НАЦІОНАЛЬНИЙ ТЕХНІЧНИИ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО» ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ ЗАХИСТУ ІНФОРМАЦІЇ

«На правах рукопису»
УДК _____

«До захисту допущено»
В.о. завідувача кафедрою
_____ М.М.Савчук
(підпис) (ініціали, прізвище)
“ ___” _____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 113 "Прикладна математика"

на тему: Побудова постквантової системи захищеного обміну повідомленнями з використанням ізогеній еліптичних кривих

Виконав (-ла): студент (-ка)

Грубіян Євгеній

(прізвище, ім’я, по батькові) (підпис)

Керівник. Фесенко А.В. к.ф.-м.н., ст. викладач
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент.
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2018 року

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

Фізико-технічний інститут Кафедра математичних методів захисту інформації Рівень вищої освіти: другий (магістерський) за освітньо-професійною програмою

Спеціальність: 113 «Прикладна математика»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедрою
_____ М.М.Савчук
(підпис) (ініціали, прізвище)
«___ » _____201_ р.

ЗАВДАННЯ на магістерську дисертацію студенту

(прізвище, ім'я, по батькові)

1. Тема дисертації _____

науковий керівник дисертації _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від _____ р. № _____

2. Термін подання студентом дисертації_____

3. Об'єкт дослідження _____

4. Предмет дослідження (Вхідні дані - для магістерської дисертації за освітньо-професійною програмою)

5. Перелік завдань, які потрібно розробити

6. Орієнтовний перелік ілюстративного матеріалу

7. Орієнтовний перелік публікацій

8. Консультанти розділів дисертації*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання

Календарний план

=	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка

Студент _____ _____
(підпис) (ініціали, прізвище)

Науковий керівник дисертації_____ _____
(підпис) (ініціали, прізвище)

' Консультантом не може бути зазначено наукового керівника магістерської дисертації.

Роботу виконано на 82 аркушах, вона містить 1 додаток та перелік посилань на використані джерела з 21 найменувань.

Метою дипломної роботи є дослідження можливості ефективної практичної реалізації алгоритму постквантового ключового обміну SIDH з використанням еліптичних кривих в формі Едвардса.

Об'єктом дослідження є алгоритм постквантового ключового обміну на основі ізогеній суперсингулярних еліптичних кривих *SIDH*.

Предметом дослідження є можливість ефективної практичної реалізації алгоритму постквантового ключового обміну SIDH з використанням еліптичних кривих в формі Едвардса.

В роботі зроблено огляд останніх публікацій по темі постквантових алгоритмів на основі ізогеній суперсингулярних еліптичних кривих, зокрема розглянуто алгоритм SIDH та можливість його ефективної реалізації з використанням еліптичних кривих в формі Едвардса, розроблена реалізація компонентів алгоритму SIDH з використанням еліптичних кривих в формі Едвардса мовою C++ та проведений аналіз можливості використання кривих Едвардса в постквантових алгоритмах на основі ізогеній суперсингулярних еліптичних кривих. В ході аналізу були виявлені деякі проблеми, для яких були запропоновані шляхи їх розв'язання.

СУПЕРСИНГУЛЯРНІ ЕЛІПТИЧНІ КРИВІ, ІЗОГЕНІЇ, ЕЛІПТИЧНІ КРИВІ В ФОРМІ ЕДВАРДСА, АЛГОРИТМ SffiH, ПОСТКВАНТОВИЙ ОБМІН КЛЮЧАМИ

Дипломная работа выполнена на 82 листах, она содержит 1 приложение и список ссылок на использованные источники с 21 наименований.

Целью дипломной работы является исследование возможности эффективной практической реализации алгоритма постквантового ключевого обмена SIDH с использованием эллиптических кривых в форме Эдвардса.

Объектом исследования является алгоритм постквантового ключевого обмена на основе изогений суперсингулярных эллиптических кривых *SIDH*.

Предметом исследования является возможность эффективной практической реализации алгоритма постквантового ключевого обмена SIDH с использованием эллиптических кривых в форме Эдвардса.

В работе сделан обзор последних публикаций по теме постквантових алгоритмов на основе изогений суперсингулярных эллиптических кривых, в частности рассмотрен алгоритм SIDH и возможность его эффективной реализации с использованием эллиптических кривых в форме Эдвардса, разработана реализация компонентов алгоритма SIDH с использованием эллиптических кривых в форме Эдвардса на языке C++ и проведен анализ возможности использования кривых Эдвардса в постквантових алгоритмах на основе изогений суперсингулярних эллиптических кривых. В ходе анализа были обнаружены некоторые проблемы, для которых были предложены пути решения.

СУПЕРСИНГУЛЯРНЫЕ ЭЛЛИПТИЧЕСКИЕ КРИВЫЕ, ИЗОГЕНИИ, ЭЛЛИПТИЧЕСКИЕ КРИВАЯ В ФОРМЕ ЭДВАРДСА, АЛГОРИТМ SIDH, ПОСТКВАНТОВЫЙ ОБМЕН КЛЮЧАМИ

ABSTRACT

The thesis is presented in 82 pages. It contains 1 appendix and bibliography of 21 references.

The target of the thesis is to study the feasibility of effective practical implementation of the SIDH quantum-resistant key exchange algorithm using elliptic curves in Edwards form.

The object is quantum-resistant key exchange algorithm *SIDH*.

The subject is possibility of effective practical realization of the *SIDH* algorithm using elliptic curves in Edwards form.

The paper reviews recent publications on the topic of quantum-resistant cryptographic algorithms based on the isogenies of supersingular elliptic curves in particular, the SIDH algorithm and the possibility of its effective implementation using elliptic curves in Edwards form are considered. The implementation of components of the SIDH algorithm using elliptic curves in Edwards form was developed. An analysis of the possibility of using Edwards curves in quantum-resistant algorithms based on isogenies of supersingular elliptic curves was conducted. During the analysis some problems were identified for which solutions were proposed.

SUPERSINGULAR ELLIPTIC CURVES, ISOGENIES, ELLIPTIC CURVES IN EDWARDS FORM, SIDH ALGORITHM, QUANTUM-RESISTANT KEY EXCHANGE

ЗМІСТ

ВСТУП

Актуальність дослідження. Сьогодні в епоху інформаційних технологій, коли криптографія стала невід'ємною частиною інформаційних процесів постає проблема в надійності існуючих алгоритмів та розробці нових алгоритмів, що залишилися би стійкими до нових викликів. Один з них - це потенційна можливість створення квантового комп'ютера, що вирішував би надзвичайно ефективно деякий клас задач, зокрема задач, на складності яких базуються сьогоднішні асиметричні криптосистеми(И$А, ECC). Тому нещодавно NIST оголосив конкурс на постквантові алгоритми асиметричного шифрування та цифрового підпису. Одним з таких алгоритмів є SIDH(Supersmgular Isogeny Dffie-Hellman) - алгоритм постквантового ключового обміну на основі ізогеній суперсингулярних еліптичних кривих, що був запропонований в 2011 році групою дослідників з Версальського університету, а пізніше реалізований розробниками та дослідниками з Microsoft, реалізація яких лягла в основу постквантового алгоритму інкапсуляції ключів *SIKE,* що був поданий на конурс оголошений NIST та інтенсивно вивчається криптографами та математиками зі всього світу.

Метою дипломної роботи є дослідження можливості ефективної практичної реалізації алгоритму постквантового ключового обміну SIDH з використанням еліптичних кривих в формі Едвардса. Для досягнення мети необхідно вирішити наступні завдання дослідження:

1) провести огляд опублікованих робіт за тематикою дослідження;

2) провесити аналіз можливості застосування еліптичних кривих в формі Едвардса до алгоритму SIDH;

3) розробити програмну реалізацію компонентів алгоритму, зокрема арифметики в полі F_{p^2}, арифметики на еліптичних кривих в формі Едвардса, обчислення 3,4-ізогеній кривих в формі Едвардса.

