого округления, очевидно, вытекает из ограничений на норму шума и по построению параметра β.

Подробную реализацию алгоритма, см. алгоритм 6.

Замечание 2.2.7. ДОБАВИТЬ ДОКАЗАТЕЛЬСТВО СТОЙКОСТИ!!!

2.2.3 Шифрование матриц

Идейно всё остаётся тем же самым, только в данном случае будет применяться также техника GSW-кодирования, использующая в себе гаджет матрицу G. В реализованной нами модификации гаджет матрица будет использоваться неявно, что будет оговорено дополнительно.

Напомним, что традиционно гаджет-матрицу определяют как:

$$G = I \otimes g$$

где I — единичная матрица, $g = (1, 2, ..., 2^{\log q - 1})$.

Исходное пространство открытых текстов в этом случае состоит из квадратных матриц $M \in \mathbb{Z}_q^{n \times n}$, т.о. каждую матрицу можно представить в виде набора n вектор-столбцов из пространства $\mathbb{Z}_q^n$: $M = [m_1, ..., m_n]$. Каждой из матриц сопоставляется она, умноженная на гаджет: $\hat{M} = MG = [\hat{m_1} ... \hat{m_m}]$, потом каждый из этих столбцов шифруется и составляется матрица из зашифрованных столбцов выступаюшая в роли шифротекста, что, на самом деле эквивалентно следующему представлению:

$$C = S^{-1}(MG + E')$$

где E' — матрица-шум.

Матрицы в дешифровании не нуждаются, поскольку они сами по себе не содержат и не сохраняют никакой информации о прочитанном входе. Тем не менее, сконструировать такую процедуру труда не составит.

2.2.4 Гомоморфность

Определение 2.2.5. ***Полностью гомоморфное шифрование*** *англ. Fully Homotorphic Encryption — шифрование, гомоморфное относительно любых операций над открытыми текстами.*

Функция шифрования $E(k, m)$, где m — это открытый текст, k — ключ, гомоморфна относительно операции $*$, если для любого ключа и открытых текстов m_1, m_2 существует эффективно вычислимая функция M, которая, принимая на вход пару $E(k, m_1), E(k, m_2)$, вернёт криптограмму c такую, что результатом её дешифрования будет $m_1 * m_2$.

Аналогично вводится понятие гомоморфности относительно операции $+$.

Пусть нам потребовалось *зашифрованно* перемножить k матриц на вектор v:

$$(M_k \cdot ... \cdot M_1)v$$

Тогда нам нужно сгенерировать $k+1$ ключей: $sk_i = (S_i, e_i), i = 0, 1, ..., k$. Обозначим $M_i' = M_i S_{i-1}$ и зашифруем матрицы и вектор следующим образом:

$$c = VecEnc_{sk_0}(v) = S_0^{-1}(\beta v + e) \mod q$$

$$C_i = MatEnc(M_i') = S_i^{-1}(M_i S_{i-1} G + E_i)$$

Для представления гомоморфного шифрования обозначим:

$$c_0 = c, v_0 = v$$

$$c_i = C_i G^{-1}(c_{i-1}) \mod q$$

$$v_i = M_i v_{i-1} = (\prod_{j=i}^{1} M_j)v$$

Гомоморфность по матрично-векторному умножению можно показать индуктивными соображениями.

Каждый c_i представим в виде $c_i = S^{-1}(\beta v_i + e_i)$ где e_i по-прежнему вектор-шум малой нормы.

Распишем c_{i+1} следующим образом:

$$c_{i+1} = C_{i+1} G^{-1}(c_i) \mod q = ... = S^{-1}(\beta M_{i+1} v_i + M_{i+1} e_i + E_{i+1} G^{-1}(c_i))$$

$\beta M_{i+1} v_i = v_{i+1}$, часть в скобках правее этой при достаточно жёстких ограничениях на нормы шумов можно проассоциировать также с шумом, т.е. обозначить за e_{i+1}. Т.о. мы показали, что $c_{i+1} = VecEnc(v_{i+1})$, откуда гомоморфизм очевиден.

Подробную реализацию алгоритма, см. алгоритм 7.

2.3 Криптостойкость схемы

Стойкость нашей системы опирается на сложность неоднородной задачи NTRU (*Inhomogeneous NTRU*, обозначение: iNTRU), существующей в обыч-

ной, т.е. подразумевающей работу с числами/элементами каких-то колец, и матричной постановке.

Обычная постановка Введём следующее определение

Определение 2.3.1. *Пусть заданы следующие параметры: кольцо R, модуль q, — симметричное распределение над R, сконцентрированное на элементах, много меньших по норме чем q. Обозначим $\lceil \log q \rceil$. Тогда iNTRU-распределением (iNTRU-distribution) с этими параметрами называют распределение, получаемое путём исполнения следующего алгоритма:*

- *Некоторым образом выберем $s \leftarrow R/qR$;*
- *Выберем $e_i \leftarrow \chi$ для $i = 0, 1, ..., l-1$;*
- $a_0 := \frac{e_0}{s} \mod q$;
- $a_i := \frac{2^{i-1} - e_i}{s} \mod q$ *для $i = 1, 2, ..., l-1$;*
- *Вернём $a = (a_0, ..., a_{l-1})$*

Набор a, как правило, называют iNTRU-кортежем, iNTRU-выборкой (iNTRU-tuple).

Параметр s, естественно, носит секретный характер.

iNTRU в форме проблемы поиска ставится так: по известному кортежу $a = (a_0, ..., a_{l-1}$ и модулю q определить секрет s состоит в том, чтобы отличить данное распределение от равномерного над $(R/qR)^l$

iNTRU в форме проблемы разрешимости: по известному кортежу $a = (a_0, ..., a_{l-1}$ и модулю q определить, выборкой из какого распределения он является: iNTRU или равномерного над $(R/qR)^l$.

Матричная постановка В такой постановке идейно всё то же самое, только s и e_i заменяются на квадратные целочисленные матрицу S и E_i размера $n \times n$, a_i заменяются на матрицы вида $A_0 := -S^{-1} \times E_0$ $A_i := S^{-1} \times (2^i I - E_i)$. Пусть также $m' = n(l+1)$, а $G' := [\nvdash | I | 2I | ... | 2^{l-1} I] \in \chi n \times m'$, где I — единичная матрица, G' — видоизменённая (расширенная слева

нулевым блоком) гаджет-матрица, а χ — распределение над $\mathbb{Z}$, сконцентрированное на элементах с нормой много меньшей q. Тогда процедура, задающая MiNTRU-распределение, выглядит так:

- Выберем некоторым образом $S \leftarrow \mathbb{Z}_q^{n\times n}$;
- Выберем $E' \leftarrow \chi^{n\times m'}$;
- Вернём $A' := S^{-1} \times (G' - E') \mod q$;

Нами препдолагается, что MiNTRU псевдослучайно, вследствие чего A' неотличимо от равномерного матричного распределения на $\mathbb{Z}_q^{n\times m'}$.

iNTRU остаётся сложной даже в том случае, когда ключ берётся из распределения χ, что перекликается с задачей LWE. Вместе с этим рассмотрим ещё одну модификацию iNTRU — MiNTRU^S (*MiNTRU with small secret*):

- n, m', q, χ остаются теми же, что и выше;
- Обозначим $m = n\lceil \log q \rceil = m' - n$;
- $G = [I|2I|...|2^{l-1}I] \in \mathbb{Z}^{n\times m}$;

Тогда MiNTRU^S-распределением называется распределение, описанное процедурой ниже:

- Выберем $S \leftarrow \chi^{n\times n}$;
- Выберем $E \leftarrow \chi^{n\times m}$;
- Вернём $A := S^{-1}(G - E) \mod q$;

Далее сформулируем две леммы, доказательство которых приведено [Gen+19] на стр. 14-15:

Лемма 1. *Для заданных параметров n, m, m', q, χ из псевдослучайности MiNTRU следует псевдослучайность MiNTRU^S;*

Лемма 2. *В случае псевдослучайности MiNTRU^S существует распределение шума, при котором GGHLM будет семантически стойким;*

2.4 Реализация схемы GGHLM

Перед тем, как перейти к реализации схемы, сделаем очень важное напоминание и дополнение.

