

Руководство по реализации криптографии на эллиптических кривых

Санан Корняков

2024

Содержание

| | | |
|----------|-----------------------------|-----------|
| 1 | Введение | 3 |
| 1.1 | Условия игры | 3 |
| 1.2 | База | 3 |
| 1.3 | Постановка задачи | 4 |
| 2 | Длинная арифметика | 4 |
| 2.1 | Каркас | 4 |
| 2.2 | Методы | 5 |
| 2.3 | Главный тип | 17 |
| 3 | Поле | 18 |
| 3.1 | Элемент поля | 18 |
| 3.1.1 | Каркас | 18 |
| 3.1.2 | Методы | 18 |
| 3.2 | Поле | 20 |
| 4 | Эллиптическая кривая | 21 |
| | Список литературы | 22 |

Аннотация

Работа является пошаговым руководством по реализации криптографии на эллиптических кривых. Реализованы объекты длинной арифметики, полей и эллиптических кривых. Изучены и имплементированы алгоритмы шифрования и дешифрования, электронной цифровой подписи, подсчёт количества точек на эллиптической кривой, быстрого умножения и деления длинных чисел. Протестированы объекты и алгоритмы по скорости, сравнивая с готовыми решениями. Руководство параллельно с имплементацией объясняет и рассказывает, что и зачем было реализовано.

Ключевые слова: эллиптические кривые, шифрование и дешифрование, криптография, ECDSA, ECC, длинная арифметика, FFT, C++, конечные поля, оптимизация, Schoof's algorithm

1 Введение

Современная криптография с нынешними вычислительными мощностями требует значительных усилий в шифровке сообщений, и шифрование с помощью эллиптических кривых - один из мощнейших инструментов. Но доступных и полных объяснений от начала до конца по шифрованию на них ничтожно мало, поэтому я решил сделать руководство для людей, которые хотят ознакомиться с данным видом криптографии.

1.1 Условия игры

Если вы искали данное руководство, то скорее всего где-то слышали/читали об эллиптических кривых и о возможности криптографии на них, поэтому я рассчитываю на базовое понимание математики и алгоритмов.

Здесь не будет дотошного доказательства теорем или строгости в описании математических объектов — в первую очередь акцент делается именно на имплементации (на языке C++). Данный язык был выбран в качестве общеизвестного языка среди программистов. Выберем C++20 для удобного использования шаблонов.

В данном руководстве мы будем стараться использовать как можно меньше готовых библиотек, чтобы не было огромных black box-ов в нашем коде. Это улучшит понимание и возможности алгоритмов.

1.2 База

Знаменитая формула

$$y^2 = x^3 + ax + b$$

обычно является самой первой, которую вы увидите при описании криптографии эллиптических кривых. Появляется несколько вопросов:

- Что такое x, y, a, b ? Где лежат данные числа?

Данные числа являются элементами некоего поля \mathbb{F} , над которым построена эллиптическая кривая, характеристики больше 3 (забьём на последние слова, так как мы будем работать с полями достаточно больших характеристик). Поле поддерживает все стандартные математические операции: сложение, вычитание, деление на ненулевой элемент, умножение, поэтому можно пока считать его \mathbb{R} .

- Что такое эллиптическая кривая?

Это группа точек в \mathbb{F}^2 , координаты которых удовлетворяют данному уравнению, и ещё точка бесконечности \mathcal{O} , которая является своеобразным нулём группы. Сложение в группе происходит по специальным формулам на координаты, которые будут рассмотрены позже. Умножение точки на натуральное число приравнивается к сложению точки с собой это число раз.

- Как это используют для шифрования?

Обычно выбирается эллиптическая кривая \mathbb{E} над неким полем \mathbb{F} , точка P на ней и производится умножение точки на натуральное число k . Криптографическая стойкость достигается сложностью нахождения числа k по точкам P и kP .

- Чем это лучше других методов шифрования?

Тем, что данный способ шифрования можно реализовать так, что он будет выполняться быстрее других алгоритмов при аналогичной задаче и данных. Также, для одинаковых показателей криптографической стойкости, криптография на эллиптических кривых требует ключей (чисел для шифрования) меньшей длины, чем другие алгоритмы.

1.3 Постановка задачи

Начитавшись статей на хабре, мы воодушевились и решили написать свою криптографию на эллиптических кривых. Сначала надо определить, какие объекты нам надо реализовать:

- Нам надо реализовать эллиптическую кривую. Но эллиптическая кривая никто без поля, значит нам надо реализовать поле.
- Так как поля бывают бесконечными, а мы работаем на компьютере с числами, то ограничимся на простые поля \mathbb{F}_p , которые представим в виде вычетов по простому модулю p . Но этот простой модуль и числа в поле надо представить в виде целых чисел, а в криптографии обычно используются числа из более чем 200 битов. Целые числа такого размера не поддерживаются языком C++, поэтому нам надо реализовать класс целых чисел и длинную арифметику на них.

Итого 3 объекта: целые числа, поле, эллиптическая кривая. Приступим наконец к реализации!

2 Длинная арифметика

Если не хотите запариваться и сделать основную рабочую лошадку блэббоксом, то можно просто установить длинку `boost::multiprecision` и скипнуть данную часть.

Так как мы хотим реализовать вычеты по большому модулю, то достаточно реализовать беззнаковые длинные целые числа.

Основной приём для имплементации длинной арифметики - хранение чисел в основании 2^{32} или 2^{64} . То есть просто массив из целых чисел, которые представляют цифры данного числа в соответствующих основаниях. Есть несколько видов данного представления:

1. Количество цифр в числе меняется в зависимости от размера числа. Нет математического ограничения длины числа, только аппаратное.
2. Количество цифр в числе фиксировано и не меняется от числа.

Последний вариант можно видеть, например, в типе `uint64_t` — присутствуют сразу все 64 бита не зависимо от содержащихся данных. Данный вид целых чисел является наиболее удобным в реализации и использовании по назначению, поэтому будем имплементировать его.