4) провести аналіз проблем, що виникли в ході дослідження та

запропонувати шляхи їх вирішення.

Об'єктом дослідження є алгоритм постквантового ключового обміну на основі ізогєній суперсингулярних еліптичних кривих *SIDH*

Предметом дослідження є можливість ефективної практичної реалізації алгоритму постквантового ключового обміну SIDH з використанням еліптичних кривих в формі Едвардса.

Наукова новизна роботи полягає у вперше проведеному аналізі застосування еліптичних кривих в формі Едвардса для повноформатної ефективної реалізації алгоритму постквантового ключового обміну SIDH. В ході дослідження були виявлені певні проблеми та запропоновано шляхи їх вирішення.

Практичне значення роботи полягає у можливості застосування результатів дослідження для побудови ефективніших алгоритмів постквантового ключового обміну на основі ізогеній суперсингулярних еліптичних кривих в формі Едвардса, оскільки арифметика кривих Едвардса, за деякими винятками, є швидшою та стійкішою до атак за побічними каналами, що робить її також цікавою в контексті реалізації на вбудованих пристроях, таких як смарт-карти, однокристальні чіпи з невеликими обсягами доступних обчислювальних ресурсів.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ

В розділі наведено огляд проблематики дослідження та основні теоретичні відомості.

1.1 Квантовий комп’ютер та небезпека для сучасної криптографії

Безпека сучасних інформаційних систем та технологій ґрунтується на стійкості криптографічних перетворень, які вони використовують для криптографічної обробки інформації. Криптографічна стійкість базується на складності розв’ язання певних математичних задач(факторизації великого цілого числа, розв’ язку дискретного логарифма тощо), для таких задач характерна субекспоненційна або експоненційна складність розв’ язання на сучасних (класичних) комп’ ютерах. Проте, використовуючи квантові алгоритми Шора [1] та Гровера [2], певні математичні задачі можна розв’ язувати навіть з поліноміальною складністю. Ідеї використання потужностей квантового середовища висунули Пауль і Фейнман [3]. Важливим стало розроблення у 1992 р. Дойтчем [4] та іншими першого квантового алгоритму, можливості якого значно перевищували можливості звичайних комп’ ютерів. У випадку появи квантового комп’ ютера, на якому може бути запущений квантовий алгоритм криптоаналізу Шора або алгоритм пошуку в неупорядкованій базі даних Гровера [2], можуть виникнути великі загрози у інформаційній сфері відносно забезпечення криптографічної стійкості як для асиметричних криптоперетворень, так і для певних симетричних. Важливим є не тільки сам факт побудови такого комп’ ютера, а й

технічні характеристики, якими володітиме квантовий комп’ ютер. Ураховуючи небезпеку, яку становлять квантові алгоритми, що можна застосувати для криптоаналізу сучасних криптосистем, за останні роки вже було запропоновано певні класи криптосистем, що будуть стійкими до квантового криптоаналізу на основі алгоритмів Шора та Гровера [5-8]. Крім цього, активні дослідження ведуться у напрямі квантових протоколів та квантових генераторів випадкових послідовностей.

У роботі [1] наведено опис квантового алгоритму Шора для знаходження періоду функції, який можна використати для факторизації цілих чисел та розв’ язання дискретного логарифмічного рівняння. Детальний аналіз можливостей алгоритму Шора наведено в роботі [9]. В роботі [2] Гровер навів квантовий алгоритм пошуку в неупорядкованій базі даних, аналіз можливостей використання алгоритму Гровера для проведення криптоаналізу сучасних криптосистем подано в роботі [10]. У роботах [9, 10] наведено оцінки стійкості сучасних асиметричних та симетричних криптосистем проти квантового криптоаналізу. В роботі [10] описано можливості використання алгоритму Гровера для криптоаналізу криптосистеми типу NTRU [6] та наведено оцінки складності криптоаналізу класичними та квантовими методами цієї системи. Після того, як Шор у 1994 р. запропонував поліноміальний алгоритм знаходження періоду функції, багато криптографів та математиків почали працювати у напрямі постквантової криптографії, тобто криптографії, яка існуватиме після появи квантового комп’ ютера [5]. Вже існують певні класи криптосистем, які за структурою будуть стійкими до квантового криптоаналізу на основі алгоритмів Шора та Гровера. Такі криптосистеми, як правило, презентуються та обговорюються на щорічній конференції PQCrypto з 2006 р. [7].

Тривалий час до створення квантового комп’ ютера ставились достатньо скептично, але у 2007 р. канадська компанія D-Wave презентувала свій 16-кубітний квантовий комп’ ютер, а вже у 2013 р. вона продала комп’ ютер із 512 кубітами Google та NASA[11]. Тобто за

достатньо невеликий проміжок часу колектив дослідників компанії D-Wave досяг значних успіхів у збільшенні кількості кубітів на процесорі квантового комп’ ютера. Створенням квантового комп’ ютера займаються багато наукових колективів, серед них такі як Каліфорнійський технологічний інститут, IBM, D-Wave Systems, Російський квантовий центр та Національний дослідницький технологічний інститут “ МГСіС ”. Кожний з колективів вже досяг значних успіхів у напрямі побудови такого комп’ ютера, але найвагоміші результати все ще у D- Wave, проте, як показує аналіз, комп’ ютер D-Wave не підходить для реалізації алгоритмів квантового криптоаналізу, тому що на ньому неможливо реалізувати алгоритми, які використовують квантові вентилі. Тобто вважається, що ні алгоритм Шора, ні алгоритм Гровера на ньому не можуть бути реалізованими [9,10]. Це пов’ язано з тим, що для обчислень використовується зовсім інший принцип - так звані адіабатичні квантові обчислення. Вказане значно обмежує його можливості, але дозволяє не турбуватися про декогеренції та інші проблеми, що характерні для звичайних квантових обчислювачів. Важливим є факт, що став відомим на початку 2014 року, на основі документів [12], що надав колишній підрядник Агентства національної безпеки США (АНБ) Едвард Сноуден. Документи свідчать про те, що АНБ запустило дослідницьку програму, на яку виділено $ 79,7 млн ., під назвою “Проникнення до важких цілей” з метою розроблення квантового комп’ ютера, здатного зламати шифрування, що є уразливим для квантових комп’ ютерів. Але варто зазначити, що на сьогоднішній день до створення справжнього квантового комп’ютера, здатного вирішувати практичні проблеми, як то факторизація цілих чисел чи дискретне логарифмування - ще далеко.

Дуже поширені сьогодні й часто застосовуються алгоритми асиметричного перетворення в кільці, в групі та в групі точок еліптичної кривої, прикладами таких криптосистем є RSA, DSA, ECC. Більшість атак на такі криптосистеми спрямовані на знаходження особистого ключа. Так, для криптосистеми RSA стійкість проти такої атаки

базується на складності факторизації модуля перетворення N. Усі відомі нині класичні алгоритми факторизації мають або експоненційну, або субекспоненційну складність. Вважається, що найкращим з погляду мінімізації складності факторизації є алгоритм загального решета числового поля або його модифікації. Часова складність таких алгоритмів оцінюється як субекспоненційна, наприклад, у вигляді [9]:

$$0(exp(1.92 + o(1)(lnN)^{1/3}(lnlnN)^{2/3}))$$

Водночас квантовий алгоритм Шора має поліноміальну складність. Він здатен розкласти складене число на прості множники приблизно за час: $0(4n^3)$ з використанням $0(2n)$ кубітів.