Гаджет-матрица по определению имеет следующий вид: $G = I \bigotimes g$, где I — единичная матрица, а вектор $g = (1, 2, ..., 2^{\log q - 1})$ можно трактовать как оператор обратный битовой декомпозиции. Секрет для матрицы G является известной.

Наша цель: используя перешифрование, минимизировать накопление шума. Для этого мы переопределим $g := (1, b, ..., b^{\log_b q - 1})$ (и, соответственно, матрицу G) как это сделано в [MP11] и воспользуемся идеей декомпозиции из [AP14].

Учёт замечения выше позволяет нам *отказаться* от явного использования гаджет-матрицы в реализации схемы. Более того, она не будет никоим образом участвовать в шифровании/дешифровании, а её вклад будет виден только на этапе запуска зашифрованного НКА (в виде битовой декомпозиции зашифрованного вектора состояний).

В предположении $circular$-стойкости схемы, нам нужен один-единственный ключ — матрицы S, S^{-1}

Итого имеем:

- Один ключ, представленный парой матриц S, S^{-1};
- Из кольца $\mathbb{Z}_q$ $\mathbb{Z}_{2^{\log q \log b}}$;
- Шифрование матриц: $Enc(M) = \beta S^{-1} M S + S^{-1} E$
- Шифрование векторов: $Enc(u) = \beta S^{-1} u + S^{-1} e$

А теперь перейдём к самой реализации:

Алгоритм 4 Семплирование «случайным шумом» 0-1 матрицы

```
def fill_matrix_zero_01_random(mat):
    for i in range(mat.nrows()):
        for j in range(mat.ncols()):
            mat[i, j] = randrange(2)
```

Алгоритм 5 Генерация GGHLM-ключа

```python
@dataclass
class GGHLMKeys:
    n: int
    logb: int
    logq: int
    S: object = None
    S_inv: object = None

    def __post_init__(self):
        self.q = 2  ** self.logq
        self.b = 2  ** self.logb
        self.qq = pow(2.0, self.logb * self.logq)


        self.S = zero_matrix(self.ZZq, self.n, self.n)
            fill_matrix_zero_01_random(self.S)
            det = self.S.det()
            if det % 2 == 1:
                break
        self.S_inv = self.S.inverse_of_unit()
```

```
def fill_matrix_zero_01_random(mat):
    for i in range(mat.nrows()):
        for j in range(mat.ncols()):
            mat[i, j] = randrange(2)
GGHLMKeys(
    n=9,
    logb=7,
    logq=6,
    S=[1 1 0 1 1 1 1 1 0]
[0 0 1 1 1 1 1 1 1]
[0 1 0 1 0 1 0 0 0]
[0 1 1 1 0 0 0 1 0]
[1 0 1 1 1 1 0 1 0]
[1 0 0 0 0 1 0 1 0]
[0 1 0 0 1 0 1 1 0]
[0 1 1 0 1 1 0 1 1]
[0 0 1 1 1 1 1 1 0],
    S_inv=[            2 2932031007402 146601550370
[            1 2932031007402 1466015503701 29320310
[            2 1466015503700 2932031007401 14660155
[4398046511103             1             1
[4398046511103 2932031007403 1466015503702 29320310
[            0 1466015503701 2932031007403 14660155
[            2 2932031007402 1466015503700 29320310
[4398046511102             1             1
[            0             1             0
)
```

Чтобы расшифровать криптосообщение мы умножим секретный ключ на шифротекст.

$$SK^T C = SK^T (PK \cdot R + M \cdot G) = e^T \cdot R + M \cdot SK^T \cdot G$$

Нам надо расшифровать и получить M. Т.к. $e^T R$ — мало, мы сфокусируем свое внимание на втором слагаемом, и попробуем подобрать такое M, чтоб оно легло максимально близко.

Вычислим массив $r = \frac{sk^T C}{t^T G}$. В идеале, каждый индекс этого массива должен содержать значение M, но на практике это маловероятно. Чтобы вычислить правильное значение значение M, мы берем каждое уникальное значение из этого массива и умножаем его на $SK^T G$. Затем вычисляем дистанцию $d = ||rSK^T G - SK^T C||$ для каждого уникального значения в r. Правильное значение M должно иметь минимальную дистанцию d

Более формально, см. алгоритмы 8 и 9.

Согласно теореме 1, и сложности обучения с ошибками (LWE), мы знаем, что для равномерного распределения публичных ключей, и случайно выбранных R, произведение $PK \cdot R$ тоже будет истинно вероятностным, и таким образом, шифертекст $C(M)$ будет эффективно скрывать инфор-

Алгоритм 6 GGHLM-шифрование векторов и неквадратных матриц

- M — шифруемый вектор или неквадратная матрица;
- E — сгенерированный случайный матричный шум
- Шифр:

$$Enc(M) = \beta S^{-1} \cdot M + S^{-1} \cdot E$$

```
def gghlm_encrypt_nonsquare(key, msg, log_beta):
    beta = 2  ** log_beta
    E = get_random_01_matrix(key.ZZq, msg.nrows(), msg.ncols())
    return beta * key.S_inv * msg + key.S_inv * E
```

Алгоритм 7 GGHLM-шифрование квадратных матриц

- M — шифруемая матрица,
- E — сгенерированный случайный матричный шум,
- Шифр:

$$Enc(M) = \beta S^{-1} M S + S^{-1} E,$$

```
def gghlm_encrypt_square_matrix(key, msg, log_beta):
    beta = 2  ** log_beta
    E = get_random_01_matrix(key.ZZq, msg.nrows(), msg.ncols())
    return  beta * key.S_inv * msg * key.S + key.S_inv * E
```

мацию M.

Теперь рассмотрим операции.

Определим сложение как $C^+ = C_1 + C_2$. Распишем:

$$PK^T \cdot C^+ = PK^T \cdot (C_1 + C_2) = (e_1^T + e_2^T) + (M_1 + M_2) \cdot PK^T G.$$

Т.к. членом с возмущениями можно пренебречь, мы получим $(M_1 + M_2) \cdot PK^T G$.

