МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра математического моделирования и анализа данных

БЖЕЗИНСКИЙ ТАРАС АЛЕКСАНДРОВИЧ

ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ СТБ 34.101.31-2011 И СТБ П 34.101.45–2011.

Курсовая работа
студента 4 курса 9 группы

"Допустить к защите"
с предварительной оценкой ____

Руководитель работы
____
«___»_____________ 2012 г

Руководитель
Агиевич Сергей Валерьевич
заведующий НИЛ ПБИТ,
кандидат физ.-мат. наук

Минск 2012

Содержание

1 Проблема реализации в современной криптографии

В настоящее время существует достаточное количество криптографических комплексов и примитивов. Однако, реализацию каждого из алгоритмов нужно рассматривать не только с точки зрения криптографической стойкости и ассимптотического времени работы, но и с некоторых других. Схематично мы можем описать это следующим образом:

- $\mathcal{THEORY}$ - под этим мы будем понимать криптографическую стойкость, суть и актуальность алгоритма.
- $\mathcal{HARDWARE}$ - особенности архитектуры и емкости ресурсов ЭВМ, на которых будет применяться алгоритм.
- $\mathcal{SOFTWARE}$ - особенности програмной реализации, зависящие от предыдущих 2 пунктов.

Каждый из криптографических методов могут быть реализованы либо способом $\mathcal{HARDWARE}$, либо $\mathcal{SOFTWARE}$.

Возможность программной реализации обуславливается тем, что все методы криптографического преобразования формальны и могут быть представлены в виде конечной алгоритмической процедуры.

При аппаратной реализации все процедуры шифрования и дешифрования выполняются специальными электронными схемами.

В последнее время стали появляться комбинированные средства шифрования, так называемые программно-аппаратные средства. Такой метод объединяет в себе достоинства программных и аппаратных методов. В нашей работе будут использованы оба подхода, и среди главных особенностей выделим следующие:

1. Представление длинных чисел и многочленов как записи подряд некоторых двоичных слов, длина которых равна машинному слову.
2. Эффективный алгоритм умножение многочленов - реализация арифметики в $\mathbb{GF}_{2^{128}}$.
3. Эффективный алгоритм работы с длинными числами - реализация арифметики `BigInteger`.
4. Группа эффективных алгоритмов работы с точками эллиптической кривой - реализация арифметики `Elliptic curve`.

Выбор последних 3 особенностей обусловлен рядом причин, среди которых следует выделить работу над проектом гражданской карточки в РБ, где нужны данные разработки, а также их широкое применение в двух белорусских стандартах [10] и [9]. Схематично взаимосвязь этих особенностей в стандартах можно изобразить следующим образом:

- $\mathbb{GF}_{2^{128}}$ + `BigInteger` = [10].
- `Elliptic curve` + [10] = [9].

В данной работе мы рассмотрим общее понятия, варианты выбора алгоритмов, а также объясним выбор алгоритма для нашей реализации для пунктов 2 и 3.

Отметим, что первый из них будет применяться в каждой реализации.

2 Эффективная реализация арифметики в $\mathbb{GF}_{2^{128}}$

2.1 Понятие о $\mathbb{GF}_{q^m}$

Многочленом $f(x)$ над конечным полем $\mathbb{GF}_{q^m}$ называется формальная сумма вида

$$f(x) = f_0 + f_1 x + \ldots + f_m x^m, \quad f_i \in \mathbb{GF}_q, \quad f_m \neq 0.$$

Здесь m — целое неотрицательное число, называемое степенью многочлена $f(x)$, а x^k — элементы алгебры над $\mathbb{GF}_{q^m}$ умножение которых задаётся правилами:

1. $x^k \cdot x^m = x^{k+m}$.

2. $x^0 \equiv 1$.

Также ,как и в случае с обычными многочленами верно положение о единственности деления с остатком.

С $\mathbb{F}_{q^m}$ будем ассоциировать неприводимый многочлен $F(x)$ степени m. Тогда умножение двух многочленов $a(x)$ и $b(x)$ определим следующим образом:

$$[a(x) \cdot b(x)]_{\mathbb{GF}_{q^m}} = (a(x) \cdot b(x)) \;\; mod \;\; F(x),$$

где mod обозначает остаток от деления.

2.2 Постановка задачи

Нас будет интересовать кольцо многочленов в $\mathbb{GF}_{2^{128}}$, где в качестве неприводимого модульного многочлены выступает

$$f(x) = x^{128} + x^7 + x^2 + x^1 + x^0 \tag{1}$$

Из базовых операций в кольце нас будут интересовать следующие:

- Сложение.

- Умножение.

В обоих пунктах будем использовать, что степень многочлена не превосходит 128 ,а также эффективное хранение многочлена в памяти. 128 бит, необходимых на хранение, можно разбить на 4 группы по 32 бита, в свою очередь 32 бита - это размер машинного слова на архитектуре X86. Таким образом, для хранение многочлена необходимо 4 машинных слова, при этом следующие операции могут выполняться за $O(1)$:

1. **isSet(i)** - функция принимает на вход степень, и выдает `ДА`, если x^i входит в многочлен с коэффициентом 1 и `НЕТ` в противном случае.

2. **set(i, value)** - функция принимает на вход степень и коэффициент. Результат работы: установка $coeff_i$ (коэффициента перед x^i) равным $value$.

3. **toggle(i)** - функция принимает на вход степень и устанавливает коэффициент перед x^i равный $coeff_i \leftarrow 1 \oplus coeff_i$.

4. **mul(i)** - функция умножения данного многочлена на x^i. Эквивалента логическому сдвигу влево.

5. **degree(g(x))** - функция возвращает степень многочлена.

Для взятия многчлена $g(x)$ по модулю 1 будем использовать алгоритм **MOD(g(x), f(x))**, который работает за $O(degree)$ следущим образом:

1. $m \leftarrow degree(g(x))$..
2. ПОКА $m \geq 128$ ВЫПОЛНЯТЬ:
3. ЕСЛИ **isSet(m)** ПЕРЕЙТИ К 4, ИНАЧЕ ПЕРЕЙТИ К 1.
4. $toggle(m)$.
5. $toggle(m - 128)$.
6. $toggle(m - 127)$.
7. $toggle(m - 126)$.
8. $toggle(m - 121)$.

2.3 Сложение

Рассмотрим вопрос о сложение двух многочленов $a(x) = a_{127}x^{127} + \ldots + a_1x + a_0$ и $b(x) = b_{127}x^{127} + \ldots + b_1x + b_0$.

Обозначим представление $a(x)$ в памяти процессора как $A_1A_2A_3A_4$, где $A_i \in 0, 1^{32}$ - машинное слово. Аналогично, $b(x)$ представлен в памяти как $B_1B_2B_3B_4$. Заметим, что сложение можно производить независимо для каждой степени - операция сложения сводится к сложению соотвествующих коэффициентов, а так как поле конечно, то $a_i + b_i \quad \in \mathbb{F}_2 \forall i$. Заметим, что сложение по модулю 2 является по сути логическим исключащим `ИЛИ`. Таким образом, простейшей версией алгоритма будет следующая схема:

1. Для $i = 0, \ldots, 127$ повторить:
2. $c_i = a_i \oplus b_i$.
3. $set(i, c_i)$

Однако, команда `XOR` является встроенной процессорной командой, в связи с чем сложение можно производить по блокам. Тогда, сложение 2 многочленов сведется к 4 процессорным атомарным командам, и таким образом, выполняться будет за $4O(asmOperation)$. Итоговый алгоритм имеет вид:

1. Для $i = 1, \ldots, 4$ повторить:
2. $C_i = A_i \oplus B_i$.