2.1 Каркас

Так как количество бит в числе может разительно отличаться от задачи к задаче, то общим решением будет создать шаблонный класс по количеству содержащихся в нём бит:

```
1 template<size_t c_bits>
2 class uint_t {
3 };
```

Теперь надо определиться с представлением цифр в нашем классе. Из-за того, что для алгоритма деления, который будет позже, потребуется деление по две цифры, то возьмём за цифру `uint32_t`, чтобы можно было спокойно делить в `uint64_t`.

С помощью `constexpr` определим размеры и длину необходимого массива. Так как длина не изменяется во время жизни объекта, то возьмём `std::array` за контейнер. Он будет гарантировать, что длина массива сохраняет свой инвариант. Итого получилось:

```
1 template<size_t c_bits>
2 class uint_t {
3     using digit_t = uint32_t;
4     using double_digit_t = uint64_t;
5
6     static constexpr size_t c_bits_in_byte = 8;
7     static constexpr size_t c_digit_size = sizeof(digit_t) * c_bits_in_byte;
8     static constexpr size_t c_digit_number = c_bits / c_digit_size;
9     static constexpr size_t c_double_digit_size = sizeof(double_digit_t) * c_bits_in_byte;
10    static constexpr size_t c_double_digit_number = c_bits / c_double_digit_size;
11
12    template<size_t V>
```

```

13     friend class uint_t;
14
15     using digits = std::array<digit_t, c_digit_number>;
16     digits m_digits = {};
17 };

```

Дали псевдонимы используемым типам, чтобы улучшить читаемость и не менять все типы, если вдруг захотим использовать за цифру `uint16_t` и какой-то другой тип. В `m_digits` храним число в основании 2^{32} в little-endian. 12-13 строчкой мы подружили все шаблоны друг с другом для общего взаимодействия.

2.2 Методы

Теперь мы хотим как-то общаться с нашими данными. Желательно имплементировать все операции, которые можно применять к обычным типам, таким как `uint32_t`, `uint64_t`.

- Конструирование: Есть несколько сценариев:

1. Хотим сконструироваться от целочисленного типа. Заметим, что количество бит в нём может быть как больше 32, так и меньше, поэтому ограничиться назначением `m_blocks[0]` нельзя. Тогда определим вспомогательный приватный метод, который будет определять через концепты отношения количества бит:

```

1  template<typename From, typename To>
2  concept is_upcastable_to = sizeof(From) <= sizeof(To) && is_convertible_to<From, To>;
3
4  template<typename From, typename To>
5  concept is_downcastable_to = sizeof(From) > sizeof(To) && is_convertible_to<From, To>;
6
7  template<typename T>
8  requires std::numeric_limits<T>::is_integer && is_upcastable_to<T, digit_t>
9  static constexpr digits split_into_digits(T value) {
10     return {static_cast<digit_t>(value)};
11 }
12
13 template<typename T>
14 requires std::numeric_limits<T>::is_integer && is_downcastable_to<T, digit_t>
15 static constexpr digits split_into_digits(T value) {
16     digits result = {};
17
18     for (size_t i = 0; i < c_digit_number; ++i) {
19         result[i] = static_cast<digit_t>(value);
20         value >>= c_digit_size;
21
22         if (value == 0) {
23             break;
24         }
25     }
26
27     return result;
28 }

```

2. Хотим сконструироваться от контейнера целых чисел, например как наш `uint_t`. Для этого пишем концепт, который определяет, что данный тип действительно является контейнером целых чисел, и копируем его данные:

```

1  template<typename Container, typename T>
2  concept is_convertible_container = requires(Container t, size_t i) {
3      { t[i] } -> is_convertible_to<T>;
4      { t.size() } -> std::same_as<size_t>;
5  };
6
7  template<typename T>
8  requires is_convertible_container<T, digit_t> || requires(T x) {
9      { uint_t {x} } -> std::same_as<T>;
10 }
11
12 static constexpr digits split_into_digits(const T& other) {
13     const size_t min_size = std::min(size(), other.size());
14     digits result = {};

```

```

14     for (size_t i = 0; i < min_size; i++) {
15         result[i] = static_cast<digit_t>(other[i]);
16     }
17
18     return result;
19 }

```

3. Хотим сконструироваться от строк (С-строк). Действительно, это единственный удобный способ задать необходимое нам число с количеством бит больше чем у `size_t`.

Для этого нам нужно сначала написать парсер строки, которая представляет число в двоичном, шестнадцатеричном и десятичном форматах. Так как мы хотим использовать методы `uint_t`, то нам нужно парсить в любой класс, который удовлетворяет критериям целочисленных классов:

```

1  template<typename T>
2  concept is_integral = std::is_integral_v<T> || requires(T t, T* p, void (*f)(T)) {
3      f(0);
4      p + t;
5  };
6  template<typename T>
7  requires is_integral<T>
8  constexpr T parse_into_uint(const char* str) {
9      assert(str != nullptr && "parse_into got nullptr");
10
11     T value = 0;
12     uint16_t radix = 10;
13
14     if (str[0] == '0' && str[1] == 'x') {
15         radix = 16;
16         str += 2;
17     } else if (str[0] == '0' && str[1] == 'b') {
18         radix = 2;
19         str += 2;
20     } else if (str[0] == '0') {
21         radix = 8;
22         ++str;
23     }
24
25     while (*str != '\0') {
26         value *= static_cast<T>(radix);
27         uint16_t symbol_value = radix + 1;
28
29         if (*str >= '0' && *str <= '9') {
30             symbol_value = static_cast<uint16_t>(*str - '0');
31         } else if (*str >= 'a' && *str <= 'f') {
32             symbol_value = static_cast<uint16_t>(*str - 'a') + 10;
33         } else if (*str >= 'A' && *str <= 'F') {
34             symbol_value = static_cast<uint16_t>(*str - 'A') + 10;
35         }
36
37         if (symbol_value >= radix) {
38             assert(false && "parse_into got incorrect string");
39         }
40
41         value += static_cast<T>(symbol_value);
42         ++str;
43     }
44
45     return value;
46 }