Определим умножение как $C^\times = C_1 \cdot G^{-1} \cdot C_2$. Распишем:

Алгоритм 8 GGHLM-дешифрование неквадратных матриц

Шифротекст:

$$C = S^{-1}(\beta \cdot M + E) = \beta S^{-1} \cdot M + S^{-1}E$$

- Поскольку E — вектор малой нормы, то

$$C \approx \beta S^{-1} \cdot M$$

- «Деления в кольцах — нет» — используем битовый сдвиг:

$$M = S \cdot (C >> \text{log}\, \beta)$$

```
def mat_right_shift(mat, shift):
    for i in range(mat.nrows()):
        for j in range(mat.ncols()):
            mat[i, j] >>= shift
```

```
def gghlm_decrypt_nonsquare(key, C, log_beta):
    # Раскодируем неквадратные матрицы (вектора)
    res = key.S * C
    mat_right_shift(res, log_beta)
    return res
```

$$C_1 \cdot G^{-1} \cdot C_2 = \left(-A_1 G^{-1}(C_2) s^T A_1 G^{-1}(C_2)\right) + M_1 \left(-A_2 s^T A_2 + e_2^T\right) + M_1 \cdot M_2 \cdot G.$$

Также пренебрегаем первыми двумя членами с возмущениями, оставив нас с $M_1 \cdot M_2 \cdot G$.

Алгоритм 9 GGHLM-дешифрование квадратных матриц

Шифротекст:

$$C = S^{-1}(\beta \cdot M + E) = \beta S^{-1} \cdot M + S^{-1} E$$

- Поскольку E — вектор малой нормы, то

$$C \approx \beta S^{-1} \cdot M$$

- «Деления в кольцах — нет» — используем битовый сдвиг:

$$M = S \cdot (C >> \log \beta)$$

```
def gghlm_decrypt_square_matrix(key, C, log_beta):
    # Раскодируем квадратные матрицы
    return gghlm_decrypt_nonsquare(key, C, log_beta) * key.S_inv
```

2.5 Работа зашифрованного НКА-автомата

Замечание 2.5.1. Тут надо расписать этот текст может быть более по-подробнее.

Итак, с учетом результатов предыдущих разделов, работа зашифрованного НКА у нас будет выглядеть следующим образом

- Инициализация системы представлена в алгоритме 10, там мы специальным образом, для каждой матрицы перехода (соотвествующей каждому символу алфавита), получаем набор зашифрованных матриц («obfs_nfa») — именно набор, чтобы в процессе вычислений выполнять процедуру перешифровки («bootstrapping»), и избавляться от накопления ошибок умножения.

- При инициализации нашей системы исполнения зашифрованных НКА, мы получаем набор обфусцированных матриц, построенных по нашим исходным матрицам перехода, см. рис. 2.6

Алгоритм 10 Инициализация зашифрованного НКА-автомата

```
GGHLMRegexByNFA(
    regex_str='(a*|b)',
    key=gghlm.GGHLMKeys(
        n=16,
        logb=6,
        logq=7,
        S=[0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0]
[1 1 0 1 1 1 1 0 1 0 1 1 0 0 0 0]
[0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1]
[1 0 0 1 0 0 1 1 1 1 1 1 1 0 0 0]
[1 1 0 1 0 0 0 0 0 1 0 1 0 1 1 0]
[0 0 0 1 1 0 0 0 0 0 0 1 1 0 1 0]
[0 0 1 1 0 0 0 1 1 1 1 1 1 0 1 0]
[0 0 1 1 1 1 1 1 1 0 1 0 0 0 0 0]
[0 0 0 1 1 0 0 1 1 1 0 1 0 0 0 1]
[1 0 1 1 0 0 0 1 1 1 1 1 1 0 1 0]
[1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0]
[1 1 1 0 1 0 1 1 0 0 0 0 0 1 0 0]
[1 0 0 1 0 0 1 1 0 0 1 0 1 0 1 0]
[0 0 1 1 0 1 0 0 1 1 1 1 0 0 0 0]
[1 0 0 0 0 0 1 1 1 1 0 1 1 1 0 1]
[0 1 1 0 0 0 1 1 0 1 1 1 1 1 0 1],
        S_inv=[            0            0            0            0            0
[1599289640402  399822410101 3198579280803 1599289640401 2398934460602  799644820200 35984016909
[2398934460602 2798756870703  399822410100 2398934460601 3598401690902 1199467230301 31985792808
[3598401690903 1999112050502 2798756870702 3598401690903 3198579280804 3998224101004  39982241006
[3198579280803  799644820200 1999112050502 3198579280804  399822410101 1599289640402 27987568707
[1199467230302 3598401690903 2398934460603 1199467230301 3998224101003 2798756870702 15992896404
[1199467230301 3598401690904 2398934460602 1199467230300 3998224101001 2798756870702 15992896404
[ 799644820201 2398934460602 1599289640402  799644820201 1199467230301  399822410100 39982241016
[2398934460602 2798756870703  399822410100 2398934460602 3598401690904 1199467230301 31985792808
[3598401690904 1999112050501 2798756870703 3598401690904 3198579280802 3998224101003  39982241011
[2798756870702 3998224101003 1199467230301 2798756870704 1999112050504 3598401690904  79964482011
[1599289640401  399822410101 3198579280803 1599289640401 2398934460601  799644820201 35984016909
[1599289640402  399822410100 3198579280803 1599289640402 2398934460603  799644820201 35984016909
[3998224101003 3198579280802 3598401690903 3998224101005 1599289640405 1999112050503 23989344605
[3198579280803  799644820201 1999112050502 3198579280802  399822410099 1599289640401 27987568707
[2398934460602 2798756870703  399822410100 2398934460601 3598401690903 1199467230301 31985792808
    ),
    log_beta=35
)
```

```python
@dataclass
class GGHLMRegexByNFA(RegexByNFA):
    key: GGHLMKeys
    log_beta: int

    def __post_init__(self):
        self.N = max(self.key.n, self.N)
        self.ring = self.key.ZZq

        self.obfs_nfa = [None] * self.sigma * self.key.logq
        for i in range(self.sigma):
            for j in range(self.key.logq):
                entry = i * self.key.logq + j
                self.obfs_nfa[
                    entry
                ] = gghlm_encrypt_square_matrix(
                    self.key,
                    self.mats[i],
                    self.key.logb * j)
```

- Обработка зашифрованного текста представлена в алгоритме 11 — изначально мы на нашей территории шифруем вектор-состояние НКА, затем на «вражеской территории» для каждого символа выполняем процедуру умножения на матрицу с перешифрованием («eval»), и в конце мы, снова на нашей территории, дешифруем зашифрованное состояние НКА, чтобы узнать, есть ли вероятности нахождения в финальном состоянии.

- Перешифрование происходит при обработке каждого символа в процедуре «eval», см. алг. 12.

Замечание 2.5.2. Ну тут надо сделать верификацию кодом гомоморфизма, может поисследовать, когда тут появляется ошибка (на множестве итераций), и когда расшифровывается не то, или гомоморфизм нарушается.
Графики всякие (время от длины ключа например), вероятности ошибок, что-то такое.