2.4 Умножение

Умножение многочленов будем проводить в 2 этапа:

- **MUL** - собственно умножение.
- **MOD** - взятие результата по модулю 1.

Рассмотрим первый этап подробнее. В данной работе будут рассмотрены 4 подхода и найдено оптимальное их сочетание:

1. Умножение в столбик.
2. Умножение Карацубы.
3. Умножение Тоома-Кука.
4. Умножение с табличными данными.

2.4.1 Умножение в столбик

Алгоритм умножения в столбик ничем не отличается от умножения чисел, однако будем использовать операции сложения и умножения, определенные в $\mathbb{F}_{2^128}$. На вход алгоритму поступают 2 многочлена $a(x), degree(a) = m$ и $b(x), degree(b) = n$. На выходе алгоритм выдает многочлен $c(x) = a(x) \cdot b(x)$.

Шаги алгоритма:

- $c \leftarrow 0$.
- `Для` $i = 0, \ldots n$ `повторять`:
- `ЕСЛИ` $isSet(b, i)$, `ТО` $c \leftarrow c + a(x) \cdot x^i$.

Несложно заметить, что сложность алгоритма $O(n \cdot m)$.

2.4.2 Умножение Карацубы

Данный метод был придуман А.А. Карацубой в 1960 году и имеет трудоемкость $M(n) = O(n^{\log_2 3})$. На вход алгоритму поступают 2 многочлена $A(x), degree(A) = m$ и $B(x), degree(B) = m_1$. На выходе алгоритм выдает многочлен $c(x) = a(x) \cdot b(x)$. В основе алгоритма лежит равенство

$$(a + bx)(c + dx) = ac + ((a + b)(c + d) - ac - bd)x + bdx^2.$$

Пусть,$n = 2^m$ - максимальная степень многочлена $+1, n = 2k$. Представляя A и B в виде: $A = A_1 + x^k A_2, \quad B = B_1 + x^k B_2$, где $A_1,\ A_2,\ B_1,\ B_2$ — многочлены степени меньшей k, находим:

$$AB = A_1B_1 + x^k((A_1 + A_2)(B_1 + B_2) - (A_1B_1 + A_2B_2)) + x^n A_2B_2. \tag{2}$$

Теорема 1. *Пусть $\varphi(n)$ - количество операций, достаточное для умножения двух многочленов степени менее n по формуле 2. Тогда справедливо неравенство:$\varphi(n) < (2c + 1)n^{\log_2 3}, \quad \log_2 3 = 1{,}5849\ldots$*

Важным свойством метода Карацубы является то, что его можно легко представить в рекурсивной форме. Однако на практике, при маленьких n ввиду выполняемых нетривиальных операций метод проигрывает обычному умножению в столбик. Поэтому обычно метод реализуется следующим образом:

- `ЕСЛИ` $n \geq threshold$, `ТО` $KARATSUBA(\frac{n}{2})$.
- `ИНАЧЕ УМНОЖИТЬ В СТОЛБИК.`

При этом значение *threshold* зависит от архитектуры ЭВМ и часто выбирается экспериментальным путем.

2.4.3 Умножение Тоома-Кука

Другим методом умножения двух многочленов является метод Тоома-Кука, который является обобщением метода Карацубы (который часто называется `Cook-2`). Расммотрим работу алгоритма `Cook-k`, для понимания общей сути алгоритма. Будем считать, что на вход алгоритму поступают 2 многочлена $a(x), b(x)$ степени $n, k \div n/$ Алгоритм состоит из нескольких фаз:

- Фаза Разделение. Каждый из многочленов делится на k частей длины $\frac{n}{k}$:

$$A(X) = A_0(X) + A_1(X) \cdot X + \ldots + A_{k-1}(X) \cdot X^{k-1},$$

$$B(X) = B_0(X) + B_1(X) \cdot X + \ldots + B_{k-1}(X) \cdot X^{k-1},$$

$$X \in \mathbb{GF}_{2^k}.$$

- Фаза Выбор. Произвольным образом выбираются $2 \cdot k - 1$ точек $\{T_i\}, i = 0, \ldots, k-1, T_i \in \mathbb{GF}_{2^k}$. К примеру, при $k = 2$ такими точками могут служить $0, 1, \infty$.
- Фаза Подсчет. Подсчитываются два вектора $eval_A(i), eval_B(i)$, где $eval_A(i) = A(T_i), eval_B(i) = B(T_i)$.
- Фаза Свертка. Производится вычисление свертки $eval_{AB} = eval_A * eval_B$ по формуле

$$eval_{AB}(i) = eval_A(i) \cdot eval_B(i)$$

.

- Фаза Интерполяция. На основе вычисленных значений $eval_{AB}(i)$ строится многочлен AB, такой что $AB(T_i) = eval_{AB}(i)$.

- Фаза Контакенация. Операцией, обратной к действиям из фазы Разделение формируется многочлен $ab(x) = a(x) \cdot b(x)$.

Американский математик Дональд Кнут в своей работе [1] доказал следующую теорему о времени работы вышеописанного алгоритма.

Теорема 2. *Время работы алгоритма* `Cook-k` *при умножении двух многочленов длины n равняется $\Theta(c(k) \cdot n^e)$, где $e = \frac{\log(2k-1)}{\log(k)}$, n^e - время, затраченное на умножения, $c(k) = O(k)$ - время, затраченное на промежуточные сложения и умножения базовых типов.*

На практике алгоритм часто реализуется способом, похожим на указанный в 2.4.2.

2.4.4 Табличное умножение

Суть данного метода состоит в следующем: предпросчитаем таблицу $H[x][y]$, где $H[x][y] = x \cdot y, \forall x, y \in \mathbb{GF}_{2^t}$. Тогда умножение многочленов $a(x), b(x)$ можно выполнить следующим образом:

1. Представить $a(x), b(x)$ в виде контакенации бинарных слов A, B длины, кратной t (при необходимости дополнить лидирующими нулями).
2. Представить результат $c(x)$ в аналогичном виде.
3. `для` $i = 0, \dots, length(A) - 1$
4. `для` $j = 0, \dots, length(B) - 1$
5. $C(i+j) \leftarrow C(i+j) \oplus H[A(i)][B(j)]$
6. Возвратить c

Данный метод имеет сложность $O(\frac{degree(a) \cdot degree(b)}{t})$. На практике часто таблица предпросчитывается для байтов, и занимает 2^{17} байт памяти (около 66 килобайт).