```

Теперь мы готовы определить конструкторы класса:

```

1  constexpr uint_t() = default;
2
3  template<typename T>
4  constexpr uint_t(const T& value) : m_digits(split_into_digits<T>(value)) {}
5
6  constexpr uint_t(const char* str) : m_digits(parse_into_uint<uint_t>(str).m_digits) {};
7

```

```
8 constexpr uint_t& operator=(const uint_t& value) = default;
```

Специально не делаем их explicit для неявных конвертаций. С помощью шаблонного конструктора мы можем конструироваться от других экземпляров нашего класса, например:

```
1 uint_t<128> a = ...;
2 uint_t<160> b(a);
```

- Сложение: Самое простое, но тем не менее лучшее решение - это сложение в столбик. Используем стандартизированное переполнение беззнаковых типов в C++ для определения, есть ли остаток от сложения наших 32-битных чисел:

```
1 constexpr uint_t& operator+=(const uint_t& other) {
2     digit_t carry = 0;
3
4     for (size_t i = 0; i < c_digit_number; ++i) {
5         digit_t sum = carry + other[i];
6         m_digits[i] += sum;
7         carry = (m_digits[i] < sum) || (sum < carry);
8     }
9
10    return *this;
11 }
```

Используем ключевое слово constexpr для вычисления значения некоторых констант во время компиляции.

Теперь мы хотим определить простое сложение, т.е. operator+. Его можно было бы сделать через

```
1 constexpr uint_t operator+(const uint_t& other) const {
2     uint_t result = *this;
3     result += other;
4     return result;
5 }
```

Обратите внимание, что мы не пишем:

```
1 constexpr uint_t operator+(const uint_t& other) const {
2     uint_t result = *this;
3     return result += other;
4 }
```

так как тогда мы будем возвращать ссылку uint_t&, что не затриггерит NRVO.

Вместо определения метода класса, мы напишем 4 дружественных функции:

```
1 friend constexpr uint_t operator+(const uint_t& lhs, const uint_t& rhs) {
2     uint_t result = lhs;
3     result += rhs;
4     return result;
5 }
6
7 friend constexpr uint_t operator+(uint_t&& lhs, const uint_t& rhs) {
8     lhs += rhs;
9     return lhs;
10 }
11
12 friend constexpr uint_t operator+(const uint_t& lhs, uint_t&& rhs) {
13     rhs += lhs;
14     return rhs;
15 }
16
17 friend constexpr uint_t operator+(uint_t&& lhs, uint_t&& rhs) {
18     lhs += rhs;
19     return lhs;
20 }
```

Тут есть два оптимизационных момента:

1. Эффективно используется то, что нам передали `r` — value, и не копируем данные. Обычно это возникает при многократном сложении или в других сложных формулах:

```
1 uint_t a,b,c = ...;
2 uint_t result = a + b + c;
```

2. Теперь можно неявно заапкастить другие типы к `uint_t`, чтобы применить данное сложение. Это позволяет писать:

```
1 uint_t a = ...;
2 uint_t b = 3 + a;
```

что было бы невозможно при внутреннем методе. Везде далее будем использовать по возможности внешние `friend` функции для возможности неявного апкаста других типов.

- Вычитание: Оно абсолютно аналогично делается через вычитание в столбик:

```
1 constexpr uint_t& operator--(const uint_t& other) {
2     digit_t remainder = 0;
3
4     for (size_t i = 0; i < c_digit_number; ++i) {
5         digit_t prev = m_digits[i];
6         digit_t sum = other[i] + remainder;
7         m_digits[i] -= sum;
8         remainder = (m_digits[i] > prev) || (sum < remainder);
9     }
10
11     return *this;
12 }
```

Так как мы хотим писать такие конструкции как:

```
1 uint_t a = ...;
2 uint_t b = -a;
```

то нужно определить отрицание. В компьютерах отрицательные целые числа представляются как флиппнутые биты + 1. Рассмотрим на примере:

0000000000000000000101

+

111111111111111111011

=

0000000000000000000000

Определим `inplace` отрицание как приватный метод:

```
1 constexpr void negative() {
2     for (size_t i = 0; i < c_digit_number; ++i) {
3         m_digits[i] = ~(m_digits[i]);
4     }
5
6     ++*this;
7 }
```

Значит отрицанием будет:

```
1 constexpr uint_t operator-() const {
2     uint_t result = *this;
3     result.negative();
4     return result;
5 }
```

Определяем внешние `friend` для вычитания:


```

1  friend constexpr uint_t operator-(const uint_t& lhs, const uint_t& rhs) {
2      uint_t result = lhs;
3      result -= rhs;
4      return result;
5  }
6
7  friend constexpr uint_t operator-(uint_t&& lhs, const uint_t& rhs) {
8      lhs -= rhs;
9      return lhs;
10 }
11
12 friend constexpr uint_t operator-(const uint_t& lhs, uint_t&& rhs) {
13     rhs -= lhs;
14     rhs.negative();
15     return rhs;
16 }
17
18 friend constexpr uint_t operator-(uint_t&& lhs, uint_t&& rhs) {
19     lhs -= rhs;
20     return lhs;
21 }

```

- Умножение: Самым лёгким и относительно быстрым способом будет метод сканирования операнда (или построчное умножение). Заключается в построчном обновлении цифр ответа и накоплении прежних результатов умножения для новой строки. Получается такое динамическое программирование:

```

1  friend constexpr uint_t operator*(const uint_t& lhs, const uint_t& rhs) {
2      uint_t result;
3
4      for (size_t i = 0; i < c_digit_number; ++i) {
5          uint64_t u = 0;
6
7          for (size_t j = 0; i + j < c_digit_number; ++j) {
8              u = static_cast<uint64_t>(result[i + j])
9                  + static_cast<uint64_t>(lhs[i]) * static_cast<uint64_t>(rhs[j])
10                 + (u >> c_digit_size);
11              result[i + j] = static_cast<uint32_t>(u);
12          }
13      }
14
15      return result;
16 }

```

Заметим, что мы отбрасываем цифры, чей номер не помещается в `c_digit_number`. Улучшения этого алгоритма можно найти в [1].