1.1.1 Алгоритм Шора

Основа алгоритму Шора: здатність інформаційних одиниць квантових комп'ютерів — кубітів — приймати кілька значень одночасно і перебувати в стані «квантової заплутаності». Роботу алгоритму Шора можна розділити на 2 частини: перша — класичне зведення розкладання на множники до знаходження періоду деякої функції, друга — квантове знаходження періоду цієї функції. Нехай:

M — число, яке ми хочемо розкласти на множники (воно не повинно бути цілим степенем простого числа);

N — розмір регістра пам'яті, який використовується (без врахування додаткової пам'яті). Бітовий розмір цієї пам'яті $n = log_2 N$ приблизно в 2 рази більше розміру M. Точніше, $M^2 < N = 2^n < 2M^2$.

t — випадковий параметр такий, що $1 < t < M$ і $gcd(t, M) = 1$, де gcd – найбільший спільний дільник).

Відзначимо, що t, N, M — фіксовані. В алгоритмі Шора

використовується стандартний спосіб зведення задачі факторизації до задачі пошуку періоду *г* функції від випадково підібраного числа t.

Класична частина алгоритму

Мінімальне *г* таке, що $t^r = 1 \bmod M$ — це порядок *t* по модулю *M*.

Порядок *г* є періодом функції $f(x) = t^x \bmod M$, де $x = 0,1,2,\ldots, N-1$ Якщо можна ефективно обчислити *г* як функцію від *t,* то можна знайти власний дільник *M* за час, обмежений поліномом від $\log_2 M$ з ймовірністю ^ 1 — M^{-m} для будь-якого фіксованого *m*.

Припустимо, що для даного *t* період *г* парний *(г* = 0 mod 2) і задовольняє умові *1*2 = —1 mod *M*. Тоді gcd(£2 + 1,M) — власний дільник *M*. Функція *gcd* вирішується за поліноміальний час.

Ймовірність виконання цієї умови ^ 1 — , де *к* — число різних непарних простих дільників *M*, отже, ^ *2* в даному випадку. Тому хороше значення *t* t з ймовірністю ^ 1 — M^{-m} знайдеться за 0(log *M*) спроб. Найдовше обчислення в одній спробі — обчислення t^2

Квантова частина алгоритму

Для здійснення квантової частини алгоритму необхідна обчислювальна схема, що складається з 2-х квантових регістрів *X* і *Y*. Спочатку, кожен з них складається із сукупності кубітів в нульовому булевому стані. |0).

Регістр *X* використовується для розміщення аргументів *x* функції *f (x)*. Регістр *Y* (допоміжний) використовується для розміщення значень функції *f (x)* з періодом *г,* що підлягають обчисленню.

Квантове обчислення складається з 4 кроків:

1) На першому кроці за допомогою операції Уолша - Адамара, яка здійснює перетворення кубіта за допомогою оператора

11 1

—1

H =

|0) регістра *X* перекладається в рівноймовірнісну суперпозицію всіх

булевих станів *X*. Другий регістр *Y* залишається в стані |0). В результаті виходить наступний стан для системи двох регістрів:

$$\frac{1}{\sqrt{2^N}} \sum_{x=0}^{N-1} |x,0\rangle$$

2) Нехай *Uf*— унітарне перетворення, яке переводить |ж,0) в \x,f(x)). На другому кроці застосовується унітарне перетворення до системи двох регістрів. Виходить наступний стан системи:

$$\frac{1}{\sqrt{2^N}} \sum_{x=0}^{N-1} |x,f(x)\rangle$$

тобто між станами обох регістрів утворюється певний зв'язок.

3) Квантове Фур'є-перетворення є унітарним перетворенням стану квантового регістра, що описується Ж-мірним вектором стану виду $\sum_{x=0}^{N-1} f(x)|x\rangle$ в інший стан $\sum_{k=0}^{N-1} f(k)|k\rangle$:

$$\text{QFT}_n : \sum_{x=0}^{N-1} f(x)|x\rangle \wedge \sum_{k=0}^{N-1} f(k)|k\rangle$$

де амплітуда перетворення Фур'є $f(x)$ має вигляд:

$$f^{(k)} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} \exp(2\pi i kx/N) f(x)$$

У двовимірній ж^-площині перетворення Фур'є відповідає повороту осей координат на 90, яке веде до перетворення шкали *x* в шкалу k. На третьому кроці над станом першого регістра здійснюється перетворення Фур'є, і виходить

$$\frac{1}{N} \sum_{x=0}^{N-1} \sum_{k=0}^{N-1} \exp(2\pi i kx/N)|k,t^x \bmod M\rangle$$

4) Четвертий крок. На четвертому кроці виконується вимірювання першого регістра *X* щодо ортогональної проекції виду:

$$|0,0\rangle \otimes I,\ |1,1\rangle \otimes I,\ \ldots,\ |N-1,N-1\rangle \otimes I$$

де I — тотожний оператор на гільбертовому просторі другого регістра Y. В результаті виходить $|k, t^k \bmod M\rangle$ з ймовірністю:

$$\left| \frac{1}{N} \sum_{x:\ t^x = t^k \bmod} \exp(2\pi i\ kx/N) \right|^2$$

На тій частині прогону, що залишилась, працює класичний комп'ютер: Знаходиться найкраще наближення (знизу) до ^ зі знаменником $r' < M < \sqrt{N}$:

$$\left| \frac{k}{N} - \frac{d'}{r'} \right| < \frac{1}{2N}$$

Пробуємо r' в ролі r:

- Якщо $r' = 0 \bmod 2$, то слід обчислити $gcd(t^{r'/2} \pm 1, M)$

- Якщо r' непарне, або якщо r' парне, але власний дільник M невиявлений, то слід повторити прогін $O(\log\log M)$ раз з тим же самим t. У разі невдачі, необхідно змінити t і почати новий прогін алгоритму.

В деякій мірі визначення періоду функції за допомогою перетворення Фур'є аналогічне вимірюванню періоду кристалічної ґратки методом рентгенівської або нейтронної дифракції. Щоб визначити період r не потрібно обчислювати всі значення $f(x)$. У цьому сенсі задача близька до алгоритму Дойча — Йожи, в якому важливо знати не всі значення функції, а тільки деякі її властивості.

1.1.2 Порівняльний аналіз квантового та класичного алгоритмів для факторизації

В розділі наведено порівняльний аналіз кількості ресурсів для факторизації чисел різного розміру в квантовій та класичній моделях. В якості алгоритмів для класичної моделі - використовується узагальнений алгоритм решета числового поля, а в квантовій - розглянутий вище алгоритм Шора.

Bit length of N	Qubits	Quantum gates	Classical complexity
512	1024	$0.54 \cdot 10^9$	$1.6 \cdot 10^{19}$
768	1536	$1.8 \cdot 10^{10}$	$9.9 \cdot 10^{22}$
1024	2048	$4.3 \cdot 10^9$	$1.2 \cdot 10^{26}$
2048	4096	$34 \cdot 10^9$	$1.3 \cdot 10^{35}$
3072	6144	$12 \cdot 10^{10}$	$5 \cdot 10^{41}$
15360	30720	$1.5 \cdot 10^{13}$	$9.2 \cdot 10^{80}$

Аналіз даних таблиці показує , що для зламу RSA криптосистеми з розміром модуля у 15360 бітів (розмір головного сертифіката відкритого ключа США) необхідно лише $1.5 \cdot 10^{13}$ операцій на квантовому комп’ ютері, тоді як класична обчислювальна система повинна виконати близько 10^{80} операцій . Тобто RSA система буде зламана за поліноміальний час. Крім того, як випливає із таблиці, навіть істотне збільшення модуля не врятовує RSA криптосистему від її злому.

1.1.3 Протидія постквантовому криптоаналізу

Основним напрямом протидії квантовим алгоритмам криптоаналізу є винайдення нових класів криптосистем , стійкість яких не ґрунтується на таких математичних проблемах, як складність факторизації цілого числа чи розв’ язку дискретного логарифма . Напрям таких досліджень названо терміном “постквантова криптографія”, тобто криптографія, яка буде стійкою навіть після появи квантового комп’ ютера , що матиме велику кількість кубітів для своєї роботи . Цей напрям робіт популяризувався після 2006 р ., саме тоді проведено першу конференцію за цією тематикою (PostQuantumCrypto 2006) [7]. Така конференція проводиться щорічно, щоб обговорити найкращі здобутки за рік у напрямі постквантової криптографії.