2.4.5 Умножение в нашей схеме

Схема/Критерий	2.4.1	2.4.2	2.4.3	2.4.4
Ассимптотическое время работы	$O(n \cdot m)$	$O(n^{\log_2 3})$	$\Theta(c(k) \cdot n^e)$	$O(\frac{degree(a) \cdot degree(b)}{t})$
Затраты памяти	$O(1)$	$O(n)$	$O(n)$	$O(2^{17})$
Максимальная глубина рекурсии	0	$\log_2(\frac{n}{threshold})$	$\log_2(\frac{n}{threshold})$	0

Таблица 1: Сравнительная характеристика 4 схем умножения

Из таблицы видно, что наилучшим отношением памяти и времени работы выделяется алгоритм Тоома-Кука. Однако, его ассимптотика проявляет себя лишь при степенях, больших 1000. Так, по результатам стресс-тестов,

- На длине 512 он сравним с методом Карацубы.
- На длине 256 он проигрывает методу Карацубы.
- На длине 128 он проигрывает методу Карацубы и сравним с методом умножения в столбик.

Таким образом, для необходимой нам длины 128 оптимальным является метод Карацубы. Попытаемся найти оптимизации для него.

Рассмотрим 3-ий этап метода Карацубы, когда длина равна 32 бита. Обратимся к таблице $H[i][j]$, определенной для произведений многочленов степеней 7 и менее. Умножив 2 машинных слова табличным методом на данном этапе, мы уже получаем ассимптотическое улучшение по сравнению со стандартным методом. При этом таблица $H[i][j]$ будет помещаться в кэш любой современной $x86$ и $x64$ архитектурной реализации, что позволит ассимптотическому улучшению проявиться и в реальности. Таким образом, схема нашего метода будет иметь следующий вид:

1. Вызывать 2.4.2 для длин 128, 64.

2. Вызвать 2.4.4 для длины 32.

Данный метод достаточно красиво и легко реализуется, при этом достигается достаточно оптимальное время работы (как реальное, так и ассимптотическое). Пример конкретной реализации можно увидеть в приложении к работе.

Нами были рассмотрены одни из самых широко распространенных и ассимтотически оптимальных алгоритмов и был придуман на их основе собственный алгоритм умножения многочленов. Был на практике подтвержден давний принцип о том, что ассимптотическое преимущество начинает проявляться начиная лишь с достаточно больших размерностях.

3 Эффективная реализация арифметики эллиптических кривых

3.1 Общее понятие о кривых

Определение 1. *Эллиптическая кривая - это кривая над $\mathbb{F}_p$ в двумерном евклидовом пространстве (p - простое) вида*

$$\begin{cases} y^2 = x^3 + ax + b, \\ a, b, x, y \in \mathbb{F}_p \end{cases} \tag{3}$$

Обозначим $E^{\star}_{a,b}(\mathbb{F}_p)$ - множество таких точек (x, y), которые удовлетворяют уравнению 3 . Данное множество называют аффиными точками кривой. Добавим к аффинным точкам специальную бесконечно удаленную точку O и образуем множество $E_{a,b}(\mathbb{F}_p)$. Пусть $4a^3 + 27b^2 \neq 0 \ (mod\ p)$, тогда множество $E_{a,b}(\mathbb{F}_p)$ будет являться аддитивной группой (все вычисления ведутся в $\mathbb{F}_p$) при следующих правилах сложения:

1. $O + P = P + O = P \quad \forall P \in E_{a,b}(\mathbb{F}_p)$.

2. Если $P = (x, y) \in E_{a,b}(\mathbb{F}_p)$, то $-P = (x, p - y)$ и $P + (-P) = O$.

3. Если $P_1 = (x_1, y_1) \in E_{a,b}(\mathbb{F}_p)$ и $P_2 = (x_2, y_2) \in E_{a,b}(\mathbb{F}_p)$ и $P2 \neq (-P1)$, то$P_1 + P_2 = P_3 = (x_3, y_3)$, где

$$x_3 = \lambda^2 - x_1 - x_2,\ y_3 = \lambda(x_1 - x_3) - y_1,\ \lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1}, & P_1 = P_2 \end{cases}.$$

Сумма k экземпляров точки P называется k-кратной ей точкой и обозначается через kP. Также считается, что $0P = O$.

В настоящем разделе будут рассмотрены вопросы эффективного вычисления кратной точки и выражения вида $\alpha A + \beta B$, где $\alpha,\ \beta \in \{1, \ldots, q-1\}$, а $A, B \in E_{a,b}(\mathbb{F}_p),\ |E_{a,b}(\mathbb{F}_p)| = q$.

Перед непосредсвенным рассмотрением этих вопросов заметим следующее: при сложение двух точек всегда производится операция деления, то есть, умножения на обратный элемент. Данная операция является достаточно трудоемкой, поэтому в следующем параграфе мы рассмотрим способы избежать этого.

3.2 Проективные координаты

Описываемую в предыдущем разделе систему координат принято называть аффинной. На практике кроме нее существуют другие системы, которые могут быть более полезны в каких-либо ситуациях. Рассмотрим следующее преобразование координат:

$$(x, y) \longmapsto (X : Y : Z) = \langle (X, Y, Z) : X, Y, Z \in \mathbb{F}_p, Z \neq 0, \frac{X}{Z^n} = x, \frac{Y}{Z^m} = y \rangle = \langle (\lambda^n x, \lambda^m y, \lambda), \lambda \in \mathbb{F}_p^{\star} \rangle \tag{4}$$

Полученную плоскость назовем проективной, а тройку координат $(X : Y : Z)$ - проективными, при этом n, m - параметры. При $n = 2, m = 3$ получаем Якобиановы координаты. Точка $O = (0, 0)$ в аффинных координатах переходит в точку $(1 : 1 : 0)$ в проективных. Выведем формулы удвоения (формулы сложения получаются точно таким же образом) точки, используя проективные координаты.

- $(x, y) \longmapsto (x : y : 1)$.

- $(x : y : 1) \longmapsto (X : Y : Z)$.

- $Z \to Z^2 \to Z^3 \to Z^{3^{-1}}, \Rightarrow \frac{Y}{Z^3}, \frac{X}{Z^2} = \frac{X}{Z^3} \cdot Z$.

Рассмотрим подробнее шаг 2.

$$\left[2(x_1,y_1)=(x_3,y_3),\lambda=\frac{3x_1^2+a}{2y_1},x_3=\lambda^2-2x_1,y_3=\lambda(x_1-x_3)-y_1\right]$$

$$2(\frac{X_1}{Z_1^2}:\frac{Y_1}{Z_1^3}:1)=(X_3:Y_3:1),\lambda=\frac{3\frac{X_1}{Z_1^2}^2+a}{2\frac{Y_1}{Z_1^3}}=\frac{3X_1^2+aZ_1^4}{2Y_1Z_1}$$

$$x_3=\frac{3X_1^2+aZ_1^4}{2Y_1Z_1}^2-2\frac{X_1}{Z_1^2},y_3=(\frac{3X_1^2+aZ_1^4}{2Y_1Z_1})(\frac{X_1}{Z_1^2}-x_3)-\frac{Y_1}{Z_1^3},$$

$$Z_3=2Y1Z1,X_3=x_3Z_3^2,Y_3=y_3Z_3^3,$$

$$x_3=(3X_1^2+aZ_1^4)^2-8X_1Y_1^2,y_3=(3X_1^2+aZ_1^4)(4X_1Y_1^2-X_3)-8Y_1^4$$

Таким образом, удвоение в проективных координатах можно представить в виде следующей схемы:

Действие	Количество операций
$A\leftarrow Y_1^2$	$1S$
$B\leftarrow 4X_1A$	$1M$
$C\leftarrow 8A^2$	$1S$
$D\leftarrow 3X_1^2+aZ_1^4$	$3S+1M$
$X_3\leftarrow D^2-2B$	$1S$
$Y_3\leftarrow D(B-X_3)-C$	$1M$
$Z_3\leftarrow 2Y_1Z_1$	$1M$

Таблица 2: Схема удвоения в проективных координатах, S - операция возведения в квадрат, M - операция умножения

Получаем, что общая сложность удвоения точки в проективных координатах составляет $6M+4S$ операций.