- Деление: Чтобы поделить два целочисленных длинных числа используем алгоритм-D Кнута [2]:

Задача - поделить два длинных числа, представленных цифрами с основанием b , где b в имплементации 2^{32} или 2^{64} .

Рассмотрим сначала $u = (u_n u_{n-1} \dots, u_0)_b$ и $v = (v_{n-1} \dots v_0)_b$, где $u/v < b$. Найдём алгоритм для вычисления $q := \lfloor u/v \rfloor$:

Заметим, что $u/v < b \Leftrightarrow u/b < v \Leftrightarrow \lfloor u/b \rfloor < v$, а это условие того, что

$$(u_n u_{n-1} \dots, u_1)_b < (v_{n-1} \dots v_0)_b$$

Если обозначить $r := u - qv$, то q - это уникальное число, такое что $0 \leq r < v$. Пусть

$$\hat{q} := \min \left(\left\lfloor \frac{u_n b + u_{n-1}}{v_{n-1}} \right\rfloor, b - 1 \right)$$

Т.е. мы получаем гипотетическое значение q , поделив первые две цифры u на первую цифру v , а если результат деления больше или равен b , то берём $b - 1$. Для такого \hat{q} выполняются две теоремы:

Теорема 1. $\hat{q} \geq q$

Теорема 2. Если $v_{n-1} \geq \lfloor b/2 \rfloor$, то $\hat{q} - 2 \leq q \leq \hat{q}$.

Существенно ограничили нашу гипотезу. Умножив u и v на $\lfloor b/(v_{n-1} + 1) \rfloor$, мы не изменим длину числа v и результат деления. После этого умножения станет выполняться вторая из данных теорем.

Алгоритм D: Дано неотрицательное целое число $u = (u_{m+n-1}, \dots, u_1, u_0)_b$ и $v = (v_{n-1}, \dots, v_1, v_0)_b$, где $v_{n-1} \neq 0$ и $n > 1$. Мы хотим посчитать $\lfloor u/v \rfloor = (q_m, q_{m-1}, \dots, q_0)_b$ и остаток $u \bmod v = (r_{n-1}, \dots, r_0)_b$:

1. $d := \lfloor b/(v_{n-1} + 1) \rfloor$. Тогда пусть $(u_{m+n}u_{m+n-1} \dots u_1u_0)_b := (u_{m+n-1} \dots u_1u_0)_b \cdot d$, аналогично, $(v_{n-1}, \dots, v_1, v_0)_b = (v_{n-1}, \dots, v_1, v_0)_b \cdot d$. Заметим, что новая цифра могла появиться только у u
2. Итерироваться будем по j , которая в начале равна m (Делить в следующих шагах будем $(u_{j+n} \dots u_{j+1}u_j)_b$ на $(v_{n-1} \dots v_1v_0)_b$ чтобы получить цифру q_j)
3. $\hat{q} := \left\lfloor \frac{u_{j+n}b + u_{j+n-1}}{v_{n-1}} \right\rfloor$ и пусть \hat{r} будет остатком, т.е. $\hat{r} := u_{j+n}b + u_{j+n-1} \pmod{v_{n-1}}$
4. Если $\hat{q} \geq b$ или $\hat{q}v_{n-2} > b\hat{r} + u_{j+n-2}$, то уменьшаем \hat{q} на 1 и увеличиваем \hat{r} на v_{n-1} . Если $\hat{r} < b$, то повторяем данный шаг
5. Заменим $(u_{j+n} \dots u_{j+1}u_j)_b$ на

$$(u_{j+n} \dots u_{j+1}u_j)_b - \hat{q}(0v_{n-1} \dots v_1v_0)_b$$

6. Назначаем $q_j = \hat{q}$
7. Если число u на 5 шаге получилось отрицательным, то добавляем к нему b^{n+1} и переходим к шагу 8, иначе переходим к шагу 9.
8. (Вероятность данного шага крайне мала, за счёт чего достигается асимптотическая быстрота алгоритма) Уменьшаем q_j на 1 и добавляем $(0v_{n-1} \dots v_1v_0)_b$ к $(u_{j+n} \dots u_{j+1}u_j)_b$ (при сложении появится цифра u_{j+n+1} , её следует проигнорировать)
9. Уменьшаем j на 1. Если $j \geq 0$, то возвращаемся на шаг 3
10. Теперь $q = (q_m \dots q_1q_0)$ - это искомое частное, а искомый остаток можно получить, поделив $(u_{n-1} \dots u_1u_0)$ на d наивным способом
11. Возвращаем (q, r)

Алгоритм сверху применяется только при размере делителя больше 1 цифры и не больше количества цифр в делимом, так как при меньших размерах есть более быстрые оптимизации, значит нам понадобится приватный метод определения количества цифр в числе:

```

1 constexpr size_t actual_size() const {
2     size_t result = c_digit_number;
3
4     while (result > 0 && m_digits[result - 1] == 0) {
5         --result;
6     }
7
8     return result;
9 }

```

Для удобной работы с `m_digits` определим приватные методы для `operator[]`, которые будут проталкивать его внутрь:

```

1 constexpr const digit_t& operator[](size_t pos) const {
2     return m_digits[pos];
3 }
4
5 constexpr digit_t& operator[](size_t pos) {
6     return m_digits[pos];
7 }