Що стосується симетричних криптосистем, то більшість сучасних симетричних шифрів та геш-функцій захищені від квантових комп’ ютерів. Квантовий алгоритм Гровера може прискорити криптоаналіз таких криптосистем , але йому можна протидіяти збільшенням розміру ключа чи блока повідомлення [10].

Отже, враховуючи останні досягнення у напрямі постквантової криптографії , можна виділити такі класи криптосистем , що будуть стійкими до квантового криптоаналізу [5]:

1) Криптографія на основі решіток.
2) Мультиваріативна криптографія.
3) Криптографія на основі геш-функцій.
4) Криптографія на основі кодів.
5) Симетрична криптографія.
6) Криптографія на основі ізогеній суперсингулярних еліптичних кривих.

В даній роботі ми розглядатимемо та будемо здійснювати синтез

криптосистеми придатної для захищеного обміну повідомленнями на основі суперсингулярних еліптичних кривих. Далі ми проведемо огляд ІСНУЮЧИХ джерел по криптосистемам на ОСНОВІ суперсингулярних елштичних кривих.

1.2 Еліптичні криві

Поняття еліптичної кривої в математиці не нове, перші використання еліптичних кривих присутні в роботах таких математиків як Діофант, К. Вейєрштрас, І. Ньютон, К. Гаус, проте еліптичні криві до недавнього часу не знаходили широкого практичного застосування, хоча розробка теорії досить активно велась різними математиками. Зокрема на початку 1980х років Кобліц та Мілер запропонували використовувати еліптичні криві в криптографії, запропонувавши аналог протоколу ключового обміну Діффі-Гелмана на основі еліптичних кривих ($ECDH$). В 1995 році сталась визначна подія, пов'язана з теорією еліптичних кривих - Ендрю Вайлс запропонував доведення 300-річної Великої Теореми Ферма про нерозвязність в цілих числах рівняння $x^n + y^n = z^n, n > 2$ за допомогою апарату еліптичних кривих.

Поняття еліптичної кривої можна визначити за допомогою апарату алгебраїчної та проективної геометрії більш широко, проте через надмірність викладок та деяку складність ми використаємо більш простіші алгебраїчні визначення. Для початку ми визначимо еліптичну криву в канонічній формі Вейєрштраса. В роботі ми використовуємо скінчені поля характеристики більше трьох, тому без втрати загальності будь яка еліптична крива над полем $F, char(F) > 3$ може бути представлена в канонічній формі Вейєрштраса.

Визначення 1.1. *Еліптичною кривою E в формі Вейєрштраса над*

скінченим полем F_q, $q = p^n$, $p > 3$ ми називаємо множину точок (x,y); $x,y \in F_q$, що задовольняють рівняння:

$$E: y^2 = x^3 + ax + b; a,Ь \in F_q$$

Таким чином, змінюючи параметри $a,Ь$ ми будемо отримувати нові єліптичні криві над скінченим полем. Для зручності еліптичну криву в формі Вейєрштраса з коефіцієнтами $a,Ь$ будемо позначати $E_{a,ь}$ Зазначимо, що взагалі кажучи визначення вище застосовне з деякою модифікацією(узагальнення форми та збільшення кількості коефіцієнтів) до будь якого поля(в тому числі Q R, C), над якими еліптичні криві мають доволі гарну геометричну інтерпритацію(над комплекними числами варто зазначити аналогії з комплексним тором та комплекними решітками, що приводять до ізоморфних групових структур точок еліптичних кривих).

Еліптичні криві можна поділити на *сингулярні* і не *сингулярні* відповіно до рівності чи не рівності нулю *дискримінанта кривої* $A = —16(4a^3 + 27Ь^2)$. Сингулярні криві містять самоперетин(англ *casp*), тому з точки зору алгебри та криптографії вони не відіграють важливу теоретичну роль і не мають практичного застосування. Тому, як правило, в теорії еліптичних кривих прийнято апріорі вважати $A = 0$.

Відомо, що на множині точок еліптичної кривої можна ввести груповий закон з формальною операцією додавання точок(в афінних координатах) $P = (x_1,y_1), Q = (x_2,y_2), R = (x_3,y_3) = P + Q$ наступним чином:

$(x\text{-}З,$ Ы = (A^2 — Xi — X_2, —\x_3 — Yi + Ажі)

$$_A = \begin{cases} \wedge P = Q \\ 1 \quad \wedge \quad P = Q \end{cases}$$

Обернена точка обчислюється наступним чином:

$Q = {}^{-P} = {}^{-(}xi{}^{\cdot}yi^{)} = (>_ь {}^{-}yi^{)}$

Нейтральний елемент групи точок еліптичної кривої в формі Вейєрштраса - точка в безкінечності: $O : VQ \in E: Q + O = Q$. Точка є винятковою, оскільки її немає на афінній площині над полем F_q, але її можна зобразити точкою на проективній площині.

Важливою операією для криптографічних застосунків є так званий скалярний добуток:

$$Q = [к]P = P + \bullet\bullet\bullet + P$$

$$к$$

Визначення 1.2. *Проективною площиною* P^2 над полем F_q ми називаємо множину $\{(X,Y,Z) : X,Y,Z \in F_q, (X,Y,Z) = (0,0,0)\}$ з відношенням еквівалентності:

$$(X_1,Y_1,Z_1) \sim (X2,У2,^2) \wedge ЗА \in F_q: (^2,^2, ^2) = (XX_1,XY_1,XZ_1)$$

Між проективними та афінними координатами існує наступний зв'язок:

$$_ж = x/Z, y = Y/Z$$

Заміною ми можемо привести рівняння кривої E з афінних координат в проективні:

$$E: Y^2Z = X^3 + aXZ^2 + bZ^3$$

Над проективними координатами точка в безкінечності O буде (0,1,0)

Введення проективних координат є доволі корисним з точки зору криптографії, оскільки також дозволяє ввести більш ефективні закони додавання та подвоєння точок, не використовуючи інверсії в базовому полі, оскільки кожна інверсія є доволі важкою операцією.

Еліптичні криві в загальному алгебраїчному сенсі визначаються над замиканням поля F_q і структура їх груп розглядається в найбільш широкому сенсі, тобто над полем, що містить всі можливі корені(алгебраїчно замкнуте поле).

Важливо зазначити, що структура групи точок еліптичної кривої над F_q - алгебраїчним замиканням поля F_q не є довільною, а визначається

згідно з наступної теореми:

Теорема 1.1. *Якщо m не дыиться на характеристику поля p = char(F_q), група точок порядку m має наступну структуру:*

$$E\ [m] = Z_m\ x\ Z_m$$

Якщо $m = p^2$*:*

$$E\ [m] = Z_m\ або\ O$$

Проте при розгляді еліптичної кривої з координатами із базового поля *K* структура групи *K*-раціональних точок (позначається *E(K)*) може дещо змінюватись в сторону спрощення. Саме групи *K* - раціональних точок і знайшли найширше використання в криптографії та інших практичних застосуваннях, оскільки такі групи є скінченими.

Важливою класифікацією еліптичних кривих є поділ на *суперсингулярт* та *ординарт* криві.

Визначення 1.3. Еліптична крива *E* над полем F_q, $q = p^m$ називається суперсингулярною, якщо $E[p^\kappa] = O$ або ординарною в протилежному випадку, коли $E\ [p^\kappa] = Z_p k$.

В класичній криптографії найбільш широко використовуються ординарні криві майже простого порядку, а суперсингулярні криві вважаються трохи слабшими для практичного використання оскільки до них можна легше застосовувати MOV-атаки, проте саме суперсингулярні криві стануть базою для побудови постквантових протоколів, оскільки вони мають ряд властивостей, що роблять їх практично менш вразливими до атак з використанням квантового комп’ютера, оскільки кільця ендоморфізмів в них мають складнішу структуру і є, взагалі кажучи, некомутативними.