Рассмотрим следующее выражение $3X_1^2+aZ_1^4$. Если $a=-3 \mod P$, тогда $3X_1^2+aZ_1^4=3(X_1-Z_1^2)(X_1+Z_1^2)$, и общая сложность подсчета становится $1S+1M$.

Вид координат	Сложение	Удвоение
Якобиановы	$12M+4S$	$4M+6S$
Якобиановы ($a=-3 \mod P$	$12M+4S$	$4M+4S$
Стандартные проективные ($n=m=1$)	$12M+3S$	$7M+3S$

Таблица 3: Сравнительная характеристика сложности операций в разных проективных системах координат

Как видно из 3, наиболее быстрой будет являться для нас вторая система, которая и будет использована в нашей схеме. Очевидно, что сложности операций в ней существенно меньше сложности операций в аффинной системе координат.

Далее будем считать, что все примитивные операции над точками (сложение и удвоение) производятся в проективной системе координат.

3.3 Вычисление кратной точки

В данном разделе будут рассмотренны 2 алгоритма вычисления кратной точки и произведено их сравнение на основе стресс-тестов.

3.3.1 Аддитивные цепочки

Определение 2. *Последовательность* $(a_0 = 1, a_1, \dots, a_n = d)$ *называют аддитивной цепочкой для* d*, если* $\forall i = 1, 2, \dots, n, \exists j, k < i : a_i = a_j + a_k$.

`Алгоритм нахождения кратной точки с помощью аддитивной цепочки`
Входные данные: $P, d, (a_0, \dots, a_n)$ - аддитивная цепочка для d.
Выходные данные: кратная точка dP.
Шаги:

1. $P_0 \leftarrow P$.
2. `ДЛЯ` $i = 1, \dots, n$:
   (a) `НАЙТИ` $j, k < i : a_i = a_j + a_k$.
   (b) $P_i = P_j + P_k$.
3. `ВОЗВРАТИТЬ` P_n.

Оценим трудоемкость и память алгоритма: n сложений точек по времени и n точек, хранимых в памяти.

Для более точной оценки воспользуемся следующей теоремой (Эрдеш, 1960):

Теорема 3.

$$n = l(d) = \log_2(d) + (1 + o(1))\frac{\log_2(d)}{\log_2(\log_2(d))}$$

Существует эффективный алгоритм Брауэра, позволяющий строить такие цепочки оконным методом.

`Алгоритм Брауэра`
Входные данные: d.
Выходные данные: $B(d)$ - аддитивная цепочка Брауэра.
Переменные: $b = 2^k$.
Шаги:

1. Представить d в виде $(d_{s-1} \dots d_1 d_0)_b$, $d_i \in 0, \dots, b-1$, $d = \sum\limits_0^s d_i b^i$.
2. `ЕСЛИ` $d < b$, `ТО` $B(d) = (1, 2, \dots, b-1)$.
   `ИНАЧЕ` $B(d) = (B(d^{'}, 2d^{'}, 4d^{'}, \dots, bd^{'}, d$, где $d^{'} = (d_{s-1} \dots d_1)_b$.

3.3.2 Бинарные методы

В данном разделе будет показано применение метода бинарного возведения в степень для нахождения кратной точки.

Входные данные: d, P, $d = (d_{l-1} \dots d_1 d_0)_2$.
Выходные данные: dP.
Шаги:

1. $U \leftarrow 0, V \leftarrow P$.
2. `ДЛЯ` $i = 0, \dots, l-1$:
   (a) $V \leftarrow 2V$
   (b) `ЕСЛИ` $d_i \neq 0$, `ТО` $U \leftarrow U + V$.
3. Возвратить U.

Сложность алгоритма - $O(W(d))$, где $W(d)$ - количество единичных бит в d.

3.3.3 Метод нашей схемы

Как и в случае с умножением многочленов, в нашей схеме будут использоваться оба вышеописанных метода. Стресс-тесты показали, что для $d \leq MAGICVALUE$ бинарный алгоритм работает быстрее, где $MAGICVALUE = \sqrt{q}, q$ - параметр эллиптической кривой (256 битное число).

Таким образом наш алгоритм будет объединением вышеперечисленных и работать следующим образом:

- `ЕСЛИ` $d \leq \sqrt{q}$, `ТО` используется бинарный алгоритм.
- `ИНАЧЕ` используется алгоритм с применением аддитивных цепочек (строятся с помощью алгоритма Брауэра).

3.4 Вычисление выражений вида $\alpha A + \beta B$, где $\alpha,\ \beta \in \{1, \ldots, q-1\}$, а $A, B \in E_{a,b}(\mathbb{F}_p),\ |E_{a,b}(\mathbb{F}_p)| = q$

Введем обозначения времени, необходимого для выполнения некоторых примитивов:

- τ_K - время, необходимое для вычисления кратной точки ЭК.

При тривиальной реализации вычисление выражения займет $2\tau_K$, однако при использовании алгоритма из [7], называемого также трюком Шамира, вычисление занимает уже $1.5\tau_K$ и дает еще больший выигрыш при большем количестве точек.

Дадим краткое описание алгоритму в случае суммы 2 кратных точек в предположении, что α, β имеют одинаковое число двоичных разрядов (если это не так, то просто дополним одно из чисел лидирующими нулями). Обозначим α_m – m-тый по старшинству бит числа α, β_m – m-тый по старшинству бит числа β. Алгоритм состоит из следующих шагов:

1. $R \leftarrow O$.
2. $m \leftarrow Highest$, где $Highest$ - максимальный номер единичного бита, выбираемый из 2 чисел(нумерация битов ведется с нуля).
3. $C \leftarrow A + B$.
4. `ПОКА` $m \geq 0$, `ВЫПОЛНЯТЬ` следующие шаги:
   (a) $R \leftarrow R + R$.
   (b) `ЕСЛИ` $\alpha_m = 1$ `И` $\beta_m = 1$, `ТО` $R \leftarrow R + C$.
   (c) `ЕСЛИ` $\alpha_m = 1$ `И` $\beta_m = 0$, `ТО` $R \leftarrow R + A$.
   (d) `ЕСЛИ` $\alpha_m = 0$ `И` $\beta_m = 1$, `ТО` $R \leftarrow R + B$.
   (e) $m \leftarrow m - 1$.
5. Возвратить R в качестве результата.

Данный алгоритм реализуется в нашей схеме без изменений.

4 Проектирование программного интерфейса

Целью данной работы является эффективная реализация двух действующих белорусских стандартов. Реализацию можно разделить на два этапа:

1. Реализация арифметики.

2. Использование арифметики и реализация программного интерфейса стандартов.