Алгоритм 11 Распознавание строки с зашифрованными переходами

```
def match_text_encrypted(self, text):

        self.set_start_state()
        self.enc_state = gghlm_encrypt_nonsquare(self.key,
                self.state, self.log_beta)

        print('state: ' + str(self.state.T))
        print('encstate: ' + str(self.enc_state.T))

        # Это может происходить на «вражеской» территории.
        for chr_ in text:
            print(f'chr: {chr_}')
            idx = self.symbol2index[chr_]
            self.eval(idx)
            print('encstate: ' + str(self.enc_state.T))

        self.state  = gghlm_decrypt_nonsquare(self.key,
                    self.enc_state, self.log_beta)
        print('state: ' + str(self.state.T))

        for state in self.nfa.final_states:
            if self.state[self.state2index[state]] > 0:
                return True
        return False
```

```
state: [1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0]
encstate: [4363686772736 2879970797754 1949134249239 724
1190096392565 3311029333643 3242309856907 3673368392797 19
896476810146 1518075713351 2811251321019 2811251321018 293
1258815869300 1914774510871]
chr: a
encstate: [ 236 3985531327330 2697214672169 2405280845858
1408898016834 2989148498695 2989148498564 1992765664249 26
1288316659324 412515180836 704449008899 704449007942 12058
291933826768 2697214670977]
chr: a
encstate: [4398046510970 3903028290434
3236657606582 247509107326 2570286922402
1827759594298 1827759594747 4150537401622
3236657602489 666370689958 495018216654
3484166717584 3484166715695 1903916230384
2989148499276 3236657604484]
chr: a
encstate: [ 234 177698850509 2221235613003
4309197082743 4264772374714 133274142831
133274143238 88849426216 2221235607953
2354509755624 4220347655930 2132386188357
2132386186232 1910262616384 2310085041401
2221235610920]
state: [ 0 0 0 0 0 56 0 0 0 0 0 0 0 0 0 0]
True
```

Рис. 2.1: Представление регулярных выражений в виде НКА

```
def demo_regex_to_nfa():
    regex = Regex("(a*|b)")
    print(regex.get_tree_str())
    enfa = regex.to_epsilon_nfa()
    G = enfa.to_networkx()
    write_dot(G, log.dotprefix + 'demo_regex_to_enfa.dot')

    nfa = enfa.remove_epsilon_transitions()
    G = nfa.to_networkx()
    write_dot(G, LOG.dotprefix + 'demo_regex_to_nfa_no_eps.dot')

    mats, state2index, symbol2index = nfa2_zzq_matrices(nfa)


    for sym in symbol2index:
        print(f'Матрица переходов для «{sym}»')
        print(str(mats[symbol2index[sym]]))

    printverb('states',  state2index)
    printverb('symbols', symbol2index)
```

```
Operator(Union)
 Operator(Kleene Star)
  Symbol(a)
 Symbol(b)
```

Матрица переходов для «a»

```
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[1 0 1 0 1 1 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

Граф переходов для НКА с эпсилон-переходами:

Граф переходов для НКА без эпсилон-переходов:

Матрица переходов для «b»

```
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 1 0]
```

Рис. 2.2: «NFAs for «(a*|b)»)

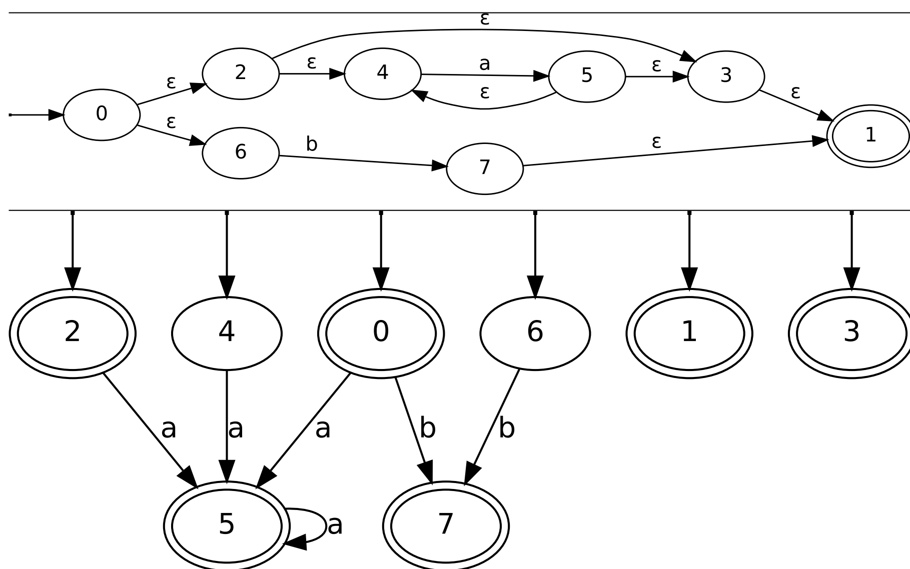
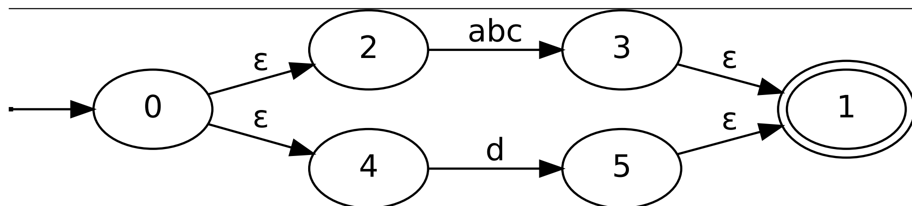


Рис. 2.3: «NFAs for «abc|d»)



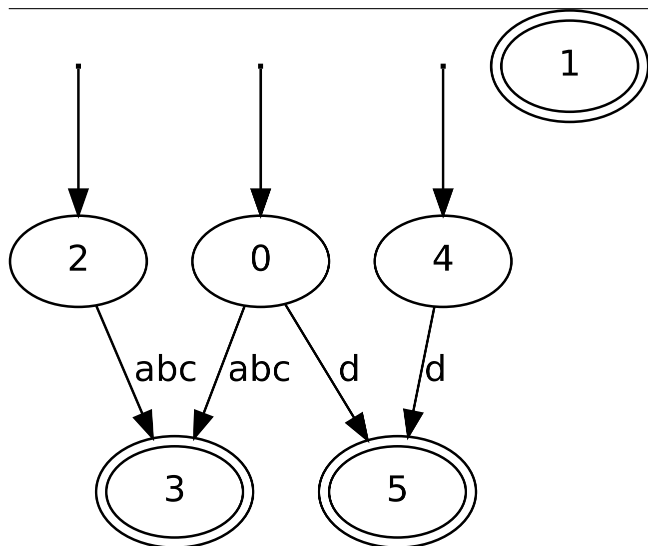
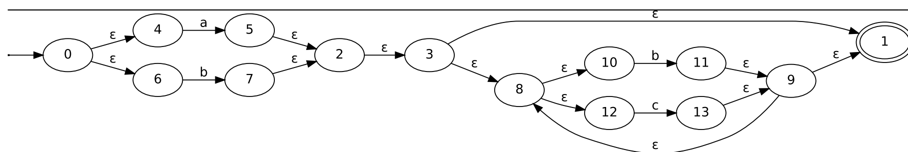


Рис. 2.4: «NFAs for «(a|b)(b|c)*»)



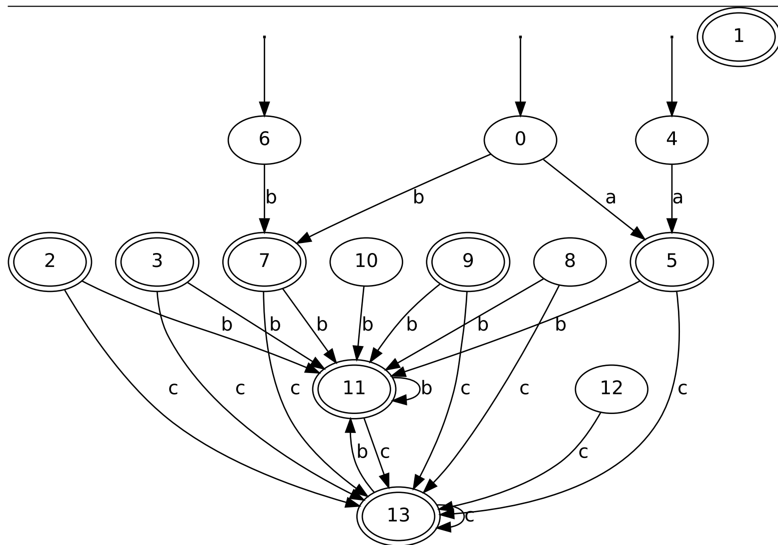


Рис. 2.5: «NFAs for «(a|b)*c»)

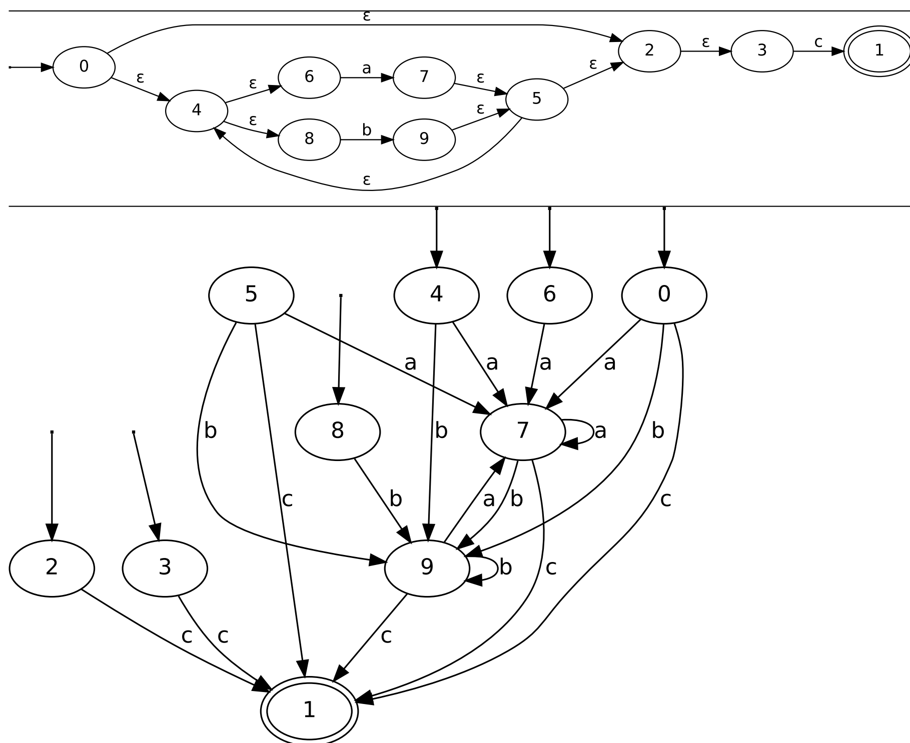


Рис. 2.6: Зашифрованные матрицы перехода

