

# Реализация алгоритмов шифрования и расшифрования для криптосистемы HFE

## Параллельные вычисления

ИУ8-112

МГТУ им. Н.Э. Баумана

30 ноября 2025 г.

# Что такое HFE?

- **HFE (Hidden Field Equations)** — криптографическая система с открытым ключом
- Предложена Жаком Патарином в 1996 году
- Основана на многочленных уравнениях над конечными полями
- Использует скрытую структуру конечного поля для обеспечения безопасности

## Основная идея

Преобразование сложных уравнений над конечным полем в систему квадратичных уравнений над GF(2)

## Конечное поле $GF(2^n)$

- Поле размерности  $n$
- $2^n$  элементов
- Операции: сложение (XOR), умножение по модулю неприводимого многочлена
- Неприводимый многочлен:  $x^n + \dots + 1$

## HFE многочлен

- $P(x) = \sum_{i \leq j \leq d} a_{ij} \cdot x^{2^i + 2^j}$
- Степень  $d$  ограничена
- Коэффициенты  $a_{ij} \in GF(2^n)$

### Важно

Для расшифрования необходимо решить уравнение  $P(x) = y$ , что требует знания секретной структуры

Открытый текст  $\rightarrow S \rightarrow P(x) \rightarrow T \rightarrow$  Шифротекст



## Алгоритм шифрования:

- ① Применение секретного аффинного преобразования  $S$ :  $x' = S_1 \cdot x + S_0$
- ② Вычисление HFE многочлена:  $y' = P(x')$
- ③ Применение секретного аффинного преобразования  $T$ :  $y = T_1 \cdot y' + T_0$

- ➊ Обратное преобразование  $T$ :  $y' = T_1^{-1} \cdot (y - T_0)$
- ➋ Решение HFE уравнения: найти  $x'$  такой, что  $P(x') = y'$ 
  - Требует перебора или знания структуры многочлена
  - В нашей реализации используется перебор всех возможных значений
- ➌ Обратное преобразование  $S$ :  $x = S_1^{-1} \cdot (x' - S_0)$

## Вычислительная сложность

Решение HFE уравнения — самая затратная операция, требует  $O(2^n)$  операций в худшем случае

## Основные компоненты

- GF2n — класс для работы с конечным полем  $GF(2^n)$
- HFEBase — базовая реализация HFE
- Последовательная обработка данных

## Последовательность операций:

- ① Инициализация: генерация ключей (матрицы  $S_1, S_0, T_1, T_0$ )
- ② Для каждого байта данных:
  - Преобразование байта в вектор бит
  - Применение  $S$ , вычисление  $P(x)$ , применение  $T$
  - Преобразование результата обратно в байт

# Что выполняется последовательно?

```
def encrypt_block(self, data: bytes) -> bytes:
    result = []
    for byte_val in data: # Sequential processing
        # Convert byte to bit vector
        plaintext = [(byte_val >> i) & 1
                     for i in range(self.n)]

        # Encrypt single byte
        ciphertext = self.encrypt(plaintext)

        # Convert back to byte
        cipher_byte = sum(bit << i
                          for i, bit in enumerate(ciphertext))
        result.append(cipher_byte)

    return bytes(result)
```

## Узкое место

Каждый байт обрабатывается независимо, но выполнение происходит последовательно на одном ядре CPU

## Шифрование одного байта:

- Аффинное преобразование  $S$ :  $O(n^2)$
- HFE многочлен:  $O(d)$
- Аффинное преобразование  $T$ :  $O(n^2)$
- Итого:  $O(n^2 + d)$

## Расшифрование одного байта:

- Обратное  $T$ :  $O(n^2)$
- Решение HFE:  $O(2^n)$  – перебор!
- Обратное  $S$ :  $O(n^2)$
- Итого:  $O(2^n)$

Для  $N$  байт:  $O(N \cdot (n^2 + d))$  для шифрования,  $O(N \cdot 2^n)$  для расшифрования

## Используемая технология

`multiprocessing` — создание нескольких процессов Python для параллельной обработки

### Стратегия распараллеливания:

- ➊ Разделение данных на `chunks` (части)
- ➋ Каждый процесс обрабатывает свой `chunk` независимо
- ➌ Объединение результатов

Процесс 1	Процесс 2	Процесс 3	Процесс 4
Байты 0-255	Байты 256-511	Байты 512-767	Байты 768-1023

# Что конкретно распараллелено?

```
def encrypt_block(self, data: bytes) -> bytes:
    # Convert all bytes to vectors
    plaintexts = [[(byte_val >> i) & 1
                  for i in range(self.n)]
                  for byte_val in data]

    # Split into chunks
    chunk_size = len(plaintexts) // num_processes
    chunks = [plaintexts[i:i + chunk_size]
              for i in range(0, len(plaintexts), chunk_size)]

    # PARALLEL PROCESSING
    with Pool(processes=num_processes) as pool:
        results = pool.map(_encrypt_chunk_worker, chunks)

    # Merge results
    ciphertexts = []
    for chunk_result in results:
        ciphertexts.extend(chunk_result)
```