В настоящем разделе будет рассмотрен второй этап. Перед программным интерфейсом поставим следующие задачи:

- Универсальность. Под этим мы будем понимать:

  1. Работа на внешнем уровне с типами данных, которые имеются в наибольшем числе современных архитектур и языков программирования.
  2. Кроссплатформенность интерфейса.

- Единость программного стиля. Под этим будем понимать единую общую структуру методов и переменных.

- Наличие спецификаций. Под этим будем понимать некоторые надстройки над интерейсом, позволяющие эффективно решать задачи по внедрению интерфейса в другие библиотеки и модули.

Рассмотрим каждый из пунктов подробнее.

4.1 Универсальность

В данный момент единственными двумя инвариантами хранения данных независимо от архитектуры и языка являются биты и байты. Мы будем использовать байты, а точнее массивы байт. Таким образом, все параметры методов нашего интерфейса можно разделить на три типа:

- $\langle$ `byte []data ,` $[in]\rangle$ `параметр` . Под этим будем понимать массив байт, поступающий как входной параметр.

- $\langle$ `byte []outData,` $[out]\rangle$ `параметр` . Под этим будем понимать массив байт, поступающий на вход методу для записи в него результата.

- $\langle$ `byte []changableData,` $[in/out]\rangle$ `параметр` .Под этим будем понимать массив байт, который будет использоваться в ходе выполнения метода и содержимое которого может измениться.

Наиболее мощным языком, позволяющим решать задачи любого уровня практически на всех платформах является C++, что и обеспечило наш выбор именно этого языка для разработки интерфейса. Наряду с массивами байт в методы будут также передаваться $[in]$ `параметры` длины машинного слова для передачи информации о размере $\langle$ `byte []data ,` $[in]\rangle$ параметров.

4.2 Единость программного стиля

Будем придерживаться следующего стиля заголовков методов:

- Первыми идут $\langle$ `byte []data ,` $[in]\rangle$ `параметры` , за каждым из которых следует его длина (в случае, если длина не является константой и заранее известной).

- Далее идут параметры, контекст которых может быть изменен. Правило про длину остается прежним.

- В конце будут идти [*out*] `параметры`.

Название методов будем формировать следующим образом: *STB* _ *NAME*, где

- *STB* - название стандарта, в котором определен алгоритм, реализуемый методом.
- *NAME* - название алгоритма из этого стандарта.

4.3 Наличие спецификаций

Зачастую возникает ситуация, когда для внедрения интерфейса в некую библиотеку (к примеру,`OpenSSL`) необходимо строгое соотвествие ее определенным типам данных, среди которых могут быть:

- Контекст алгоритма (к примеру, шифрования или хэширования).
- Структуры личных и открытых ключей.
- Структуры, обеспечивающие работу с данными (аналог массивов байт в нашем интерфейсе).

Для того, чтобы решить эту проблему, необходимо разработать единый подход, который, в свою очередь, будет иметь конкретную реализацию для каждого случая.

В данной работе предлагается следующий подход:

- Выбирается множество методов, необходимых для внедрения.
- Создается новое множество методов, созданное на основе методов из пункта 1. Имена методов формируются по правилу *LIB* _ *OLDNAME*, где
  - *LIB* - название библиотеки, в которую происходит внедрение.
  - *OLDNAME* - название алгоритма из пункта 1.
- Сигнатура методов из пункта 2 будет отличаться от сигнатуры методов пункта 1 следующим образом:
  - Входными и выходными параметрами будут служить типы библиотеки, в которую происходит внедрение.
  - За каждым таким входным и выходным параметров будет следовать объект интерфейса `CONVERT`, который будет реализовывать конвертацию из типов в массив байт, и наоборот.

Для большего понимания приведем пример подобной реализации.

```


/* фрагмент сигнатур методов нашего интерфейса */

extern "C" void belt_ctr(byte *X, uint32 x_size, byte *sigma, byte *s, byte *to);
/* X, x_size, sigma - [in] параметры
     s - [in/out] параметры
     to - [out] параметр
*/

/*****************************************************/

/*фрагмент сигнатур библиотеки */

extern "C" void encrypt_step( ELEMENATRY_TYPE *data, CYPHER_ALG_CTX *ctx);

```

```
/*************************************************/

/* фрагмент надстройки */

class LIB_CONVERT {
public:
    static void convert_to_our_data(ELEMENTARY_TYPE *data, byte &*to, uint32 *to_size);
    static void convert_to_our_context(CYPHER_ALG_CTX *data, byte &*to, byte &*to1, byte&* t
    static void convert_to_their_data(byte *data, uint32 data_size, ELEMENTARY_TYPE *to);
    static void convert_to_their_context(byte *data, byte *data1, byte *data2,CYPHER_ALG_CTX
};

extern "C" lib_belt_ctr(ELEMENATRY_TYPE *data, CYPHER_ALG_CTX *ctx) {
    byte *t1, t2, t3, t4;
    uint32 size;
    LIB_CONVERT::convert_to_our_data(data, t1, size);
    LIB_CONVERT::convert_to_our_context(ctx, t2, t3, t4);
    belt_ctr(t1, size, t2, t3, t4);
    LIB_CONVERT::convert_to_their_data(t1, size, data);
    LIB_CONVERT::convert_to_our_context(t2, t3, t4, ctx);
}
```

example.cpp

4.4 Общие положения

Сформулированные выше три требования позволяют создать мощный интерфейс, который может как быть и универсальным, так и служить каким-то конкретным целям. Под последним понимается реализация интерфейса изначально для другого модуля. Подобное требование может быть вызвано ограниченностью ресурсов: как временных, когда необходимо будет максимально слаженное взаимодействие, так и ресурсов памяти, когда, к примеру, размеры всех массивов будут заранее известны, и будет экономиться память на передаче их длин. Пример подобной реализации будет приведен в приложении, где демонстрируется реализация интерфейса для проекта гражданской карты AvToken Smart.

5 Заключение

В настоящей работе были рассмотрены проблемы арифметики в некоторых алгебарических структурах: рассмотрены как и с теоретической стороны, так и с практической, что делает наши решения достаточно гибкими. Был поставлен вопрос о создании интерфейса для реализации стандартов шифрования и ЭЦП в Республике Беларусь, и на него был получен ответ в виде мощного аппарата, который может служить самым разнообразным целям. В приложении к работе содержится программная реализация интерфейса для нового криптографического проекта нашей страны - гражданской карты. Приложение написано на языке `C++`.

Список литературы

[1] D. Knuth. The Art of Computer Programming, Volume 2. Third Edition, Addison-Wesley, 1997. Section 4.3.3.A: Digital methods, pg.294.

[2] Richard P. Brent, Pierrick Gaudry, Emmanuel Thome, Paul Zimmermann. Faster Multiplication in GF(2)[x]

[3] Nicolas Meloni, Christophe Negre and M. Anwar Hasan1. High Performance GHASH Function for Long Messages. University of Waterloo, Canada

[4] David A. McGrew. The Galois/Counter Mode of Operation (GCM). San Jose, CA.

[5] Ayad F.Barsoum and M.Anwar Hasan. On Implementation of Quadratic and Sub-Quadratic Complexity Multipliers using Type II Optimal Normal Bases. Department of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada.