```
{
    0: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0],
    1: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
}
```

```
[
    [           1           0           0           0 4398046511103           0
[2398934460602 2398934460602 3198579280802 1199467230302 1999112050503  799644820202  3998224101
[3598401690903 3598401690904  399822410099 3998224101005  799644820201 1199467230303 27987568707
[3198579280803 3198579280801 2798756870705 1599289640402 1199467230298 3998224101001 19991120505
[ 399822410101  399822410101 1999112050503 2398934460601 3998224101004 1599289640401  7996448202
[3998224101004 3998224101008 2398934460602 1999112050502  399822410106 2798756870705 35984016909
[3998224101002 3998224101007 2398934460599 1999112050503  399822410105 2798756870707 35984016909
[1199467230301 1199467230302 1599289640401 2798756870702 3198579280805  399822410101 23989344606
[3598401690903 3598401690901  399822410101 3998224101004  799644820198 1199467230299 27987568707
[3198579280803 3198579280807 2798756870701 1599289640401 1199467230307 3998224101008 19991120505
[1999112050503 1999112050498 1199467230304 3198579280802 2398934460597 3598401690899 39982241010
[2398934460602 2398934460602 3198579280801 1199467230301 1999112050502  799644820202  3998224101
[2398934460603 2398934460602 3198579280804 1199467230301 1999112050501  799644820199  3998224101
[1599289640402 1599289640397 3598401690908  799644820199 2798756870696 1999112050495 31985792808
[ 399822410099  399822410103 1999112050500 2398934460603 3998224101008 1599289640405  7996448202
[3598401690903 3598401690903  399822410101 3998224101005  799644820199 1199467230301 27987568707
    [           1           0           1           0 4398046511103           0
[3598401690901 1599289640320 3998224100838  799644820121 3598401690824 1999112050503 11994672303
[3198579280799 2398934460480 1599289640155 1199467230182 3198579280682  799644820204 39982241010
[ 399822410106 3598401691069 2398934460942 3998224101169  399822410267 1199467230295 15992896404
[2798756870706 3198579280897 3598401691093 1599289640493 2798756870794 3998224101003 23989344606
[1599289640396 1199467230147  799644819883 2798756870550 1599289640246  399822410107 19991120505
[1599289640390 1199467229955  799644819496 2798756870361 1599289640057  399822410112 19991120504
[3998224101002  799644820161 1999112050419  399822410061 3998224100964 3198579280804 27987568707
[3198579280806 2398934460734 1599289640670 1199467230432 3198579280936  799644820195 39982241010
[ 399822410095 3598401690692 2398934460169 3998224100794  399822409887 1199467230311 15992896404
[ 799644820212 2798756871037  399822410783 3598401691233  799644820534 2398934460592 31985792808
[3598401690898 1599289640257 3998224100711  799644820057 3598401690760 1999112050505 11994672302
[3598401690907 1599289640512 3998224101226  799644820309 3598401691012 1999112050499 11994672303
[2398934460617 3998224101500 1199467231311 1999112050991 2398934461096 2798756870687  7996448202
[2798756870695 3198579280578 3598401690445 1599289640180 2798756870479 3998224101011 23989344605
[3198579280804 2398934460606 1599289640413 1199467230307 3198579280809  799644820200 39982241010
    [4398046511103           0 4398046511103           1           1           1
[3198579280806  799644814986 3198579270379 1999112045286 4398046505891 3998224101002 43980465111
[ 399822410103 1199467222479  399822394464  799644812379 4398046503285 1599289640399 43980465111
[2798756870698 3998224111806 2798756892296 1199467241103         10800 2398934460604 43980465111
[1999112050499 1599289646361 1999112062415 3998224106964          5958 3598401690906
[2398934460609 2798756860646 2398934440499  399822400043 4398046501049  799644820199
[2398934460612 2798756848354 2398934415928  399822387751 4398046488761  799644820194 43980465111
[1599289640403  399822407493 1599289635190 3198579278195 4398046508497 1999112050501
[ 399822410098 1199467238868  399822427226  799644828767          8565 1599289640403 43980465111
[2798756870709 3998224087222 2798756843153 1199467216519 4398046497325 2398934460600
[1199467230291 3598401712506 1199467273486 2398934482207         21598  399822410107
[3198579280805  799644810889 3198579262188 1999112041190 4398046501794 3998224101001 43980465111
[3198579280801  799644827278 3198579294950 1999112057579          7075 3998224101006
[3598401690890 1999112082535 3598401754938 2798756902737         32024 1199467230310
[1999112050509 1599289625875 1999112021464 3998224086477 4398046496581 3598401690899
```

Алгоритм 12 Переход по символу в зашифрованных НКА