```

Наконец, определим приватный метод `divide`, который будет вычислять, в каком случае мы находимся.

```

1  static constexpr uint_t divide(const uint_t& lhs, const uint_t& rhs, uint_t* remainder = nullptr) {
2      size_t dividend_size = lhs.actual_size();
3      size_t divisor_size = rhs.actual_size();
4
5      // CASE 0:
6      if (dividend_size < divisor_size) {
7          if (remainder != nullptr) {
8              *remainder = lhs;
9          }
10
11         return uint_t(0);
12     }
13
14     // CASE 1:
15     if (divisor_size == 1) {
16         return divide(lhs, rhs[0], remainder);
17     }
18
19     // CASE 2:
20     return d_divide(lhs, rhs, remainder);
21 }
22
23 static constexpr uint_t divide(const uint_t& lhs, const digit_t& rhs, uint_t* remainder = nullptr) {
24     uint_t result;
25     double_digit_t part = 0;
26
27     for (size_t i = c_digit_number; i > 0; --i) {
28         part = (part << (c_digit_size)) + static_cast<double_digit_t>(lhs[i - 1]);
29
30         if (part < rhs) {
31             continue;
32         }
33
34         result[i - 1] = static_cast<digit_t>(part / rhs);
35         part %= rhs;
36     }
37
38     if (remainder != nullptr) {
39         *remainder = uint_t(static_cast<digit_t>(part));
40     }
41
42     return result;
43 }
44
45 static constexpr uint_t d_divide(const uint_t& lhs, const uint_t& rhs, uint_t* remainder = nullptr) {
46     size_t dividend_size = lhs.actual_size();
47     size_t divisor_size = rhs.actual_size();
48
49     uint_t<c_bits + c_digit_size> dividend(lhs);
50     uint_t divisor(rhs);
51     uint_t quotient;
52
53     size_t shift_size = 0;
54     digit_t divisor_head = divisor[divisor_size - 1];
55     static constexpr double_digit_t c_HalfBlock = static_cast<double_digit_t>(1)
56                                                     << (c_digit_size - 1);
57
58     while (divisor_head < c_HalfBlock) {
59         ++shift_size;
60         divisor_head <= 1;
61     }
62
63     dividend <= shift_size;
64     divisor <= shift_size;
65
66     double_digit_t divisor_ = divisor[divisor_size - 1];
67     static constexpr double_digit_t c_Block = static_cast<double_digit_t>(1) << c_digit_size;
68
69     for (size_t i = dividend_size - divisor_size + 1; i > 0; --i) {
70         double_digit_t part =

```

```

71         (static_cast<double_digit_t>(dividend[i + divisor_size - 1]) << c_digit_size)
72         + static_cast<double_digit_t>(dividend[i + divisor_size - 2]));
73     double_digit_t quotient_temp = part / divisor_;
74     part %= divisor_;
75
76     if (quotient_temp == c_Block) {
77         --quotient_temp;
78         part += divisor_;
79     }
80
81     while (part < c_Block
82            && (quotient_temp * divisor[divisor_size - 2]
83               > (part << c_digit_size) + dividend[i + divisor_size - 3])) {
84         --quotient_temp;
85         part += divisor_;
86     }
87
88     int64_t carry = 0;
89     int64_t widedigit = 0;
90
91     for (size_t j = 0; j < divisor_size; ++j) {
92         double_digit_t product =
93             static_cast<digit_t>(quotient_temp) * static_cast<double_digit_t>(divisor[j]);
94         widedigit = (static_cast<int64_t>(dividend[i + j - 1]) + carry) - (product & UINT32_MAX);
95         dividend[i + j - 1] = static_cast<digit_t>(widedigit);
96         carry = (widedigit >> c_digit_size) - static_cast<double_digit_t>(product >> c_digit_size);
97     }
98
99     widedigit = static_cast<int64_t>(dividend[i + divisor_size - 1]) + carry;
100    dividend[i + divisor_size - 1] = static_cast<digit_t>(widedigit);
101
102    quotient[i - 1] = static_cast<digit_t>(quotient_temp);
103
104    if (widedigit < 0) {
105        --quotient[i - 1];
106        widedigit = 0;
107
108        for (size_t j = 0; j < divisor_size; ++j) {
109            widedigit += static_cast<double_digit_t>(dividend[i + j - 1]) + divisor[j];
110            dividend[i + j - 1] = static_cast<digit_t>(widedigit);
111            widedigit >>= 32;
112        }
113    }
114 }
115
116 if (remainder != nullptr) {
117     *remainder = uint_t(0);
118
119     for (size_t i = 0; i < divisor_size - 1; ++i) {
120         (*remainder)[i] =
121             (dividend[i] >> shift_size)
122             | (static_cast<double_digit_t>(dividend[i + 1]) << (c_digit_size - shift_size));
123     }
124
125     (*remainder)[divisor_size - 1] = dividend[divisor_size - 1] >> shift_size;
126 }
127
128 return quotient;
129 }

```

Теперь можем определить операторы деления и остатка:

```

1 friend constexpr uint_t operator/(const uint_t& lhs, const uint_t& rhs) {
2     uint_t result = divide(lhs, rhs);
3     uint_t less = result * rhs;
4     uint_t greater = (result + 1) * rhs;
5     if (less > lhs || greater <= lhs) {
6         result = 0;
7     }
8     return result;
9 }

```

```

10
11 friend constexpr uint_t operator%(const uint_t& lhs, const uint_t& rhs) {
12     uint_t remainder;
13     divide(lhs, rhs, &remainder);
14     return remainder;
15 }
16
17 constexpr uint_t& operator*=(const uint_t& other) {
18     return *this = *this * other;
19 }
20
21 constexpr uint_t& operator/=(const uint_t& other) {
22     return *this = *this / other;
23 }

```

Так как нам не нужны r-value при делении, то не пишем оптимизации на них.