Однією з важливих задач теорії еліптичних кривих над скінченим полем є обчислення порядку групи *K*-раціональних точок кривої $N_E = \#E(K)$, оскільки в сучасній неквантовій еліптичній криптографії важливу роль відіграють еліптичні криві майже простого порядку, тобто

криві з $N_E = кр$, де $р$ - просте, а $к$ - малий кофактор. Майже простий порядок кривої дозволяє спертись на складність задачі дискретного логарифмування в циклічній групі точок великого порядку еліптичної кривої при побудові криптографічних протоколів, оскільки найкращі алгоритми для дискретного логарифмування(КЬс-метод Поларда) дають складність порядку $0(\sqrt{N_E})$.

Теорема 1.2. *Границя Гассе. Порядок групи точок еліптичної кривої* $N_E = |E|$ *над полем* F_q *задовольняє наступну нерівність:*

$$| N_E - (q + 1) | < 2\sqrt{q}$$

Величину $t = N - (q + 1)$ також називають *слідом Фробетуса* кривої E. Теорема Гассе наводить допустимі границі даного параметра. Слід Фробеніуса однозначно визначає порядок кривої над полем. Таким чином задача пошуку даного параметра і є ключовою при знаходженні порядку кривої.

В 1985 році голандський математик Рене Скуф навів поліноміальний алгоритм для знаходження сліду Фробеніуса, таким чином зробивши більшість сучасних криптографічних застосувань еліптичних кривих доступними, хоча і складність оригінального алгоритму складала
$0 (log^5 q)$ при умові використання ефективної арифметики. Пізніше зявилися ефективніші модифікації алгоритму, зокрема модифікація Елкінса та Еткіна - алгоритм *SEA* з складністю $0(log^4 q)$. Аналогічний алгоритм пошуку порядку для поля малої характеристики відомий як алгоритм Сато.

В нашій роботі ми найчастіше матимемо справу з суперсингулярними кривими, тому дамо чітку характеризацію їх порядків.

Теорема 1.3. *Нехай Е - суперсингулярна крива над полем* F_q *характеристики р. Тоді для порядку групи* F_q *раціональних точок* $N = \#E(F_q)$ *справедливо*

$$N = 1 \bmod p$$

Зокрема наведемо деякі приклади можливих порядків: у випадку коли $q = p$: $N = p + l, t = 0$

Якщо $q = p^2$: $N = (p + l)^2, t = 2\sqrt{q}$

По аналогії з іншими алгебраїчними об'єктами над еліптичними кривими також можна ввести поняття *ІЗОМОРФІЗМУ*.

Визначення 1.4. Еліптичні криві $E\backslash, E_2$ називаються ізоморфними над F_q, якщо існує деяке раціональне перетворення координат $'ф$ над F_q, що дозволяє перетворити рівняння кривої $E\backslash$ до рівняння E_2, а також ■$Ф (Oi) = 02$

Якщо криві $E\backslash$, E_2 ізоморфні - натуральним чином індукується ізоморфізм їх груп над F_q, проте ізоморфізм груп F_q - раціональних точок не гарантований. Випадок, коли криві ізоморфні над F_q, але їх групи F_q - раціональних точок називається *квадратичним зсувом*.

Характеризацію класів ізоморфізмів еліптичних кривих дає поняття *j-інваріанта*.

$$j\ (E_{a,b}) = 1728\text{-}\frac{An^3}{4a^3 + 27b^2}$$

Теорема 1.4. *Елттичт КРШІ E\ та E_2 ІЗОМОРФНІ над F_q тоді і тыьки тоді, коли $j(E\backslash) = j(E_2)$*

Таким чином ізоморфні криві будуть мати такий самий j-інваріант.

1.2.1 Криві в формі Монтгомєрі та Едвардса

Введене вище визначення еліптичної кривої в формі Вейєрштраса є не єдиною можливим визначенням алгебраїчної еліптичної кривої. Інші широко розвсюджені в криптографії форми еліптичних кривих є форма *Монтгомері* та форма *Едвардса*.

Визначення 1.5. *Еліптичною кривою в формі Монтгомері МА^В* над

полем F_a ми називаємо множину точок із F_a , що задовольняють рівнянню:

$$M_{A,B} : By^2 = x^3 + Ax^2 + x; A, B \in F_q$$

Можна показати, що кожна крива в формі Монтгомері зводиться шляхом заміни координат та алгебраїчних перетворень до аналогічної кривої Вейєрштраса $M_{A,B} \wedge E_{a,b}$, проте зворотне перетворення має місце не завжди, основною умовою існування такого перетворення є подільність порядку кривої на чотири. Еліптичні криві в формі Монтгомері стають особливо в пригоді при побудові швидких еліптичних криптосистем з мінімальними вимогами до пам'яті(як то на мікроконтролерах з обмеженим обсягом оперативної пам'яті), оскільки формули для додавання та дублювання точок в проективних координатах не використовують координату Y, а для її відновлення наприкінці кожного скалярного добутку використовують метод різниці точок, що дозволяє обрати правильну точку із двох можливих.

Еліптичні криві в формі Едвардса - поруч із кривими Монтгомері є дуже зручними в криптографії, оскільки симетричні групові закони та доволі невелика кількість операцій при додаванні точок роблять їх дуже привабливими в побудові сучасних криптосистем. В ході роботи ми будемо переважний час працювати з кривими Едвардса, тому опишемо їх трохи глибше.

Визначення 1.6. Еліптичною кривою в формі Едвардса(з модифікацією Бернштейна-Ланге) Ed над полем F_q ми називаємо

• ~TT~2 •

множину точок із r_q , що задовольняють рівнянню:

$$M_{A,B} : x^2 + y^2 = 1 + dx^2y^2; d \in F_q$$

Зазначимо, що оригінальна форма кривих Едвардса була визначена дещо по іншому, проте в роботах Бернштейна вона була дещо уточнена і спрощена, а повна і чітка класифікація кривих Едвардса була дана в роботі А.В. Бессалова [21]. Далі ми будемо використовувати термінологію

і класифікацію із цієї роботи. Опишемо закон додавання точок:

$$/ \quad \backslash \ . \ / \quad \backslash \quad / X_1Y_2 + X_2Y_1 \qquad Y_1Y_2 \ \overline{} \ X_1X_2 \ ,$$
$$X\backslash, Y\backslash \qquad X_2' Y2 \qquad 1 + (i_X 1_X 2 Y1 Y2' \ 1 \ .4X — X2 Y1 Y2$$

Як бачимо, закон є доволі симетричним і зручним, особлива формула для дублювання точки відсутня, тому може бути отримана простою підстановкою $x_1 = x_2$, $y_1 = y_2$. Точка на безкінечності відсутня в формі Едвардса, а роль нейтрального елемента групи відіграє точка кривої $OE_Л = (0,1)$. Обернена точка також доволі просто може бути отримана із групового закону: $—(x_1,y_1) = (—x_1,y_1)$. Також крива завжди містить принаймні одну точку порядку 2 - $D = (0, —1)$, а також принаймні дві точки порядку 4 - $F_{1i2} = (±1,0)$.

Криві Едвардса(в сенсі F^-раціональних точок) класифікуються згідно квадратичності параметра d. Так, якщо параметр d є квадратичним нелишком в полі F_q, то закон додавання точок буде повним(тобто буде діяти для будь яких точок кривої) і відповідна крива називається *повною кривою Едвардса*. Зазначимо, що для повних кривих Едвардса групова структура є досить чіткою і визначені вище точки порядку 2 і 4 є єдиними такими точками відповідних порядків.