```
def bit_dec(self, state_bd, state):
        for i in range(state.nrows()):
            t = state[i, 0]
            for j in range(self.key.logq):
                state_bd[j, i] = t % self.key.b
```

alg-gghlmregexbynfa

```
def eval(self, idx):

        work_space = zero_vector(self.ring, self.state.nrows() )
        state_bd = zero_matrix(self.ring, self.key.logq, self.state.nrows() )
        temp_state = zero_vector(self.ring, self.state.nrows() )
        self.bit_dec(state_bd, self.enc_state)
        printverb('enc-state', self.enc_state)
        printverb('state-bd', state_bd)

        for i in range(self.key.logq):
            entry = idx * self.key.logq + i
            work_space = self.obfs_nfa[entry] * state_bd[i]
            temp_state += work_space

        printverb('temp-state', temp_state)

        for i in range(self.enc_state.nrows()):
            self.enc_state[i, 0] = temp_state[i]
```

«enc_state»:

```
[4398046510970]
[3903028290434]
[3236657606582]
[ 247509107326]
[2570286922402]
[1827759594298]
[1827759594747]
[4150537401622]
[3236657602489]
[ 666370689958]
[ 495018216654]
[3484166717584]
[3484166715695]
[1903916230384]
[2989148499276]
[3236657604484]
```

«state_bd»:

```
[58  2 54 62 34 58 59 22 57 38 14 16 47 48 12  4]
[58  2 54 62 34 58 59 22 57 38 14 16 47 48 12  4]
[58  2 54 62 34 58 59 22 57 38 14 16 47 48 12  4]
[58  2 54 62 34 58 59 22 57 38 14 16 47 48 12  4]
[58  2 54 62 34 58 59 22 57 38 14 16 47 48 12  4]
[58  2 54 62 34 58 59 22 57 38 14 16 47 48 12  4]
[58  2 54 62 34 58 59 22 57 38 14 16 47 48 12  4]
```

«temp_state»:

```
(234, 177698850509, 2221235613003, 4309197082743, 4264772374714, 133274142831, 133274143238, 888
```

Глава 3

Приложения

3.1 Схема гомоморфного шифрования Gentry-Sahai-Waters

Будем использовать генерацию простого числа с проверкой вероятностными или детерминированным тестом Миллера-Рабина.

Алгоритм 13 Генерация простого числа

```
from miller_rabin import miller_rabin
def gen_prime(b):
    '''Простое число из b бит'''
    OK = False
    while not OK:
        p = random.randint( 2**(b-1), 2**b )
        OK = miller_rabin(p)
    return p
```

Определение 3.1.1. ***Простое число Софи Жермен*** *(Sophie Germain prime) — это такое простое число p, что число $2p + 1$ также простое. Как и для простых чисел-близнецов, предполагается, что количество простых Софи Жермен бесконечно, но это не доказано.*

Опишем неформально алгоритм 16

Алгоритм 14 Генерация простого числа Софи Жермен

```
def generate_sophie_germain_prime(k):
    OK = False
    while not OK:
        p = gen_prime(k - 1)
        sp = 2 * p + 1
        OK = miller_rabin(sp)
    return p
```

Алгоритм 15 Описание GSW-ключа

```
class GSWKeys:
    def __init__(self, k, q, SK, e, A, PK, datatype):
        self.n  = k
        self.q  = q
        self.SK = SK
        self.e  = e
        self.A  = A
        self.PK = PK
        self.datatype = datatype

        self.l  = math.ceil(math.log2(q))
        self.m  = self.n * self.l
```

- Выберем параметр безопасности, влияющий на длину ключа — k.
- Вероятностно выберем простое число Софи-Жермен (опр. 3.1.1 «SGP» длиной k бит.
- Положим $l = \lceil \log q \rceil$ и $m = n \cdot l$
- Вероятностно выбираем секрет

$$s \leftarrow Z_q^{n-1}$$

- Вероятностно выбираем матрицу

$$A \leftarrow Z_q^{(n-1) \times m}$$

- Семплируем вектор ошибок $e \leftarrow \xi^m$, где ξ — нормальное распределение целых по модулю q.
- Секретный ключ будет $SK = (s||1)$
- Публичный ключ
$$PK = \binom{-A}{s^T A + e^T}$$

Чтобы расшифровать криптосообщение мы умножим секретный ключ на шифротекст.

$$SK^T C = SK^T (PK \cdot R + M \cdot G) = e^T \cdot R + M \cdot SK^T \cdot G$$

Нам надо расшифровать и получить M. Т.к. $e^T R$ — мало, мы сфокусируем свое внимание на втором слагаемом, и попробуем подобрать такое M, чтоб оно легло максимально близко.

Вычислим массив $r = \frac{sk^T C}{t^T G}$. В идеале, каждый индекс этого массива должен содержать значение M, но на практике это маловероятно. Чтобы вычислить правильное значение значение M, мы берем каждое уникальное значение из этого массива и умножаем его на $SK^T G$. Затем вычисляем дистанцию $d = ||rSK^T G - SK^T C||$ для каждого уникального значения в r. Правильное значение M должно иметь минимальную дистанцию d

Более формально, см. алгоритм 19.

Теорема 1. ***Теорема об остатке хеширования*** *(«Leftover Hash Lemma», см. [ILL89]).*

Пусть X — секретный ключ, состоящий из n бит, выбранных одинаково и независимо из равномерного распределения над $\{0, 1\}$. Пусть противнику удалось узнать значения некоторых $t, t < n$ бит секретного ключа, причём каких именно — неизвестно. Тогда мы можем создать новый ключ, состоящий из $n - t$ бит, выбранных так же равномерно и независимо, о которых противнику не будет известно сколь либо значимой информации.

Согласно теореме 1, и сложности обучения с ошибками (LWE), мы знаем, что для равномерного распределения публичных ключей, и случайно выбранных R, произведение $PK \cdot R$ тоже будет истинно вероятностным,

и таким образом, шифертекст $C(M)$ будет эффективно скрывать информацию M.

Теперь рассмотрим операции.

Определим сложение как $C^+ = C_1 + C_2$. Распишем:

$$PK^T \cdot C^+ = PK^T \cdot (C_1 + C_2) = (e_1^T + e_2^T) + (M_1 + M_2) \cdot PK^T G.$$

Т.к. членом с возмущениями можно пренебречь, мы получим $(M_1 + M_2) \cdot PK^T G$.

Определим умножение как $C^\times = C_1 \cdot G^{-1} \cdot C_2$. Распишем:

$$C_1 \cdot G^{-1} \cdot C_2 = \left(-A_1 G^{-1}(C_2) s^T A_1 G^{-1}(C_2)\right) + M_1 \left(-A_2 s^T A_2 + e_2^T\right) + M_1 \cdot M_2 \cdot G.$$

Также пренебрегаем первыми двумя членами с возмущениями, оставив нас с $M_1 \cdot M_2 \cdot G$.

Замечание 3.1.1. Надо перечитать, подчистить, может что-то расписать.
Ну тут надо сделать верификацию кодом гомоморфизма, может поисследовать, когда тут появляется ошибка (на множестве итераций), и когда расшифровывается не то, или гомоморфизм нарушается.
Графики всякие (время от длины ключа например), вероятности ошибок, что-то такое.
Тогда это вполне можно перенести из приложений в один из основных разделов.

Алгоритм 16 Генерация GSW-ключа