[6] CETIN K. KOC AND TOLGA ACAR . Montgomery Multiplication in GF(2k). In: Designs, Codes and Cryptography, 14(1), 57–69 (April 1998).

[7] Strauss: Addition chains of vectors. American Mathematical Monthly 71(7), 806–808, 1964.

[8] Neal Koblitz. Introduction to Elliptic Curves and Modular Forms.

[9] Проект СТБ "Информационные технологии и безопасность. Алгоритмы выработки и проверки электронной цифровой подписи на основе элиптических кривых".

[10] СТБ "Информационные технологии. Защита информации. Криптографические алгоритмы шифрования и контроля целостности".

A Реализация нашей схемы на языке C++. Исходный код

```cpp
#include "belt.h"

static void phi1(uint32* u) {
    uint32 t = u[0]^u[1];
    for (size_t i = 2; i >=0; --i) u[i] = u[i + 1];
    u[3] = t;
}

static void phi2(uint32* u) {
    uint32 t = u[0]^u[3];
    for (size_t i = 1; i <= 3; ++i) u[i] = u[i - 1];
    u[0] = t;
}
static void psi(uint32* u, uint32 at) {
    uint32 _=at>>2;
    uint32 __=at&3;
    __=(3-__)&3;
    *(((byte*)(u+_))+__)=0x80;
}




template<int R> uint32 RotHi(uint32 u) {
    return (u << R) | (u >> (32 - R));
}
template<int R> uint32 RotLo(uint32 u) {
    return (u >> R) | (u << (32 - R));
}
template<int R> uint32 G(uint32 u) {
    uint32 ret = 0, t;
    ret |= (t = H[u&0xFF]);
    u >>= 8;
    ret |= (t = H[u&0xFF])<<8;
    u >>= 8;
    ret |= (t = H[u&0xFF])<<16;
    u >>= 8;
    ret |= (t = H[u&0xFF])<<24;
    change_endian((byte*)&ret);
    ret = RotHi<R>(ret);
    change_endian((byte*)&ret);
    return ret;
}


static uint32 plus_belt(const uint32& a, const uint32 &b) {
    uint32 aa = a;
    change_endian((byte*)&aa);
    uint32 bb = b;
    change_endian((byte*)&bb);
    uint32 tt = aa + bb;
    change_endian((byte*)&tt);
    return tt;
}


static uint32 minus_belt(const uint32&a, const uint32 &b) {
    uint32 aa = a;
    change_endian((byte*)&aa);
    uint32 bb = b;
    change_endian((byte*)&bb);
```

```
    uint32 tt;
    if (aa >= bb) tt = aa - bb;
    else {
        uint64 t = (1ULL) << 32;
        t += aa;
        t -= bb;
        tt= t;
    }
    change_endian((byte*)&tt);
    return tt;
}
static uint32 eval(uint32 r) {
    return (r - 1)& 7;
}
static void encrypt_block(uint32* X, uint32 *Y, uint32 *sigma) {
    uint32 a = *X, b=*(X + 1), c= *(X + 2), d = *(X + 3),e;
    for (uint32 i = 1; i<= 8; ++i) {
        b ^= G<5>(plus_belt(a, *(sigma + eval(7*i - 6))));
        c ^= G<21>(plus_belt(d, *(sigma + eval(7*i -5))));
        a = minus_belt(a, G<13>(plus_belt(b, *(sigma + eval(7*i - 4)))));
        uint32 t_i = i;
        change_endian((byte*)&t_i);
        e =G<21>(plus_belt(plus_belt(b,c), *(sigma + eval(7 * i - 3)))) ^ t_i;
        b=plus_belt(b,e);
        c=minus_belt(c,e);
        d = plus_belt(d, G<13>(plus_belt(c, *(sigma + eval(7*i -2)))));
        b ^= G<21>(plus_belt(a, *(sigma + eval(7 * i - 1))));
        c ^= G<5>(plus_belt(d, *(sigma + eval(7*i))));
        a ^=b, b^=a, a^=b;
        c ^=d, d ^=c, c ^=d;
        b ^=c, c ^= b, b ^= c;
    }
    *Y = b;
    *(Y + 1) = d;
    *(Y + 2) = a;
    *(Y + 3) = c;
}

static void decrypt_block(uint32* X, uint32 *Y, uint32 *sigma) {
    uint32 a = *X, b=*(X + 1), c= *(X + 2), d = *(X + 3),e;
    for (uint32 i = 8; i>= 1; --i) {
        b ^= G<5>(plus_belt(a, *(sigma + eval(7*i))));
        c ^= G<21>(plus_belt(d, *(sigma + eval(7*i -1))));
        a = minus_belt(a, G<13>(plus_belt(b, *(sigma + eval(7*i - 2)))));
        uint32 t_i = i;
        change_endian((byte*)&t_i);
        e =G<21>(plus_belt(plus_belt(b,c), *(sigma + eval(7 * i - 3)))) ^ t_i;
        b=plus_belt(b,e);
        c=minus_belt(c,e);
        d = plus_belt(d, G<13>(plus_belt(c, *(sigma + eval(7*i -4)))));
        b ^= G<21>(plus_belt(a, *(sigma + eval(7 * i - 5))));
        c ^= G<5>(plus_belt(d, *(sigma + eval(7*i -6))));
        a ^=b, b^=a, a^=b;
        c ^=d, d ^=c, c ^=d;
        a ^=d, d ^= a, a ^= d;
    }
    *Y = c;
    *(Y + 1) = a;
    *(Y + 2) = d;
    *(Y + 3) = b;
}
static void belt_ecb_encr(byte *XX, uint32 size, byte *Sigma, byte *to) {
    //size in bytes
    uint32 act_sz = ((size - 1) / 16 + 1) * 4;
```

```
    uint32 *X = new uint32[act_sz];
    uint32 *Y = new uint32[act_sz];
    uint32 byteSZ = act_sz << 2;
    uint32* sigma = (uint32*)Sigma;
    memcpy(X, XX, size);
    for (size_t i = 0; i < byteSZ; i += 4) {
        byte*l = ((byte*)X) + i,
            *r = ((byte*)X) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)X) + i + 1,
            r = ((byte*)X) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }
    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }
    //here goes block with r
    if ((size  & 15) == 0) {
        for (size_t i = 0; i < act_sz; i += 4) {
            encrypt_block(X + i, Y + i,sigma);
        }
    } else {
        for (size_t i = 0; i < act_sz - 8; i += 4) {
            encrypt_block(X + i, Y + i, sigma);
        }
        encrypt_block(X + (act_sz - 8), Y + act_sz - 4, sigma);
        uint32 diff = byteSZ - size; //in bytes
        uint32 diff2 = diff / 4 ; //in ints
        uint32 at = (size - 1) >> 1;
        for (size_t jj = 0; jj<diff2; ++jj) {
            X[at+jj] = Y[at+jj];
        }
        for (size_t jj = 0; jj < diff; ++jj) {
            *(((byte*)(X + act_sz - 1)) + jj) = *(((byte*)(Y + act_sz - 1)) + jj);
        }
        encrypt_block(X + (act_sz - 4), Y + (act_sz - 8), sigma);
    }
    for (int i = 0; i < act_sz; ++i) {
        change_endian((byte*)(Y + i));
    }
    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }
    memcpy(to, ((byte*)Y), size);
    delete X;
    delete Y;
}
```

```
static void belt_ecb_decr(byte *XX, uint32 size, byte *Sigma, byte *to){
