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)

Математические основы

Конечное поле GF(2^n)

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

HFE многочлен

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

Важно

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

Структура HFE криптосистемы

$$\text{Открытый текст} \to S \to P(x) \to T \to \text{Шифротекст}$$

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: $y = T_1 \cdot y' + T_0$

Алгоритм расшифрования

1. **Обратное преобразование** T: $y' = T_1^{-1} \cdot (y - T_0)$
2. **Решение HFE уравнения**: найти x' такой, что $P(x') = y'$
   - Требует перебора 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

Архитектура обычной реализации

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?

```python
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

Вычислительная сложность

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

Подход к CPU-параллелизации

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?

```
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)
```

Детали распараллеливания

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

Входные данные

$\downarrow$

Разделение на chunks

$\downarrow$

Процесс 1	Процесс 2	Процесс 3
Chunk 1	Chunk 2	Chunk 3

$\downarrow$

Объединение результатов

$\downarrow$

Выходные данные

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

Подход к GPU-параллелизации

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 $\leftrightarrow$ GPU

Особенность

GPU оптимален для больших объемов данных и однотипных операций

Что конкретно распараллелено на GPU?

```python
# Optimized CUDA kernel for affine transformation
@cuda_jit
def _gpu_affine_transform(input_data, A, b, n, output, n_bytes, total_threads):
    idx = cuda.grid(1)  # Thread index
    # Grid-stride loop: process multiple elements
    byte_idx = idx
    while byte_idx < n_bytes:
        for i in range(n): # Memory coalescing
            sum_val = 0
            for j in range(n):
                if A[i, j] == 1:
                    sum_val ^= input_data[byte_idx * n + j]
            output[byte_idx * n + i] = (sum_val ^ b[i]) % 2
        byte_idx += total_threads
# Optimized CUDA kernel for HFE polynomial
@cuda_jit
def _gpu_hfe_polynomial(x, n, d, result, total_threads):
    idx = cuda.grid(1)
    elem_idx = idx
    while elem_idx < len(x):
        res = 0
        for i in range(1, d + 1):
            power = 1 << i
            if power < (1 << n):
                # Uses optimized power function
                res ^= _gpu_power_gf2n_device(x[elem_idx], power, n, irreducible)
        result[elem_idx] = res
        elem_idx += total_threads
```

Распараллеленные операции на GPU

Полностью распараллелено (оптимизировано):

- **Аффинные преобразования** — grid-stride loop, каждый поток обрабатывает несколько элементов
- **Вычисление HFE многочлена** — параллельно для всех входных значений с оптимизированным умножением
- **Операции над полями GF(2^n)** — умножение со специализацией для n=8, возведение в степень
- **Преобразования бит<->поле** — коалесцированный доступ к памяти

Частично распараллелено:

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

Архитектура GPU-параллельной версии

CPU: Подготовка данных

$\downarrow$

Передача CPU $\rightarrow$ GPU

$\downarrow$

GPU: Параллельная обработка
(Тысячи потоков)

$\downarrow$

Передача GPU $\rightarrow$ CPU

$\downarrow$

CPU: Объединение результатов

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` — оптимизированное аффинное преобразование
- `_gpu_hfe_polynomial` — вычисление HFE многочлена
- `_gpu_bits_to_field` / `_gpu_field_to_bits` — преобразования
- `_gpu_multiply_gf2n_device` — оптимизированное умножение в $GF(2^n)$
- `_gpu_power_gf2n_device` — оптимизированное возведение в степень

Оптимизации

Все kernels используют коалесцированный доступ к памяти и grid-stride loop pattern

CUDA Kernel функции

Конфигурация выполнения (оптимизированная):

- `threads_per_block = 64` (по умолчанию, оптимизировано)
- `blocks_per_grid = max(128-256, (n_bytes + 63) / 64)`
- Grid-stride loop: каждый поток обрабатывает несколько байтов
- Адаптивная конфигурация: 256 блоков для данных < 1KB

CUDA Kernel функции

Примененные оптимизации

- Memory Coalescing — коалесцированный доступ к памяти
- Grid-stride Loop — улучшенная утилизация GPU
- Специализация для n=8 — ускорение на 20-30%
- Адаптивная конфигурация — оптимальное количество блоков

Требования

- Требуется NVIDIA GPU с CUDA
- Накладные расходы на передачу данных (минимизированы)
- Эффективно для данных > 1 KB (благодаря оптимизациям)

Оценка производительности GPU-версии

Теоретическое ускорение (после оптимизаций)

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

Факторы производительности:

- **Размер данных:** чем больше, тем эффективнее
- **Пропускная способность памяти:** улучшена на 15-25% благодаря coalescing
- **Вычислительная мощность:** тысячи потоков работают параллельно
- **Утилизация GPU:** улучшена для малых данных благодаря grid-stride loop

Результаты оптимизаций:

- Устранены предупреждения о низкой занятости GPU
- Ускорение операций в $GF(2^8)$ на 20-30%
- Улучшена эффективность для данных $>$ 1 KB (ранее $>$ 100 KB)

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)	$1\times$	$4\text{-}16\times$	$10\text{-}100\times$
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 (оптимизированная)

- Аффинные преобразования (оптимизированный kernel)
- HFE многочлен (оптимизированный kernel)
- Операции над полями (специализация для n=8)
- Решение HFE (гибрид)
- Memory coalescing, grid-stride loop

График производительности (теоретический)

Зависимость времени выполнения от размера данных

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

Оптимальная точка перехода

Для данных $>$ 1 KB CPU-версия начинает превосходить обычную
Благодаря оптимизациям, GPU-версия эффективна уже для данных $>$ 1 KB (ранее требовалось $>$ 10 MB)

Наблюдения:

- Для малых данных: обычная версия быстрее (нет накладных расходов)
- Для средних данных: CPU-версия оптимальна
- Для больших данных: GPU-версия значительно быстрее (улучшено на 20-30%

Ключевые моменты

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 оптимизированные CUDA kernels
    - Memory coalescing для лучшей пропускной способности
    - Grid-stride loop для улучшенной утилизации
    - Специализация для GF(2^8) для ускорения на 20-30%
- Обе версии эффективны для больших объемов данных