Варто зазначити, що існування точок порядку 4 накладає умову подільності порядку групи F^-раціональних точок кривої на 4, а це приводить до висновку, що подібно до форми Мотгомері - не кожна еліптична крива в формі Вейєрштраса може бути зведена до еліптичної кривої в формі Едвардса, проте обернене твердження, звісно ж, правдиве.

Значно цікавіша і не тривіальна ситуація виникає, коли параметр d є квадратичним лишком в полі F_q. Такі криві називають *квадратичними кривими Едвардса*. В цьому випадку можна показати, що закон додавання точок втрачає свою повноту, оскільки знаменники $1 \pm dx_1x_2y_1y_2$ можуть перетворюватись на 0, таким чином роблячи результат невизначеним, але все-ж можна довизначити такі особливі випадки і знайти точки, що "псують"закон додавання. Такими точками

виявляються нові точки порядку 2 та 4. Дві особливі точки порядку 2 можна формально визначити як - $D_{12} = (ж, \pm л/d^{-1})$. Формально нові точки порядку 4 - $F_{2,3} = (\pm л/d^{-1}, ж)$. Також крім двох стандартних точок порядку 4, двох особливих квадратична крива може містити ще 2 звичайні точки порядку 4 при особливих умовах.

В нашій роботі ми розглянемо побудову постквантового протоколу обміну повідомленнями на основі супесингулярних еліптичних кривих в формі Едвардса.

Висновки до розділу 1

В розділі зроблено огляд проблем сучасної криптографії, зокрема розлянуто можливості квантового комп'ютера до розв'язання задачі факторизації. Також зроблено огляд основного математичного апарату для побудови алгоритму постквантового ключового обміну SIDH, а також деякий огляд еліптичних кривих в формі Едвардса.

2 ПОБУДОВА ПОСТКВАНТОВОЇ СИСТЕМИ ОБМШУ ПОВІДОМЛЕННЯМИ НА ОСНОВХ ХЗОГЕНХЙ ЕЛІПТИЧНИХ КРИВИХ

В розділі ми розглянемо протокол SIDH та можливості його реалізації з використанням кривих Едвардса.

2.1 Хзогенії еліптичних кривих

Одним із основних понять посквантової криптографії на ізогеніях суперсингулярних еліптичних кривих є поняття *ізогенії* - спеціального відображення однієї кривої в іншу, що має ряд корисних властивостей. Наведемо алгебраїчне визначення ізогенії, проте варто зазначити що поняття ізогенії має також інтерпретацію в сенсі алгебраїчної геометрії, яку ми наводити не будемо.

Визначення 2.1. Ізогенією між еліптичними кривими $E\backslash$ та E_2 над скінченим полем F_q називають сюр'єктивний гомоморфізм їх груп, що має скінчене ядро та може бути виражений раціональними функціями:

$$\phi: Ei \wedge E2;\ \Phi^{(}{}_{X,y)} = \text{(g)}$$

Де a_x, a_y, b_x, b_y - деякі поліноми. Порядком ізогенії називають число $deg(\phi) = I = max(deg(a_x), deg(b_x))$.

Важливим класом ізогеній є *сепарабельні ізогенії,* для яких deg(0) = *#кегф*. В нашій роботі ми розглядатимемо тільки сепарабельні ізогенії, тобто такі, порядок яких співпадає з порядком ядра групового гомоморфізму, що індукує ізогенія.

Одним із класичних результатів теорії є теорема Сато-Тате.

Теорема 2.1. *(Сато-Тате); Між еліптичними кривими E_1 та E_2 існує ізогенія над полем F_q тоді і тільки тоді коли $\#E_1(F_q) = \#E_2(F_q)$.*

Також важливим результатом є теорема про думальну ізогенію:

Теорема 2.2. *Якщо ф: E_1 ^ E_2; <ієд(ф) = I над полем F_q, то також існує дуальна ізогенія ф: E_2 ^ E_1 кривої E_2 в криву E_1 над полем F_q, така що <ієд(ф) = I і*

$$\Phi^{o}\Phi = [I]_{E_1},\ \Phi^{o} = [I]_{E2}$$

Де $[I]_{E_1}$ - ізогенія кривої E_1 в криву E_{1}} що представляє собою звичайний скалярний добуток на число I на кривій E-_.

Зазначимо, що при композиції ізогеній їх порядки перемножаються, тому порядок ізогенії скалярного добутку $deg([I]_E) = I^2$. Цей факт в дечому може бути використаний для доведення теореми про квадратичність підгруп групи еліптичної кривої, яка була описана в попередньому розділі. Також даний факт допомагає будувати ізогенії великого гладкого порядку на основі ізогеній малого порядку, до прикладу ізогенія порядку I^m може бути побудована як композиція m ізогеній порядку I. Також важливим прикладом ізогенії є звичайний ізоморфізм еліптичних кривих. Ізоморфізм - це ізогенія порядку 1, оскільки єдиним елементом в ядрі індукованого гоморфізму є точка на безкінечності.

Таким чином ми можемо ввести на множині всіх еліптичних кривих над полем F_q відношення еквівалентності, яке назвемо *ізогенністю,* так криві E\^ та E_2 - ізогенні, якщо між ними існує ізогенія(в тому числі дуальна). Відповідно, всі еліптичні криві над полем можна поділити на класи ізогенних кривих. Важливою характеристикою кожного класу можна назвати кардинальність F^-раціональних груп точок його представників, згідно теореми Сато-Тате вона має бути однаковою у всіх представників класу. Певний клас ізогенних кривих можна зобразити у

вигляді неорієнтованого мультиграфу, вершинами якого будуть класи ізоморфних еліптичних кривих(або що еквівалентно значень їх j-інваріантів), а ребрами - ізогенії між ними. Представлення класу ізогенних кривих як графу ізогеній певного малого порядку дуже важливе для криптографії, оскільки воно відразу індукує проблему пошуку шляху між двома вершинами, або що еквівалентно, пошуку ядра, який індукує ізогенію гладкого порядку між двома кривими, проблему яка і ляже в основу постквантового криптографічного алгоритму генерації спільного секрету - *SIDH*.

Важливим класом ізогенних кривих є суперсингулярні криві однакового порядку, вони мають, як вже було згадано, доволі масивні кільця ендоморфізмів, що робить їх більш привабливими з точки зору криптографії. Також дуже важливо, що для суперсингулярних кривих, на відміну від ординарних - граф і-ізогеній(тобто ізогеній, що мають порядок і) є (і+1)-регулярним, тобто кожна вершина в графі має (1+1) суміжне ребро(а це максимально-можлива кількість), що робить аналіз алгоритмів на графах ізогеній більш легшим в теоретичному плані та значно ускаднює життя квантовим криптоаналітикам.

При побудові ізогенії ϕ кривої E_1 в криву E_2 ключову і визначальну роль грає ядро $G = кег(\phi) = \phi^{-1}(О_{в_2})$, саме воно і визначає ізогенію і дозволяє побудувати поліноміальний відносно розміру ядра алгоритм знаходження кривої E_2(взагалі кажучи, ми можемо потрапити в будь яку криву ізоморфну E_2, тому доцільніше говорити про знаходження j-інваріанту кривої-образу) та образу точок кривої E\^ при відображенні ϕ на E_2: $\Phi(Pei) = Pe_2$. За аналогією яку нам дає теорема про гомоморфізми груп ми можемо сформулювати аналогічне твердження щодо ізогеній:

Теорема 2.3. *Якщо* $\phi: E_1 \wedge E_2$ *- ізогенія кривих* E_1, E_2 *над полем* F_q, *тоді*

$$E_1[F_q]/ < кег(\phi) >= E_2[F_q]$$

Саме цей ізоморфізм, дозволяє шукати нам групу кривої-образу, як

деяку фактор-групу по ядру ізогенії - групі $G = кег(ф)\ C\ E\backslash[F_q]$. Подібний алгоритм для знаходження кривої образу та образів точок при ізогенії дав *Велу,* чиїм ім'ям названі відповідні формули, які на вхід приймають коефіцієнти кривої $E\backslash$, координати точок із групи ядра G, деяку точку $P \in E\backslash, P \in G$ і на виході дають відповідні коефіцієнти кривої-образу E_2 та образ вибраної точки $P_{E_2} = ф(P)$.