```
def keygen(k):

    print(f'{sect}Генерация модуля')
    q = generate_sophie_germain_prime(k)
    l = math.ceil(math.log2(q))
    print(f'q = {q}  l = {l}')

    print(f'{sect}Генерим секретный ключ')
    n = k
    m = n * l
    print(f'n = {n}   m = {m} ')
    s = np.random.randint(q, size=n-1, dtype=np.int64)
    SK = np.append(s, 1)
    print(f'SK = {SK}')

    print(f'{sect}Вектор ошибок')
    e = np.rint(np.random.normal(scale=1.0, size=m))
    print(f'e {e.shape} = {e}')

    print(f'{sect}Случайная матрица')
    A = np.random.randint(q, size=(n-1, m))
    print(f'A {A.shape} = \n {A}')

    print(f'{sect}Публичный ключ')
    PK = np.vstack((-A, np.dot(s, A) + e)) % q
    print(f'PK {PK.shape} =\n {PK}')

    check = np.dot(SK, PK) % q
    OK = np.all(check == (e % q))
    if OK:
        print('Проверка ключа — OK') # SK*PK == e
    else:
        print('Проверка ключа не удалась')

    return GSWKeys(k, q, SK, e, A, PK, datatype)
```

```
*******************
Генерация модуля

q = 3491  l = 12


*******************
Генерим секретный ключ

n = 13   m = 156

SK = [2030   89  997  687 2478 3250 2498
 3238 2854 3054 3198 1871    1]


*******************
Вектор ошибок

e (156,) = [-1  1 ...  0 -1]


*******************
Случайная матрица

A (12, 156) =
 [[1233 2638 ... 3472 3039]
 [1342 1807 ... 1362 1106]
 ...
 [ 379 1312 ... 1074 2500]
 [ 286 2254 ... 2966  604]]


*******************
Публичный ключ

PK (13, 156) =
 [[2258  853 ...   19  452]
 [2149 1684 ... 2129 2385]
 ...
 [3205 1237 ...  525 2887]
 [3041 1709 ... 2874 2434]]

Проверка ключа — OK
```

Алгоритм 17 Матрица-генератор G для схемы GSW

Матрица-генератор $G^{n \times m}$:

$$\begin{bmatrix} 1 & 2 & ... & 2^{l-1} & 0 & 0 & ... & 0 & ... & 0 & 0 & 0 & 0 \\ 0 & 0 & ... & 0 & 1 & 2 & ... & 2^{l-1} & ... & 0 & ... & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & & & & & & & & & & & & \\ 0 & 0 & ... & 0 & 0 & 0 & ... & 0 & ... & 1 & 2 & ... & 2^{l-1} \end{bmatrix}$$

```python
def generator_matrix(l, n):

    print(f'l={l}, n={n}')

    g = 2**np.arange(l)
    G = block_diag(*[g for null in range(n)])

    print(f'G = \n {G}')
    return G
```

```
l=12, n=13

G =
 [[   1    2    4    8   16   32   64  128  256  512 1024 2048    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    1    2    4    8   16   32   64 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...   32   64  128  256  512 1024 2048    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 ...    0    0    0    0    0    0    0    1    2    4    8   16   32   64  128  256  512 1024 2048]]
```

Алгоритм 18 GSW-шифрование

- $message$ — входное сообщение, целое число из некого пространства сообщений M, PK — публичный ключ.
- Выберем случайную матрицу $R \leftarrow \{0,1\}^{m \times m}$
- Вычислим шифротекст

$$C = PK \cdot R + message \cdot G \in Z_q^{n \times m}$$

```
def gsw_encrypt(keys, message):
    G = generator_matrix(keys.l, keys.n)

    print(f'{sect}Encrypting message, {message}')
    R = np.random.randint(
            2, size=(keys.m, keys.m)
            )

    print(f'{sect}Encrypting message, {message}')

    encmsg = (np.dot(keys.PK, R) + message*G) % keys.q
    print(f'encmsg {encmsg.shape} = \n {encmsg}')

    return encmsg
```

```
*******************
Encrypting message, 34


*******************
Encrypting message, 34

encmsg (13, 156) =
 [[ 371 2811 ... 1231  272]
 [1794 3130 ...  962  584]
 ...
 [1060 1770 ... 2364  398]
 [ 433 1317 ... 1726 1058]]
```

Алгоритм 19 GSW-дешифрование

```
def gsw_decrypt(keys, ciphertext):
    G = generator_matrix(keys.l, keys.n)

    print(f'{sect}Ciphertext {ciphertext.shape} = \n {ciphertext}')

    msg  = np.dot(keys.SK, ciphertext) % keys.q
    print(f'msg {msg.shape} = {msg}')

    sg = np.dot(keys.SK, G) % keys.q
    print(f'sg {sg.shape} = {sg}')

    div = np.rint((msg / sg))
    print(f'div = {div}')


    modes = np.unique(div)
    print(f'modes = {modes}')

    best_num = 0
    best_dist = float('inf')

    for mu in modes:
        dist = (msg - mu*sg) % keys.q
        dist = np.minimum(dist, keys.q - dist)
        #dist = np.linalg.norm(dist)
        sdist = np.dot(dist, dist)
        print(f'{mu} -> {dist} -> {sdist}')
        if sdist < best_dist:
            best_num = mu
            best_dist = sdist

    print(f'Deciphered -> {best_num}')
    return best_num
```

```
*******************
Ciphertext (13, 156) =
 [[ 371 2811 ... 1231  272]
 [1794 3130 ...  962  584]
 ...
 [1060 1770 ... 2364  398]
 [ 433 1317 ... 1726 1058]]

msg (156,) = [2696 1888 ... 3396 3289]

sg (156,) = [2030  569 ... 1024 2048]

div = [1 3 ... 3 2]

modes = [ 0  1  2  3  4  5  6  7  9 14 15 16 32
 33 34 35]