    //size in bytes
    uint32 act_sz = ((size - 1) / 16 + 1) * 4;
    uint32 *X = new uint32[act_sz];
    uint32 *Y = new uint32[act_sz];
    uint32 byteSZ = act_sz << 2;
    uint32* sigma = (uint32*)Sigma;
    memcpy(X, XX, size);
    for (size_t i = 0; i < byteSZ; i += 4) {
        byte*l = ((byte*)X) + i,
            *r = ((byte*)X) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)X) + i + 1,
            r = ((byte*)X) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }
    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }

    //here goes block with r
    if ((size  & 15) == 0) {
        for (size_t i = 0; i < act_sz; i += 4) {
            decrypt_block(X  + i, Y + i,sigma);
        }
    } else {
        for (size_t i = 0; i < act_sz - 8; i += 4) {
            decrypt_block(X  + i, Y + i, sigma);
        }
        decrypt_block(X + (act_sz - 8), Y + act_sz - 4, sigma);
        uint32 diff = byteSZ - size; //in bytes
        uint32 diff2 = diff / 4 ; //in ints
        uint32 at = (size - 1) >> 1;
        for (size_t jj = 0; jj<diff2; ++jj) {
            X[at+jj] = Y[at+jj];
        }
        for (size_t jj = 0; jj < diff; ++jj) {
            *(((byte*)(X + act_sz - 1)) + jj) = *(((byte*)(Y + act_sz - 1)) + jj);
        }
        decrypt_block(X + (act_sz - 4), Y + (act_sz - 8), sigma);
    }
    for (int i = 0; i < act_sz; ++i) {
        change_endian((byte*)(Y + i));
    }
    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }
    memcpy(to, ((byte*)Y), size);
```

```
    delete X;
    delete Y;
}


static  void belt_ctr(byte *XX, uint32 size, byte *Sigma, byte *S, byte *to){
    uint32 ss[4], ss2[4];
    uint32 *sigma = (uint32*)Sigma;

    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }

    uint32 act_sz = ((size - 1) / 16 + 1) * 4;
    uint32 *X = new uint32[act_sz];
    uint32 *Y = new uint32[act_sz];
    uint32 byteSZ = act_sz << 2;
    memset(X, 0U, sizeof X);
    memcpy(X, XX, size);
    for (size_t i = 0; i < byteSZ; i += 4) {
        byte*l = ((byte*)X) + i,
            *r = ((byte*)X) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)X) + i + 1,
            r = ((byte*)X) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }
    uint32 S2[4];
    memcpy(S2, S, 16);
    for (size_t jj = 0; jj < 4; ++jj) change_endian((byte*)(S2 + jj));
    encrypt_block(S2, ss, sigma);
    byte hlp[16];
    memcpy(hlp, ss, sizeof hlp);
    for (size_t jj = 0; jj < 16; jj += 4) change_endian(hlp+jj);

    BigInteger s;
    s.length = 4;
    for (size_t jj =0 ; jj < 4; ++jj) s.data[jj] = ss[jj], change_endian((byte*)(s.data + jj
    BigInteger one(1);
    change_endian((byte*)(one.data));
    for (size_t i = 0; i < act_sz; i += 4) {
        s += one;
        s.reduce(4);
        for (size_t j = 0; j < 4; ++j) ss2[j] = s.data[j],change_endian((byte*)(ss2 + j));;
        encrypt_block(ss2, ss, sigma);
        for (size_t j = 0; j < 4; ++j) Y[i + j] = X[i + j] ^ ss[j];
    }
    for (int i = 0; i < act_sz; ++i) {
        change_endian((byte*)(Y + i));
    }
    memcpy(to, Y, size);


    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
```

```
                *r = ((byte*)sigma) + 3 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
            l = ((byte*)sigma) + i + 1,
                r = ((byte*)sigma) + 2 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
        }

        delete X;
        delete Y;
}

static void belt_mac(byte *XX, uint32 size, byte *Sigma, byte *to) {
        uint32* sigma = (uint32*)Sigma;
        uint32 act_sz = ((size - 1) / 16 + 1) * 4;
        uint32 *X = new uint32[act_sz];
        uint32 *Y = new uint32[act_sz];
        uint32 byteSZ = act_sz << 2;
        memset(X, 0x00, sizeof X);
        for (size_t jj = 0; jj < act_sz; ++jj) X[jj] ^= X[jj];
        memcpy(X, XX, size);
        for (size_t i = 0; i < byteSZ; i += 4) {
            byte*l = ((byte*)X) + i,
                *r = ((byte*)X) + 3 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
            l = ((byte*)X) + i + 1,
                r = ((byte*)X) + 2 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
        }
        for (size_t i = 0; i < 32; i += 4) {
            byte*l = ((byte*)sigma) + i,
                *r = ((byte*)sigma) + 3 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
            l = ((byte*)sigma) + i + 1,
                r = ((byte*)sigma) + 2 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
        }
        uint32 s[4], r[4];
        memset(s, 0, sizeof s);
        encrypt_block(s, r, sigma);

        for (size_t i = 0; i < act_sz - 4; i += 4) {
            for (size_t j = 0; j < 4; ++j) X[i + j] ^= s[j];
            encrypt_block(X + i, s, sigma);
        }
        uint32 diff = byteSZ - size;
        if (!diff) {
            phi1(r);
            for (size_t i = 0; i < 4; ++i) s[i] ^= r[i] ^ X[act_sz - 4 + i];
        } else {
            psi(X + act_sz - 4,16- diff);
            phi2(r);
            for (size_t i = 0; i < 4; ++i) s[i] ^= r[i] ^ X[act_sz - 4 + i];
        }
        encrypt_block(s, r, sigma);
        for (size_t jj = 0; jj < 4; ++jj) change_endian((byte*)(r + jj));
        memcpy(to, r, 8);
        for (size_t i = 0; i < 32; i += 4) {
            byte*l = ((byte*)sigma) + i,
```

```
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }
    delete X;
    delete Y;
}

static void sigma1(uint32 *U, uint32 *u){
    uint32 u34[4];
    for (size_t i =0; i < 4; ++i) u34[i] = U[8 + i] ^ U[12 + i];
    encrypt_block(u34, u, U);
    for (size_t i = 0; i < 4; ++i) u[i] ^= u34[i];
}

static void sigma2(uint32 * U, uint32 *u) {
    uint32 SIGMA1[8], SIGMA2[8];
    sigma1(U, SIGMA1);