- Битовые сдвиги: Нам поступает запрос на сдвиг на `size_t shift` бит влево или вправо. Для высокой производительности выполнение операции нужно разбить на два этапа: сдвиг цифр внутри числа, сдвиг битов внутри цифр:

```

1  constexpr uint_t& operator>>=(size_t shift_size) {
2      size_t digit_shift = shift_size >> 5;
3
4      if (digit_shift > 0) {
5          for (size_t i = 0; i < c_digit_number; ++i) {
6              if (i + digit_shift < c_digit_number) {
7                  m_digits[i] = m_digits[i + digit_shift];
8              } else {
9                  m_digits[i] = 0;
10             }
11         }
12     }
13
14     shift_size %= c_digit_size;
15
16     if (shift_size == 0) {
17         return *this;
18     }
19
20     for (size_t i = 0; i + digit_shift < c_digit_number; ++i) {
21         m_digits[i] >>= shift_size;
22
23         if (i + 1 < c_digit_number) {
24             m_digits[i] |= m_digits[i + 1] << (c_digit_size - shift_size);
25         }
26     }
27
28     return *this;
29 }
30
31 constexpr uint_t& operator<<=(size_t shift_size) {
32     size_t digit_shift = shift_size >> 5;
33
34     if (digit_shift > 0) {
35         for (size_t i = c_digit_number; i > 0; --i) {
36             if (i > digit_shift) {
37                 m_digits[i - 1] = m_digits[i - digit_shift - 1];
38             } else {
39                 m_digits[i - 1] = 0;
40             }
41         }
42     }
43
44     shift_size %= c_digit_size;
45
46     if (shift_size == 0) {
47         return *this;
48     }
49
50     for (size_t i = c_digit_number; i > digit_shift; --i) {

```

```

51         m_digits[i - 1] <= shift_size;
52
53         if (i - 1 > 0) {
54             m_digits[i - 1] |= m_digits[i - 2] >> (c_digit_size - shift_size);
55         }
56     }
57
58     return *this;
59 }

```

В обоих методах в концах двигаем недостающие биты из соседней цифры, если она существует. Определяем внешние friend:

```

1  friend constexpr uint_t operator>>(const uint_t& lhs, const size_t& rhs) {
2      uint_t result = lhs;
3      return result >>= rhs;
4  }
5
6  friend constexpr uint_t operator>>(uint_t&& lhs, const size_t& rhs) {
7      return lhs >>= rhs;
8  }
9
10 friend constexpr uint_t operator<<(const uint_t& lhs, const size_t& rhs) {
11     uint_t result = lhs;
12     return result <<= rhs;
13 }
14
15 friend constexpr uint_t operator<<(uint_t&& lhs, const size_t& rhs) {
16     return lhs <<= rhs;
17 }

```

- Сравнение: Так как используются 20 плюсы, то можно определить оператор <=>, но мы не можем использовать = default, так как тогда сравнение будет с 0 индекса, а не с последнего:

```

1  friend constexpr std::strong_ordering operator<=>(const uint_t& lhs, const uint_t& rhs) {
2      for (size_t i = c_digit_number; i > 0; --i) {
3          if (lhs[i - 1] != rhs[i - 1]) {
4              return lhs[i - 1] <=> rhs[i - 1];
5          }
6      }
7
8      return std::strong_ordering::equal;
9  }

```

Так как мы не определили через default, нам придётся написать и оператор равенства, но он очевиден:

```

1  friend constexpr bool operator==(const uint_t& lhs, const uint_t& rhs) {
2      return lhs.m_digits == rhs.m_digits;
3  }

```

- Битовые операции: Наше число является по сути большой последовательностью бит одного числа, поэтому битовые операции выполняются поэлементно:

```

1  constexpr uint_t& operator^=(const uint_t& other) {
2      for (size_t i = 0; i < c_digit_number; ++i) {
3          m_digits[i] ^= other[i];
4      }
5
6      return *this;
7  }
8
9  constexpr uint_t& operator|=(const uint_t& other) {
10     for (size_t i = 0; i < c_digit_number; ++i) {
11         m_digits[i] |= other[i];
12     }
13
14     return *this;
15 }

```

```

16
17 constexpr uint_t& operator&=(const uint_t& other) {
18     for (size_t i = 0; i < c_digit_number; ++i) {
19         m_digits[i] &= other[i];
20     }
21
22     return *this;
23 }

```

Определяем внешние friend:

```

1 friend constexpr uint_t operator^(const uint_t& lhs, const uint_t& rhs) {
2     uint_t result = lhs;
3     return result ^= rhs;
4 }
5
6 friend constexpr uint_t operator^(uint_t&& lhs, const uint_t& rhs) {
7     return lhs ^= rhs;
8 }
9
10 friend constexpr uint_t operator^(const uint_t& lhs, uint_t&& rhs) {
11     return rhs ^= lhs;
12 }
13
14 friend constexpr uint_t operator^(uint_t&& lhs, uint_t&& rhs) {
15     return lhs ^= rhs;
16 }
17
18 friend constexpr uint_t operator|(const uint_t& lhs, const uint_t& rhs) {
19     uint_t result = lhs;
20     result |= rhs;
21     return result;
22 }
23
24 friend constexpr uint_t operator|(uint_t&& lhs, const uint_t& rhs) {
25     lhs |= rhs;
26     return lhs;
27 }
28
29 friend constexpr uint_t operator|(const uint_t& lhs, uint_t&& rhs) {
30     rhs |= lhs;
31     return rhs;
32 }
33
34 friend constexpr uint_t operator|(uint_t&& lhs, uint_t&& rhs) {
35     lhs |= rhs;
36     return lhs;
37 }
38
39 friend constexpr uint_t operator&(const uint_t& lhs, const uint_t& rhs) {
40     uint_t result = lhs;
41     result &= rhs;
42     return result;
43 }
44
45 friend constexpr uint_t operator&(uint_t&& lhs, const uint_t& rhs) {
46     lhs &= rhs;
47     return lhs;
48 }
49
50 friend constexpr uint_t operator&(const uint_t& lhs, uint_t&& rhs) {
51     rhs &= lhs;
52     return rhs;
53 }
54
55 friend constexpr uint_t operator&(uint_t&& lhs, uint_t&& rhs) {
56     lhs &= rhs;
57     return lhs;
58 }

```

- Унарные инкремент и декремент: Определим вспомогательные приватные методы для увеличения/уменьшения числа на 1, скопировав код `y += /- =` соответственно:

```

1 constexpr void increment() {
2     for (size_t i = 0; i < c_digit_number; ++i) {
3         m_digits[i] += 1;
4
5         if (m_digits[i] != 0) {
6             break;
7         }
8     }
9 }
10
11 constexpr void decrement() {
12     for (size_t i = 0; i < c_digit_number; ++i) {
13         digit_t temp = m_digits[i];
14         m_digits[i] -= 1;
15
16         if (temp >= m_digits[i]) {
17             break;
18         }
19     }
20 }

```

То есть мы пытаемся прибавить/вычесть остаток, пока не найдём хотя бы одну цифру, которая не переполнится от этой операции.

