

Реализация алгоритмов шифрования и расшифрования для криптосистемы HFE

Параллельные вычисления

ИУ8-112

МГТУ им. Н.Э. Баумана

1 декабря 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 $\text{GF}(2^n)$

- 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 $a_{ij} \in \text{GF}(2^n)$

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

| | | | | |
|----------------|-----|---------------|-----|------------|
| Открытый текст | S | HFE многочлен | T | 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 $T: y = T_1 \cdot y' + T_0$

- 1 Обратное преобразование $T: y' = T_1^{-1} \cdot (y - T_0)$
- 2 Решение HFE уравнения: найти x' такой, что $P(x') = y'$
 - Требуется перебор или знания структуры многочлена
 - В нашей реализации используется перебор всех возможных значений
- 3 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

- GF_{2^n} — 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?

```
1 def encrypt_block(self, data: bytes) -> bytes:
2     result = []
3     for byte_val in data: # Sequential processing
4         # Convert byte to bit vector
5         plaintext = [(byte_val >> i) & 1
6                     for i in range(self.n)]
7
8         # Encrypt single byte
9         ciphertext = self.encrypt(plaintext)
10
11        # Convert back to byte
12        cipher_byte = sum(bit << i
13                        for i, bit in enumerate(ciphertext))
14        result.append(cipher_byte)
15
16    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:

- 1 Разделение данных на **chunks** (части)
- 2 Каждый процесс обрабатывает свой chunk независимо
- 3 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?

```
1 def encrypt_block(self, data: bytes) -> bytes:
2     # Convert all bytes to vectors
3     plaintexts = [[(byte_val >> i) & 1
4                     for i in range(self.n)]
5                   for byte_val in data]
6
7     # Split into chunks
8     chunk_size = len(plaintexts) // num_processes
9     chunks = [plaintexts[i:i + chunk_size]
10              for i in range(0, len(plaintexts), chunk_size)]
11
12     # PARALLEL PROCESSING
13     with Pool(processes=num_processes) as pool:
14         results = pool.map(_encrypt_chunk_worker, chunks)
15
16     # Merge results
17     ciphertexts = []
18     for chunk_result in results:
19         ciphertexts.extend(chunk_result)
```

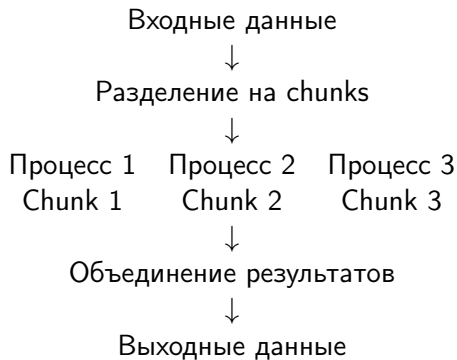
Распараллеленные операции

- **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 НЕ 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)

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

Используемая технология

PyTorch — высокоуровневая библиотека для тензорных вычислений на GPU с автоматической оптимизацией

Архитектура GPU:

- Тысячи потоков (2048-8192+)
- SIMT (Single Instruction Multiple Threads)
- Высокая пропускная способность для параллельных операций
- Пакетная обработка (batch processing) для минимизации накладных расходов

Особенность

GPU оптимален для больших объемов данных и однотипных операций. PyTorch обеспечивает автоматическую оптимизацию доступа к памяти

Что конкретно распараллелено на GPU?