2.2 Алгоритм SXDH

З появою ідей створення квантового компютера, дослідники почали активно вести пошук кандидата на роль алгоритму, що залишився би стійким в моделі, коли криптоаналітик має в розпорядженні практичний працюючий квантовий комп'ютер. Відомо, що квантовий компютер здатний розвязувати задачу пошуку прихованої підгрупи за поліноміальну кількість квантових вентилів(задача пошуку прихованої підгрупи належить до класу складності *BQP*). Зокрема, задачами, в які можна звести задачу про знаходження прихованої підгрупи є задача факторизації в цілих числах, задача дискретного логарифмування в мультиплікативній групі поля та задача дискретного логарифмування в групі F_q - раціональних точок еліптичної кривої, таким чином роблячи дані задачі вразливими до квантового компютера.

Ідея розробки постквантового протоколу, що використовував би ізогнеії еліптичних кривих завдячує роботі Столбунова та Ростовцева 2006р., що запропонували вперше використати ізогенії еліптичних кривих для побудови посквантового протоколу ключового обміну[18]. Ця робота використовувала звичайні ординарні еліптичні криві, пізніші дослідження скомпроментували використання ординарних кривих, зокрема в роботі [19] було наведено субекспоненційні алгоритми для квантового

комп'ютера, що знаходять ядро ізогенії за субекспоненційний час, також протокол ключового обміну роботи Столбунова є надзвичайно не практичним, оскільки ключовий обмін займає час порядку декількох хвилин. Хоча робота і виявилась не практичною, саме її ідеї і стали підгрунтям для створення більш практичного і стійкого протоколу *SIDH (Supersingular Isogeny Diffie-Hellman).* SIDH був запропонований групою дослідників в роботі [14] та на відміну від старіших протоклів є набагато практичнішим та швидшим, оскільки допускає використання декількох технік компресії ключів, що суттєво зменшують довжини посилок в протоколі, також протокол використовує швидку арифметику в проективних координатах на кривих в формі Монтгомері, що також обумовлює доволі невеликий час ключового обміну. Опишемо алгоритм детальніше.

- Генерація параметрів. В якості параметрів алгоритму SIDH необхідно вибрати просте число виду $p = l^{C}д\ l^{C}д$ — 1, де *I а, їв* - малі прості числа, початкову суперсингулярну криву, до прикладу - E_0*:* $y^2 = x^3 + x$ над полем F_{p^2}. Зазначимо, що друга степінь розширення поля вибрана не випадково, оскільки можна довести, що всі суперсингулярні криві над полем F_p мають j-інваріант в полі F_{p^2} [20], таким чином всі суперсингулярні криві над скінченим полем можуть бути визначені над F_{p^2} і кількість кривих з точністю до ізоморфізму є скінченою, а також граф /-ізогеній, як вже було згадано, є *I* + 1 регулярним. Важливо зазначити, що порядок кривої E_0*(i* будь якої ізогенної до неї) буде рівним $(I^e/ГВ)^2$. Таким чином група *E^A* = $E_0[I^e/]$ = *0* Z*fA* буде F_{p^2}-раціональною і буде містити $I^{e}\wedge^{-1}$ *(Ia* + 1) циклічних підгруп порядку $i^c a$, кожна з яких може бути ядром певної ізогенії, таким чином ми можемо оцінити кількість можливих I^eA-ізогеній, дана оцінка є важливою при оцінці стійкості алгоритму до криптоаналізу. Таку ж саму оцінку можна надати для структури групи = $E_0[ГВ]$. Для груп $E^A,E0^B$ необхідно вибрати генератори, тобто пари точок *(Pa,Qa), (Рв,Qb*), такі що точки *P,Q* в кожній парі будуть незалежними, тобто одна з них не

може бути виражена як скалярний добуток іншої на певне число, таким чином кожен елемент групи E_Q може бути представлений як лінійна комбінація точок-генераторів $Pд,\ Q_A$, те ж саме справедливо і для групи $E^\wedge$ з точками $Pв,\ Q_B$.

- Генеращя КЛЮЧІВ. Традиційно дві сторони ключового обміну будемо називати Алісою і Бобом. Аліса генерує свій закритий ключ, як деяку циклічну підгрупу порядку $1^{C}д$ групи E_Q. Для цього вона вибирає деякі числа $ПА,ША : 1 < ПА,ША < I^{C}A$, які не діляться на $I\ A$ одночасно та обчислює точку $R_A = [ПА]P_A + [mA]Q_A$, яка і буде генератором секретної підгрупи (R_A) Аліси. Боб робить з свого боку таку-ж процедуру, вибравши секретні числа $пв\ ,шв$, обчислюючи свою закриту підгрупу як $(R_B) = ([ПВ\}P_B + [\wedge_B]Q_B)$. Циклічні секретні підгрупи $(R_A), (R_B)$ можна розглядати як ядра деяких ізогеній кривої E_0 порядків $1^{C}д\ ,Г\%$ відповідно, для зручності ці секретні ізогенії позначимо як $Ф_A, Ф_B$ Таким чином Аліса обчислює криву $E_A = E_O/(R_A)$ та образи базових точок групи Боба, відкритий ключ $РК_A = (E_A, Ф_A(P_B), Ф_A(Я_B))$, відкритий ключ Боба

аналогічно задається кривою $Ев = E_Q/(R_B)$ та образами точок групи Аліси - $РКв = (Ев, фв(P_A), Ф_B(Q_A))$. Сторони обмінюються своїми відкритими ключами.

- Генеращя СПІЛЬНОГО секрету. Після обміну своїми відкритими ключами сторони обчислюють спільний секрет. Аліса обчислює ізогенію

$$Ев/(Ф_B(P_A)ПА + Фв(Q_A)\wedge A) = E_Q/(R_A, R_B)$$

Боб аналогічно обчислює

$$E_A/(Ф_A(P_B)пв + Ф_A\wedge_B)ШВ) = E_Q/(R_b, R_A)$$

В результаті ключвого обміну сторони отримали спільну еліптичну криву, ізоморфну $E_Q/(R_B\ ,\ R_A)$, взагалі кажучи, в ході ключового обміну коефіцієнти спільної кривої Боба та Аліси можуть відрізнятись, проте отримані криві обов’язково будуть ізоморфними, а отже будуть мати

одинаковий j-інваріант, який і можна прийняти за спільний секрет. Таким чином значення спільного секрету - $j(E_0/(R_B, R_A))$

Постквантова стійкість даного алгоритму базується на складності задачі знаходження ядра ізогенії, при відомих початковій та кінцевій еліптичних кривих(так звана проблема CSSI(Computatiomal Supersrngular Isogeny problem)). Маючи криві E_0 та $E_0/(G)$ знайти секретні числа $n,m,$ що генерують групу $G = ([n]P + [m]Q)$. Зазначимо, що задача знаходження генераторів групи при відомій точці із цієї групи є доволі легкою для суперсингулярних кривих, що використовуються в алгоритмі, оскільки порядок групи є гладким числом, а сама задача знаходження n,m є узагальненням задачі дискретного логарифмування на еліптичній кривій, що має поліноміальну складність при умові гладкості порядку. Основну складність алгоритму формує складна задача знаходження шляху на графі ізогеній, найкращий алгоритм пошуку шляху на графі ізогеній є алгоритм *meet-in-the-middle,* що є дещо аналогічним до однойменного алгоритму пошуку колізії геш-функції. Складність цього підходу в класичній моделі обчислень можна оцінити як $0(p^{1/4})$, в квантовій моделі даний алгоритм є трохи ефективнішим, проте все-ж з експоненційним числом квантових вентилів - $0(p^{1/6})$, саме на цьому факті і грунтується припущення про постквантову стійкість алгоритму SIDH, оскільки поки ефективніші алгоритми для знаходження ядра ізогенії не знайдені. До прикладу, одним з розповсюджених модулів є $p = 2^{387}3^{242} — 1$, що забезпечує постквантову стійкість на рівні 128 кубіт, а класичну на рівні 192 біт. Зазначимо, що при виборі модуля, необхідно вибирати числа $1^{є}д \sim 1^{є}д$, тобто максимально близькими один до одного щоб стійкість алгоритму була приблизно однаковою для всіх сторін(Аліси і Боба).