Теперь сами унарные операции. В C++ они бывают двух видов: префиксные и постфиксные. Они разделяются типом `int` в аргументе:

```

1 [[nodiscard("Optimize unary operator usage")]]
2 constexpr uint_t
3     operator++(int) {
4         uint_t result = *this;
5         increment();
6         return result;
7 }
8
9 constexpr uint_t& operator++() {
10     increment();
11     return *this;
12 }
13
14 [[nodiscard("Optimize unary operator usage")]]
15 constexpr uint_t
16     operator--(int) {
17         uint_t result = *this;
18         decrement();
19         return result;
20 }
21
22 constexpr uint_t& operator--() {
23     decrement();
24     return *this;
25 }

```

Дали атрибуты `nodiscard` постфиксным операторам, чтобы пользователь эффективно использовал унарные операции.

- Конвертация в стандартные типы: Два варианта:

1. Хотим обрезать наш тип до стандартных целочисленных типов. Тогда заполняем требуемый тип битами из `m_digits`:

```

1 template<typename T>
2 requires is_convertible_to<T, digit_t>
3 constexpr T convert_to() const {
4     size_t shift_size = sizeof(T) * c_bits_in_byte;
5     size_t digits_number = shift_size / c_digit_size;
6

```



```

7     if (digits_number == 0) {
8         return static_cast<T>(m_digits[0]);
9     }
10
11     T result = 0;
12
13     for (size_t i = 0; i < c_digit_number && i < digits_number; ++i) {
14         result |= static_cast<T>(m_digits[i]) << (i * c_digit_size);
15     }
16
17     return result;
18 }

```

2. Хотим получить всё число. Так как ни один стандартный тип такого размера не поддерживает, то будем переводить наше число в строку обычным делением:

```

1  template<typename T>
2  constexpr T convert_to() const;
3
4  template<>
5  constexpr std::string convert_to() const {
6      std::string result;
7      uint_t clone_of_this = *this;
8
9      do {
10         uint_t remainder;
11         clone_of_this = divide(clone_of_this, 10, &remainder);
12         result.push_back(remainder.m_digits[0] + '0');
13     } while (clone_of_this > 0);
14
15     std::reverse(result.begin(), result.end());
16     return result;
17 }

```

Заметим, что мы специально сделали шаблон, а потом его специализировали, чтобы можно было использовать два варианта одинаково:

```

1  uint_t a = ...;
2  size_t b = a.convert_to<size_t>();
3  std::string s = a.convert_to<std::string>();

```

2.3 Главный тип

Независимо от того, выбрали мы писать свою длинку или использовать `boost::multiprecision`, нам нужно задать главный целочисленный тип, на котором мы будем в дальнейшем работать.

Так как обычно для алгоритмов шифрования на эллиптических кривых мы будем использовать эллиптические кривые, предложенные NIST, то нам нужно выбрать количество бит, которое в два раза больше количества бит используемых чисел, чтобы числа не переполнялись при умножении. В данном гайде будем ориентироваться на NIST P-256, поэтому нам понадобится 512 бит в нашей длинке.

Два варианта:

1. Для бустовской длинки:

```

1  #include "boost/multiprecision/cpp_int.hpp"
2  #include "boost/multiprecision/fwd.hpp"
3
4  using uint = boost::multiprecision::uint512_t;

```

2. Для самописной:

```

1  #include "long-arithmetic.h"
2
3  using uint = uint_t<512>;

```

3 Поле

Наши поля - это простые конечные поля, значит можно представить их в виде вычетов по простому модулю. Но чем мы будем оперировать? Элементами поля. То есть нам нужен класс элементов поля. Но они должны как-то знать свой простой модуль, который будет одинаков у большинства элементов этого поля, поэтому надо как-то ввести общий элемент для множества объектов. Главная идея: создать класс поля, который сам будет оркестрировать элементами поля: создавать элементы поля и давать им общий простой модуль.

Итого нужно реализовать два класса:

3.1 Элемент поля

Так как нужно раздавать простой модуль на множество объектов, то одним из лучших решений будет делать это через умный указатель на константный объект. так как мы не хотим, чтобы наш класс создавали извне, кроме класса поля, то сделаем класс поля другом, а конструктор приватным.

3.1.1 Каркас

```
1 class FieldElement {
2     friend class Field;
3
4     FieldElement(const uint& value, std::shared_ptr<const uint> modulus);
5     uint m_value;
6     std::shared_ptr<const uint> m_modulus;
7 };
```

Так как поступившее число может быть не меньше модуля, то определим приватный метод нормализации:

```
1 uint FieldElement::normalize(const uint& value, std::shared_ptr<const uint> modulus) {
2     return value % *modulus;
3 }
```

Итого:

```
1 class FieldElement {
2     friend class Field;
3
4     FieldElement(const uint& value, std::shared_ptr<const uint> modulus) :
5         m_value(normalize(value, modulus)), m_modulus(std::move(modulus)) {};
6     uint m_value;
7     std::shared_ptr<const uint> m_modulus;
8 };
9
```

3.1.2 Методы

- Стандартным образом определим по 4 дружественных оператора на каждую операцию $+$, $-$, $*$, $/$, $<<$ через их $+$ версии. Заметим, что нельзя определить операторы $>>$, $>>=$, так как это нарушает арифметику простого поля, потому что деление в поле - это умножение на обратный к делителю.
- Метод отрицания — это просто модуль - значение:

```
1 FieldElement FieldElement::operator-() const {
2     return FieldElement(*m_modulus - m_value, m_modulus);
3 }
```