0 -> [ 795 1603 ...   95  202] -> 166703525

1 -> [ 666 1319 ... 1119 1241] -> 152180396

2 -> [1364  750 ... 1348  807] -> 162133138

3 -> [ 97 181 ... 324 636] -> 153786528

4 -> [1558  388 ...  700 1412] -> 161383785

5 -> [ 472  957 ... 1724   31] -> 144111628

6 -> [ 989 1526 ...  743 1474] -> 130962812

7 -> [1041 1396 ...  281  574] -> 150738087

9 -> [1610  258 ... 1162 1179] -> 156612370

14 -> [1287  904 ...  467  946] -> 155284358

15 -> [ 174  335 ... 1491  497] -> 147386745

16 -> [1635  234 ...  976 1551] -> 156735142

32 -> [ 574 1135 ... 1444  591] -> 156825956

33 -> [1456  566 ... 1023 1457] -> 154955287

34 -> [ 5  3 ...  1 14] -> 8229

35 -> [1466  572 ... 1025 1429] -> 154395660

Deciphered -> 34
```

Заключение

Удалось подтвердить корректность алгоритма из раздела 2, и работающий код для модификаций и экспериментов на языке Python.

Полученные в данной работе результаты инициируют следующие направления исследований:

- Расширенное тестирование для различных параметров — $log\beta$, $\log b$, $\log q$, n ...
- Проверка устойчивости работы алгоритма на длинных строках.
- Исследование производительности — теоретической и практической.
- Оптимизация получения более компактного НКА через производную Брозовского (См. [ORT]).
- Применение альтернативных GSW-схем шифрования (см. раздел 3.1) для этой задачи.
- Другие модификации алгоритма, направленные на решение этой задачи.

Список иллюстраций

2.1 Представление регулярных выражений в виде НКА 32
2.2 «NFAs for «(a*|b)») 33
2.3 «NFAs for «abc|d») 34
2.4 «NFAs for «(a|b)(b|c)*») 35
2.5 «NFAs for «(a|b)*c») 36
2.6 Зашифрованные матрицы перехода 37

Список алгоритмов

1 Использование гомоморфного шифрования с целочисленной арифметикой на Python . . . 10
2 Получение матриц переходов для каждого символа алфавита НКА . . . 15
3 НКА через матрицу переходов . . . 16
4 Семплирование «случайным шумом» 0-1 матрицы . . . 25
5 Генерация GGHLM-ключа . . . 26
6 GGHLM-шифрование векторов и неквадратных матриц . . . 27
7 GGHLM-шифрование квадратных матриц . . . 27
8 GGHLM-дешифрование неквадратных матриц . . . 28
9 GGHLM-дешифрование квадратных матриц . . . 29
10 Инициализация зашифрованного НКА-автомата . . . 30
11 Распознавание строки с зашифрованными переходами . . . 31
12 Переход по символу в зашифрованных НКА . . . 38
13 Генерация простого числа . . . 39
14 Генерация простого числа Софи Жермен . . . 40
15 Описание GSW-ключа . . . 40
16 Генерация GSW-ключа . . . 43
17 Матрица-генератор G для схемы GSW . . . 44
18 GSW-шифрование . . . 45
19 GSW-дешифрование . . . 46

Список литературы

[Ant96] Valentin Antimirov. “Partial derivatives of regular expressions and finite automaton constructions”. В: *Theoretical Computer Science* 155.2 (1996), с. 291—319. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(95)00182-4. URL: https://www.sciencedirect.com/science/article/pii/0304397595001824

[AP14] Jacob Alperin-Sheriff и Chris Peikert. “Faster Bootstrapping with Polynomial Error”. В: *Advances in Cryptology – CRYPTO 2014*. Под ред. Juan A. Garay и Rosario Gennaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, с. 297—314. ISBN: 978-3-662-44371-2.

[Bar+01] Boaz Barak и др. “On the (Im)possibility of Obfuscating Programs”. В: *Advances in Cryptology — CRYPTO 2001*. Под ред. Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, с. 1—18. ISBN: 978-3-540-44647-7.

[Dij+10] Marten van Dijk и др. “Fully Homomorphic Encryption over the Integers”. В: *Advances in Cryptology – EUROCRYPT 2010*. Под ред. Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, с. 24—43. ISBN: 978-3-642-13190-5.

[Gen+19] Nicholas Genise и др. *Homomorphic Encryption for Finite Automata*. Cryptology ePrint Archive, Report 2019/176. https://ia.cr/2019/176. 2019.

[GZ20] Steven D. Galbraith и Lukas Zobernig. “Obfuscating Finite Automata”. В: *Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers*. Под ред. Orr Dunkelman, Michael J. Jacobson

Jr. и Colin O'Flynn. Т. 12804. Lecture Notes in Computer Science. Springer, 2020, с. 90—114. DOI: 10.1007/978-3-030-81652-0_4. URL: https://doi.org/10.1007/978-3-030-81652-0%5C_4.

[HAO16] Ryo Hiromasa, Masayuki Abe и Tatsuaki Okamoto. "Packing Messages and Optimizing Bootstrapping in GSW-FHE". В: *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* 99-A.1 (2016), с. 73—82. DOI: 10.1587/transfun.E99.A.73. URL: https://doi.org/10.1587/transfun.E99.A.73.

[HMU07] John E. Hopcroft, Rajeev Motwani и Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007, с. 77. ISBN: 978-0-321-47617-3.

[ILL89] R. Impagliazzo, L. A. Levin и M. Luby. "Pseudo-Random Generation from One-Way Functions". В: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, с. 12—24. ISBN: 0897913078. DOI: 10.1145/73007.73009. URL: https://doi.org/10.1145/73007.73009.

[MP11] Daniele Micciancio и Chris Peikert. *Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller*. Cryptology ePrint Archive, Paper 2011/501. https://eprint.iacr.org/2011/501. 2011. URL: https://eprint.iacr.org/2011/501.

[ORT] Scott Owens, John Reppy и Aaron Turon. *Regular-expression derivatives reexamined*.

[RAD78] Ronald L. Rivest, Len Adleman и Michael L. Dertouzos. "On Data Banks and Privacy Homomorphisms". В: *Foundations of secure computatio* (1978), с. 169—177.

[SJ05] William Stein и David Joyner. "SAGE: system for algebra and geometry experimentation". В: *ACM SIGSAM Bulletin* 39.2 (2005), с. 61—64. URL: http://modular.math.washington.edu/sage/misc/sage_sigsam_updated.pdf.

[Дан21] Иван Тунёв Даниил Корогодский. “Обзор и модификация существующих методов защищенных облачных вычислений с применением обфускации и гомоморфного шифрования”. В: *Труды 64-й Всероссийской научной конференции МФТИ*. 2021. URL: https://conf.mipt.ru/articles/9184.

[Ири20] Александр Шокуров и Ирина Абрамова и Николай Варновский и Владимир Захаров. “О возможности стойкой обфускации программ в одной модели облачных вычислений”. В: *Труды Института системного программирования РАН* 31.6 (2020). ISSN: 2220-6426. URL: https://ispranproceedings.elpub.ru/jour/article/view/1261.

[Шок07] А. В. Шокуров. *Решетки, алгоритмы и современная криптография*. 2007. URL: https://discopal.ispras.ru/Lattices-and-cryptography.