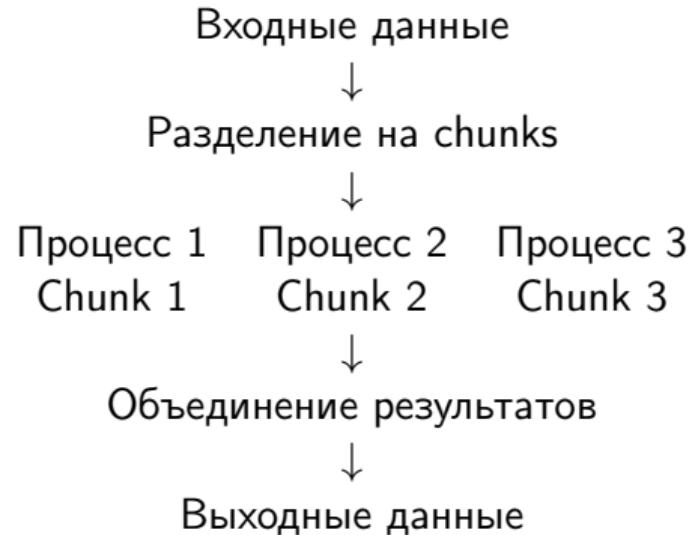
## Распараллеленные операции

- Обработка независимых байтов — каждый процесс шифрует свой набор байтов
- Аффинные преобразования — выполняются параллельно для разных данных
- Вычисление HFE многочлена — параллельно для разных входных значений
- Решение HFE уравнений — параллельно при расшифровании

## Что HFE распараллелено

- Генерация ключей (выполняется один раз)
- Преобразование данных между форматами (вектор  $\leftrightarrow$  байт)
- Объединение результатов (последовательное)

# Архитектура CPU-параллельной версии



## Преимущества:

- Использование всех ядер CPU
- Линейное ускорение (до количества ядер)
- Независимость процессов (изоляция ошибок)

# Оценка производительности CPU-версии

## Теоретическое ускорение

Для  $P$  процессов: ускорение  $\approx P \times$  (с учетом накладных расходов)

## Накладные расходы:

- Создание процессов:  $\sim 10 - 50$  мс
- Передача данных между процессами
- Синхронизация и объединение результатов

## Эффективность:

- Для больших данных ( $> 1$  КБ): эффективность  $\approx 80 - 95\%$
- Для малых данных ( $< 100$  байт): накладные расходы могут превысить выгоду

## Используемая технология

Numba CUDA — компиляция Python кода в CUDA kernels для выполнения на GPU

## Архитектура GPU:

- Тысячи потоков (2048-8192+)
- SIMD (Single Instruction Multiple Data)
- Высокая пропускная способность для параллельных операций
- Медленная передача данных CPU ↔ GPU

## Особенность

GPU оптимален для больших объемов данных и однотипных операций

# Что конкретно распараллелено на GPU?

```
# CUDA kernel for affine transformation
@cuda_jit
def _gpu_affine_transform(input_data, A, b, n, output):
    idx = cuda.grid(1) # Thread index
    if idx < len(input_data):
        for i in range(n):
            sum_val = 0
            for j in range(n):
                sum_val += (A[i, j] * input_data[idx * n + j]) % 2
            output[idx * n + i] = (sum_val + b[i]) % 2

# CUDA kernel for HFE polynomial
@cuda_jit
def _gpu_hfe_polynomial(x, n, d, result):
    idx = cuda.grid(1)
    if idx < len(x):
        res = 0
        for i in range(1, d + 1):
            power = 1 << i
            if power < (1 << n):
                # Compute x^power
                res += compute_power(x[idx], power, n)
        result[idx] = res
```

# Распараллеленные операции на GPU

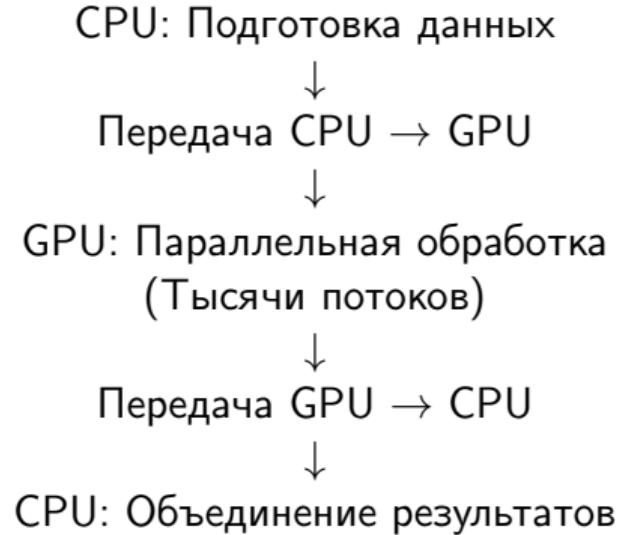
Полностью распараллелено:

- Аффинные преобразования — каждый поток обрабатывает один элемент данных
- Вычисление HFE многочлена — параллельно для всех входных значений
- Операции над полями  $GF(2^n)$  — умножение, сложение

Частично распараллелено:

- Решение HFE уравнений — использует гибридный подход (GPU + CPU)
  - Перебор выполняется на CPU из-за сложности ветвления
  - Аффинные преобразования — на GPU

# Архитектура GPU-параллельной версии



## Этапы обработки:

- ① Преобразование данных на CPU
- ② Копирование на GPU (`cuda.to_device`)
- ③ Параллельное выполнение CUDA kernels
- ④ Копирование результатов обратно на CPU

# CUDA Kernel функции

## Реализованные kernels:

- `_gpu_affine_transform` — аффинное преобразование
- `_gpu_hfe_polynomial` — вычисление HFE многочлена
- `_gpu_multiply_gf2n` — умножение в  $GF(2^n)$
- `_gpu_power_gf2n` — возведение в степень

## Конфигурация выполнения:

- `threads_per_block = 256` (настраивается)
- `blocks_per_grid = (n_bytes + 255) / 256`
- Каждый поток обрабатывает один байт данных

## Ограничения

- Требуется NVIDIA GPU с CUDA
- Накладные расходы на передачу данных
- Неэффективно для малых объемов данных

# Оценка производительности GPU-версии

## Теоретическое ускорение

Для больших данных ( $> 10$  МБ): ускорение  $\approx 10 - 100 \times$  по сравнению с CPU

## Факторы производительности:

- **Размер данных:** чем больше, тем эффективнее
- **Пропускная способность памяти:** ограничивающий фактор
- **Вычислительная мощность:** тысячи потоков работают параллельно

## Накладные расходы:

- Передача данных CPU  $\leftrightarrow$  GPU:  $\sim 1 - 10$  мс
- Запуск kernel:  $\sim 0.1 - 1$  мс
- Для данных  $< 100$  КБ накладные расходы могут превысить выгоду

# Сравнение подходов

Характеристика	Обычная	CPU	GPU
Ядра/Потоки	1	4-16	2048-8192+
Накладные расходы	Минимальные	Средние	Высокие
Оптимальный размер данных	Любой	> 1 KB	> 10 MB
Ускорение (теоретическое)	1×	4-16×	10-100×
Сложность реализации	Низкая	Средняя	Высокая
Требования	-	Многоядерный CPU	NVIDIA GPU + CUDA

## Вывод

Выбор реализации зависит от размера данных и доступного оборудования

# Что распараллелено в каждой версии?

## Обычная

- Ничего
- Последовательная обработка байтов
- Одно ядро CPU

## CPU

- Обработка независимых байтов
- Аффинные преобразования
- HFE многочлен
- Решение HFE уравнений

## GPU

- Аффинные преобразования (kernel)
- HFE многочлен (kernel)
- Операции над полями (kernel)
- Решение HFE (гибрид)

Все версии распараллеливают обработку независимых блоков данных

# График производительности (теоретический)

## Зависимость времени выполнения от размера данных

Размер данных	Обычная	CPU	GPU
< 1 KB	Быстро	Средне	Медленно
1 KB - 10 MB	Медленно	Быстро	Средне
> 10 MB	Очень медленно	Медленно	Очень быстро

## Оптимальная точка перехода

Для данных > 1 KB CPU-версия начинает превосходить обычную

Для данных > 10 MB GPU-версия становится оптимальной

## Наблюдения:

- Для малых данных: обычная версия быстрее (нет накладных расходов)
- Для средних данных: CPU-версия оптимальна
- Для больших данных: GPU-версия значительно быстрее

## HFE криптосистема

- Основана на многочленных уравнениях над  $GF(2^n)$
- Использует аффинные преобразования для скрытия структуры
- Расшифрование требует решения HFE уравнений

## Распараллеливание

- **CPU:** Распараллелена обработка независимых байтов через multiprocessing
- **GPU:** Распараллелены аффинные преобразования и вычисление HFE многочлена через CUDA kernels
- Обе версии эффективны для больших объемов данных

# Рекомендации по использованию

## Используйте обычную версию:

- Малые данные (< 1 KB)
- Отладка
- Простые задачи

## Используйте CPU-версию:

- Средние данные (1 KB - 100 MB)
- Многоядерный CPU
- Баланс производительности

## Используйте GPU-версию:

- Большие данные (> 10 MB)
- NVIDIA GPU с CUDA
- Максимальная производительность