- Возведение в степень uint — стандартный double-and-add:

```
1 template<typename T>
2 T fast_pow(const T& value, const uint& power) {
3     if ((power & 1) != 0) {
4         if (power == 1) {
5             return value;
6         }
7     }
```

```

8         return value * fast_pow<T>(value, power - 1);
9     }
10
11     T temp = fast_pow<T>(value, power >> 1);
12     return temp * temp;
13 }
14
15 FieldElement FieldElement::pow(const FieldElement& element, const uint& power) {
16     return fast_pow<FieldElement>(element, power);
17 }

```

- Обратный элемент по простому модулю можно найти по расширенному алгоритму Евклида. Не буду себя утруждать объяснениями такого базового алгоритма:

```

1  template<typename T>
2  static T extended_modular_gcd(const T& a, const T& b, T& x, T& y, const T& modulus) {
3      if (b == 0) {
4          x = 1;
5          y = 0;
6          return a;
7      }
8
9      T x1, y1;
10     T d = extended_modular_gcd<T>(b, a % b, x1, y1, modulus);
11     x = y1;
12     T temp = y1 * (a / b);
13
14     while (x1 < temp) {
15         x1 += modulus;
16     }
17
18     y = x1 - temp;
19     return d;
20 }
21
22 template<typename T>
23 T inverse_modulo(const T& value, const T& modulus) {
24     T result, temp;
25     extended_modular_gcd<T>(value, modulus, result, temp, modulus);
26     return result;
27 }
28
29 void FieldElement::inverse() {
30     m_value = inverse_modulo<uint>(m_value, *m_modulus);
31 }

```

- Заметим, что для сложения и вычитания достаточно не деления, а простого сравнения для поддержания инварианта:

```

1  FieldElement& FieldElement::operator+=(const FieldElement& other) {
2      m_value += other.m_value;
3
4      if (m_value >= *m_modulus) {
5          m_value -= *m_modulus;
6      }
7
8      return *this;
9  }
10
11 FieldElement& FieldElement::operator-=(const FieldElement& other) {
12     if (m_value < other.m_value) {
13         m_value += *m_modulus;
14     }
15
16     m_value -= other.m_value;
17
18     return *this;
19 }

```

- Битовый сдвиг влево производим по одному, чтобы случайно не получить переполнение:

```

1 FieldElement& FieldElement::operator<<=(const uint& shift) {
2     for (size_t i = 0; i < shift; ++i) {
3         m_value <<= 1;
4
5         if (m_value >= *m_modulus) {
6             m_value -= *m_modulus;
7         }
8     }
9
10    return *this;
11 }

```

- Сравнение будет в двух вариантах, так как длинка от буста не поддерживает оператор <=>:

```

1 friend bool operator==(const FieldElement& lhs, const FieldElement& rhs) {
2     return lhs.m_value == rhs.m_value;
3 }
4
5 #ifdef ECG_USE_BOOST
6 friend bool operator<(const FieldElement& lhs, const FieldElement& rhs) {
7     return lhs.m_value < rhs.m_value;
8 }
9
10 friend bool operator>(const FieldElement& lhs, const FieldElement& rhs) {
11     return lhs.m_value > rhs.m_value;
12 }
13
14 friend bool operator<=(const FieldElement& lhs, const FieldElement& rhs) {
15     return lhs.m_value <= rhs.m_value;
16 }
17
18 friend bool operator>=(const FieldElement& lhs, const FieldElement& rhs) {
19     return lhs.m_value >= rhs.m_value;
20 }
21
22 friend bool operator!=(const FieldElement& lhs, const FieldElement& rhs) {
23     return lhs.m_value != rhs.m_value;
24 }
25 #else
26 friend std::strong_ordering operator<=>(const FieldElement& lhs, const FieldElement& rhs) {
27     return lhs.m_value <=> rhs.m_value;
28 }
29 #endif

```

- Делаем публичный метод is_invertible, который проверяет, обратим ли элемент в поле. Так как в простом поле любой ненулевой элемент обратим, то

```

1 bool FieldElement::is_invertible() const {
2     return m_value != 0;
3 }

```

- Прописываем геттеры для модуля и значения и идём дальше.

3.2 Поле

Поле будем создавать от простого модуля в uint или в виде строки:

```

1 class Field {
2 public:
3     Field(const uint& modulus);
4
5 private:
6     std::shared_ptr<const uint> m_modulus;
7 };

```

Заметим, что конвертация от строки к uint произойдёт автоматически, если у uint есть не explicit конструктор от строки, и можно будет писать так:

```
1 Field F("0xFFFFFFFF123");
```

Но у бустовской длинки нет конструктора от строки, поэтому заведём отдельный дефайн ECG_USE_BOOST, если мы использовали длинку от буста, и определим новый конструктор:

```
1 class Field {
2 public:
3 #ifdef ECG_USE_BOOST
4     Field(const char* str);
5 #endif
6     Field(const uint& modulus);
7
8 private:
9     std::shared_ptr<const uint> m_modulus;
10 };
```

Теперь мы бы хотели создавать элементы поля. Из-за буста опять в двух вариантах:

```
1 class Field {
2 public:
3 #ifdef ECG_USE_BOOST
4     Field(const char* str);
5     FieldElement element(const char* str) const;
6 #endif
7     Field(const uint& modulus);
8     FieldElement element(const uint& value) const;
9     const uint& modulus() const;
10
11 private:
12     std::shared_ptr<const uint> m_modulus;
13 };
```

где element реализуем как:

```
1 #ifdef ECG_USE_BOOST
2 FieldElement Field::element(const char* str) const {
3     return FieldElement(uint(str), m_modulus);
4 }
5 #endif
6 FieldElement Field::element(const uint& value) const {
7     return FieldElement(value, m_modulus);
8 }
```

Даём оператор сравнения на равенство для поля и геттер поля:

```
1 const uint& Field::modulus() const {
2     return *m_modulus;
3 }
4
5 bool Field::operator==(const Field& other) const {
6     return *m_modulus == *other.m_modulus;
7 }
```

Идём дальше.

4 Эллиптическая кривая

Список литературы

- [1] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. Technical report, Institute for Applied Information Processing and Communications, Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria, 2011.
- [2] Donald E. Knuth. *Art of computer programming*, volume 2. Addison-Wesley Professional, 1985.