    for (size_t i = 0; i < 4; ++i) SIGMA1[4 + i] = U[12 + i];
    sigma1(U, SIGMA2);
    for (size_t i = 0; i < 4; ++i) SIGMA2[4 + i] = U[8 + i], SIGMA2[i] ^= (uint32)(((1ULL)<<
    encrypt_block(U, u, SIGMA1);
    for (size_t i = 0; i < 4; ++i) u[i] ^= U[i];
    encrypt_block(U + 4, u + 4, SIGMA2);
    for (size_t i = 0; i < 4; ++i) u[4 + i] ^= U[4 + i];
}


static  void belt_hash_step(byte* XX, uint32 cur_len, byte *STATE, byte *to) {
    uint32 s[4];
    memset(s, 0, sizeof s);
    uint32 act_sz = 8;
    byte  h[32] = { 0xB1 ,0x94 ,0xBA ,0xC8 ,0x0A ,0x08 ,0xF5 ,0x3B ,0x36 ,0x6D ,0x00 ,0x8E ,
    if (cur_len == 0 && XX != NULL) {
        memcpy(STATE + 16, s, sizeof s);
        memcpy(STATE + 16 + sizeof s, h, sizeof h);
        memset(STATE , 0, 16);
        return;
    } else {
        if (XX == NULL) {
            uint32 tmp3[8];
            uint32 tmp1[16];
            memcpy(tmp1, STATE, sizeof tmp1);
            for (size_t  i = 0; i < 16; ++i) change_endian((byte*)(tmp1 + i));
            sigma2(tmp1, tmp3);
            memcpy(to, tmp3, 32);
            return;
        }
        uint32 *X = new uint32[act_sz];
        uint32 byteSZ = act_sz << 2;
        memcpy(X, XX, cur_len);


        memcpy(s, STATE + 16, sizeof s);
        memcpy(h, STATE + sizeof s + 16, sizeof h);
        for (size_t i = 0; i < byteSZ; i += 4) {
            byte*l = ((byte*)X) + i,
                *r = ((byte*)X) + 3 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
```

```c
            l = ((byte*)X) + i + 1,
                r = ((byte*)X) + 2 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
        }
        for (size_t i = 0; i < 16; i += 4) {
            byte*l = ((byte*)s) + i,
                *r = ((byte*)s) + 3 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
            l = ((byte*)s) + i + 1,
                r = ((byte*)s) + 2 + i;

            *r ^= *l, *l ^= *r, *r ^= *l;
        }
        uint64 X_LEN;
        memcpy(&X_LEN, STATE + 8, 8);
        X_LEN += cur_len << 2;
        uint32 tmp1[4], tmp2[8], tmp3[16];
        for (size_t j = 0; j < 8; ++j) tmp3[j] = X[j];
        for (size_t j = 0; j < 32; ++j) *(((byte*)(tmp3+8)) + j) = h[j];
        sigma1(tmp3, tmp1);
        for (size_t j = 0; j < 4; ++j) s[j] ^= tmp1[j];
        sigma2(tmp3, tmp2);
        for (size_t j = 0; j < 8; ++j) change_endian((byte*)(tmp2 + j));
        memcpy(STATE + 16, s, sizeof s);
        memcpy(STATE + 16 + sizeof s, tmp2, sizeof h);
        memcpy(STATE + 8, &X_LEN, 8);
    }
}

static void belt_hash(byte *XX, uint32 size, byte *to) {
    byte STATE[64];
    belt_hash_step(XX, 0,STATE);
    uint32 actSIZE = 0;
    while (actSIZE + 8 < size) {
        belt_hash_step(XX + actSIZE, 8, STATE);
        actSIZE += 8;
    }
    size-=actSIZE;
    belt_hash_step(XX + actSIZE, size, STATE, to);
    belt_hash_step(NULL, 0, STATE, to);
}

static void belt_keyrep(byte *X, byte b, byte *to) {
    uint32 n = 256, m = 256;
    byte D[12];
    memset(D, 0, sizeof D);
    byte I[16];
    memset(I, 0, sizeof I);
    I[0] = b;
    uint32 r = 0x7B653CF3;
    change_endian((byte*)&r);
    uint32 preY[16];
    preY[0] = r;
    memcpy(preY + 1, D, sizeof D);
    memcpy(preY + 4, I, sizeof I);
    memcpy(preY + 8, X, 32);
    for (size_t jj = 1; jj < 16; ++jj) change_endian((byte*)(preY +jj));
    uint32 Y[8];
    sigma2(preY, Y);
    memcpy(to, Y, sizeof Y);
}
```

```
static void belt_keywrap(byte *X, uint32 len, byte *Sigma, byte *to) {
    uint32 *r = new uint32[ (sizeof X - 1) / 4 + 1 + 16];
    memset(r, 0x00, sizeof r);
    memcpy(r, X, sizeof X);
    for (uint32 jj =0 ; jj < ((sizeof X - 1) / 4 + 1); ++jj) change_endian((byte*)(r + jj));
    uint32 s[4];
    uint32 *sigma = (uint32*)Sigma;

    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }

    uint32 n = sizeof r >> 2;
    for (uint32 step = 1; step <= (n << 1); ++step) {
        for (uint32 aa=0;aa<4;aa++) {
            s[aa] = 0x00000000;
            for (uint32 bb=aa;bb<n - 4;bb+=4)s[aa]^=r[bb];
        }
        uint32 h[4];
        encrypt_block(s, h, sigma);
        for (size_t qq=0;qq<4;++qq) r[n - 4 + qq] ^= h[qq]^ step;
        for (size_t jj=4;jj<n;++jj)r[jj-4]=r[jj];
        for (size_t jj=0;jj<4;++jj)r[n-4+jj]=s[jj];
    }
    for (uint32 tt=0;tt<n;++tt) change_endian((byte*)(r+tt));
    memcpy(to, r, sizeof r);
    delete r;
}

static void belt_keyuwrap(byte *X,uint32 len,  byte *Sigma, byte *to) {
    uint32 *r = new uint32[ (sizeof X - 1) / 4 + 1 + 16];
    memset(r, 0x00, sizeof r);
    memcpy(r, X, sizeof X);
    for (uint32 jj =0 ; jj < ((sizeof X - 1) / 4 + 1); ++jj) change_endian((byte*)(r + jj));
    uint32 s[4];
    uint32 *sigma = (uint32*)Sigma;

    for (size_t i = 0; i < 32; i += 4) {
        byte*l = ((byte*)sigma) + i,
            *r = ((byte*)sigma) + 3 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
        l = ((byte*)sigma) + i + 1,
            r = ((byte*)sigma) + 2 + i;

        *r ^= *l, *l ^= *r, *r ^= *l;
    }

    uint32 n = sizeof r >> 2;
    for (uint32 step = n << 1; step <(n << 3); --step) {
        for (size_t jj=0;jj<4;++jj)s[jj] =r[n-4+jj];
        for (size_t jj=0;jj<n-4;++jj)r[jj+4]=r[jj];
        for (uint32 aa=0;aa<4;aa++) {
            s[aa] = 0x00000000;
            for (uint32 bb=aa;bb<n - 4;bb+=4)s[aa]^=r[bb];
        }
        uint32 h[4];
```

```
        encrypt_block(s, h, sigma);
        for (size_t qq=0;qq<4;++qq) r[n - 4 + qq] = h[qq]^ step;
        for (uint32 aa=0;aa<4;aa++) {
            r[aa] = s[aa];
            for (uint32 bb=aa + 4;bb<n - 4;bb+=4)r[aa]^=r[bb];
        }
    }
    for (uint32 tt=0;tt<n;++tt) change_endian((byte*)(r+tt));
    memcpy(to, r, sizeof r);
    delete r;
}
```

belt.cpp