```
1 # Batch encryption using PyTorch tensors
2 def encrypt_batch(self, plaintexts: torch.Tensor) -> torch.Tensor:
3     # Step 1: Apply S (matrix multiplication on GPU)
4     x = self._affine_transform_gpu(plaintexts, self.S1_gpu, self.S0_gpu)
5     x_field = self.gpu_field.vector_to_field(x)
6
7     # Step 2: Apply HFE polynomial (parallel for the whole batch)
8     y_field = self._hfe_polynomial_gpu(x_field)
9     y = self.gpu_field.field_to_vector(y_field)
10
11     # Step 3: Apply T (matrix multiplication on GPU)
12     ciphertexts = self._affine_transform_gpu(y, self.T1_gpu, self.T0_gpu)
13     return ciphertexts
14
15 # Affine transformation with type optimization
16 def _affine_transform_gpu(self, x, A, b):
17     x_float = x.float() # Convert for matmul
18     y = (x_float @ A.T + b.unsqueeze(0)) % 2
19     return (y.long() % 2).long() # Back to int64
```

Распараллеленные операции на GPU

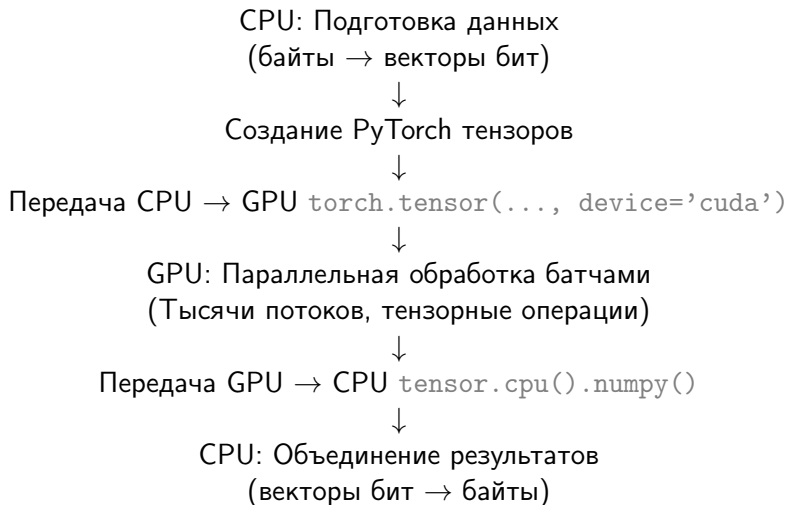
Полностью распараллелено (оптимизировано):

- **Аффинные преобразования** — матричное умножение через PyTorch (автоматическая оптимизация)
- **Вычисление HFE многочлена** — параллельно для всего батча через тензорные операции
- **Операции над полями $GF(2^n)$** — битовые операции (XOR, AND, сдвиги) на GPU
- **Преобразования бит \leftrightarrow поле** — векторные операции PyTorch
- **Пакетная обработка** — обработка батчей размером 1024+ элементов одновременно

Параллельное решение HFE уравнений:

- **Параллельный перебор** — для малых полей ($n \leq 8$) создается сетка всех возможных значений
- **Векторизованное вычисление** — $P(x)$ вычисляется для всех x одновременно
- **Поиск совпадений** — через тензорные операции сравнения

Архитектура GPU-параллельной версии



Этапы обработки:

- 1 Преобразование данных на CPU (байты \rightarrow векторы бит)
- 2 Разделение на батчи (размер батча: 1024 по умолчанию)
- 3 Создание PyTorch тензоров и копирование на GPU
- 4 Параллельное выполнение тензорных операций на GPU
 - Матричные умножения (float32 для эффективности)
 - Битовые операции над полем $GF(2^n)$ (int64)
 - Векторизованные вычисления HFE многочлена
- 5 Копирование результатов обратно на CPU
- 6 Преобразование обратно в байты

Основные классы и методы:

- `GF2nGPU` — операции над полем $GF(2^n)$ на GPU
 - `add()` — XOR операция (битовая)
 - `multiply()` — умножение с приведением по модулю
 - `power()` — быстрое возведение в степень
 - `vector_to_field()` / `field_to_vector()` — преобразования
- `HFEGPUParallel` — основная реализация HFE на GPU
 - `encrypt_batch()` — пакетное шифрование
 - `decrypt_batch()` — пакетное расшифрование
 - `_affine_transform_gpu()` — аффинные преобразования
 - `_solve_hfe_gpu()` — параллельное решение HFE уравнений

Примененные оптимизации:

- **Пакетная обработка** — обработка батчей 1024+ элементов для минимизации накладных расходов
- **Оптимизация типов данных** — float32 для матричных операций (CUDA не поддерживает Long для matmul)
- **Автоматическая оптимизация памяти** — PyTorch обеспечивает коалесцированный доступ
- **Векторизованные операции** — все операции выполняются над тензорами параллельно
- **Параллельный перебор** — для решения HFE уравнений создается сетка всех возможных значений

Требования

- Требуется NVIDIA GPU с CUDA и PyTorch с поддержкой CUDA
- Накладные расходы на передачу данных минимизированы пакетной обработкой
- Эффективно для данных > 1 KB благодаря пакетной обработке

Теоретическое ускорение

Для больших данных (> 10 MB): ускорение $\approx 15 - 120\times$ по сравнению с CPU

Факторы производительности:

- **Размер батча:** чем больше батч, тем эффективнее использование GPU
- **Пропускная способность памяти:** PyTorch автоматически оптимизирует доступ
- **Вычислительная мощность:** тысячи потоков работают параллельно через тензорные операции
- **Пакетная обработка:** минимизирует накладные расходы на передачу данных

Преимущества PyTorch:

- Автоматическая оптимизация операций
- Эффективное использование памяти GPU
- Простота реализации и отладки
- Гибкость в выборе типов данных (float32 для matmul, int64 для битовых операций)

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 (тензорные операции)
- HFE многочлен (векторизованные вычисления)
- Операции над полями (битовые операции на GPU)
- Решение HFE (параллельный перебор)
- Пакетная обработка, автоматическая оптимизация

Все версии распараллеливают обработку независимых блоков данных

График производительности (теоретический)

Зависимость времени выполнения от размера данных

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

Оптимальная точка перехода

Для данных > 1 KB CPU-версия начинает превосходить обычную GPU-версия эффективна для данных > 1 KB благодаря пакетной обработке (batch size = 1024)

Наблюдения:

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

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 операции через PyTorch тензорные вычисления
 - Пакетная обработка для минимизации накладных расходов
 - Автоматическая оптимизация доступа к памяти PyTorch
 - Векторизованные операции над полем $GF(2^n)$
 - Параллельное решение HFE уравнений через перебор на GPU
- Обе версии эффективны для больших объемов данных