Алгоритм SIDH ліг в основу постквантового алгоритму інкапсуляції ключа SIKE, що був завпропонований в 2017 році як офіційний кандидат на роль постквантового алгоритму інкапсуляції ключа в конкурсі, що зараз проводить NIST. Алгоритм повноцінно може конкурувати з

алгоритмами на решітках та корегуючих кодах, проте через відносну складність опису основних елементів алгоритму він все ще інтенсивно вивчається як алгебраїстами, так і криптографами і є не таким популярним, як алгоритми на решітках. В загальному вигляді конструкція SIDH не може бути застосована до схеми Ель-Гамаля цифрового підпису, оскільки відсутній груповий закон на відкритих даних, що є необхідною умовою застосування схеми Ель-Гамаля, проте на основі протоколу з нульовим розголошенням, що базований на знанні секретного ядра ізогенії все-ж можна побудувати схему електронно-цифрового підпису(або ж автентифікації).

Висновки до РОЗДІЛУ 2

В розділі було розглянуто алгоритм постквантового ключового обміну SIDH, сферу його застосування та основи його стійкості до криптоаналізу з використанням квантових комп’ютерів.

3 ЗАСТОСУВАННЯ КРИВИХ В ФОРМ! ЕДВАРДСА ДО АЛГОРИТМУ SIDH

В розділі ми проаналізуємо можливість реалізації протоколу SIDH з застосуванням кривих Едвардса та запропонуємо реалізацію деяких основних елементів протоколу з використанням кривих Едвардса.

3.1 Практична реалізація алгоритму SIDH та виявлені проблеми

Ідея реалізації протоколу SIDH на кривих Едвардса відносно нова, зокрема в роботі [16] наведено алгоритми обчислення 3 та 4-ізогеній на кривих в формі Едвардса, а в роботі [13] зроблено спробу гібридної реалізації протоколу SIDH з використанням арифметики кривих Едвардса і ізогеній кривих в формі Монтгомері.

В оригінальній роботі автори реалізації протоколу SIDH використали криві в формі Монтгомері та швидку арифметику в проективних координатах $(X : Z)$ без використання координати Y. Оригінальна реалізація дослідників із Microsoft доступна за посиланням https://github.com/Microsoft/PQCrypto-SIDH.

В нашому дослідженні ми проаналізували існуючі роботи і реалізували складові частини алгоритму SIDH з використанням кривих Едвардса, зокрема виконана реалізація довгої арифметики в полі F_{p2} з можливістю розширення розміру поля, в тому числі реалізовано ефективні алгоритми для квадратного кореня та інверсій в полі F_{p^2}, а також здійснено реалізацію арифметики на кривих Едвардса в афінних та проективних координатах, при цьому реалізація в частині арифметики

може бути використані в практичних криптографічних застосуваннях, оскільки алгоритм скалярного добутку працює константний час, що нівелює частину атак з використанням побічних каналів.

Також в нашій роботі було запропоновано алгоритм для генерації точок-генераторів груп $E_0[\Gamma\hat{}]$ на кривих Едвардса. По аналогії з алгоритмом для знаходження аналогічних точок-генераторів в оригінальній реалізації Microsoft [15] ми використали формулу для розсіюючого перетворення(^^геіоп map), застосувавши перетворення координат із форми Монтгомері в форму Едвардса та отримали наступну формулу для знаходження точки Q, незалежної від знайденої раніше точки $P = (x,y)$ з ^-раціональними координатами:

$$Q = \phi(P) = (ix, y^{-1})$$

Також, проаналізувавши роботу [16] було виконано реалізацію алгоритмів для обчислення ізогеній порядків 3 та 4. В проаналізованій роботи автори не наводили посилань на практичну реалізацію отриманих ними алгоритмів для 3,4-ізогеній над кривими Едвардса. Нами було помічено деякі проблеми в практичній реалізації алгоритму SIDH з використанням еліптичних кривих в формі Едвардса, зокрема наявність особливих точок 2-го та 4-го порядків ніяк не згадана в роботі [16], проте дана проблема виникає при спробі реалізувати повноформатний протокол SIDH.

При переході від форми Монтгомері до форми Едвардса початкова суперсингулярна крива M_0: $y^2 = x^3 + x$ перетворюється в біраціонально-еквівалентну криву E_0: $y^2 + x^2 = 1 - x^2y^2$. Як бачимо значення параметра $d = -1$, що є квадратичним нелишком в полі F_p стає квадратичним лишком в полі F_{p^2}, оскільки будь який елемент поля F_p є квадратичним лишком в полі F_{p^2}, таким чином крива E є квадратичною над полем F_{p^2}.

Факт наявності особливих точок будь-якої суперсингулярної кривої в формі Едвардса засвідчує порядок групи F_{p^2}-раціональних точок - $4^{\epsilon A}3^{\text{ев}}$,

що використовується в оригінальній реалізації та теорема про структуру графа l-ізогеній суперсингулярних кривих, що стверджує наявність 3-х можливих ізогеній порядку 2, що еквівалентно наявності 3-х точок порядку 2. Зазначимо, що при практичній реалізації наявність особливих точок значно ускладнює реалізацію та робить алгоритми менш-практичними, оскільки наявність будь-яких виняткових випадків в арифметиці на еліптичних кривих виливається в вразливості до атак за побічними каналами, а також зменшує швидкість алгоритмів за рахунок наявності умовних непередбачуваних переходів, що досить "важко"сприймаються процесорами, хоча саме від цього недоліку і захищались впровадженням арифметики на кривих в формі Едвардса, проте цього крива має бути повною. Замість однієї точки на безкінечності на кривій в формі Монтгомері при аналогічній реалізації протоколу з використанням кривих Едвардса необхідно враховувати 2 додаткові точки на безкінечності порядку 2, а також 2 точки на безкінечності порядку 4. Отримані результати, на жаль, не показують що алгоритм SIDH в оригінальному вигляді можна реалізувати з використанням кривих Едвардса.

Наша реалізація елементів протоколу SIDH з кривими Едвардса описана в додатку [А.1], а також доступна за посиланням https://github.com/juja256/Isogenies

3.2 Напрямки подальших досліджень

Оскільки при реалізації класичного протоколу SIDH з використанням еліптичних кривих в формі Едвардса виникли певні пробелми - можна запропонувати шляхи для їх подолання. На даному етапі можна запропонувати замість використання ізогеній порядку 3 і 4

використовувати ізогенії порядку 3 і 5, вибираючи p наприклад у вигляді $p = 4/3^{CA}5^{ев} - 1$ - цей підхід в дечому вирішить проблеми особливих точок, оскільки ядра для ізогеній будуть вибиратись із груп $E[3^{eA}]$, $E[5^{ев}]$, в яких особливі точки відсутні, але потребує також нових формул для обчислення 5-ізогеній.

Висновки до розділу 3

В розділі було проведено аналіз застосування еліптичних кривих в формі Едвардса до алгоритму SIDH, були виявлені певні проблеми з ефективною інтеграцією кривих Едвардса в алгоритм SIDH та запропоновано шляхи їх вирішення. Також в ході дослідження була розроблена імплементація елементів алгоритму SIDH мовою C++ з використанням кривих Едвардса.

ВИСНОВКИ

На сьогоднішній день вчені продовжують спроби створення квантового комп’ютера, проте стикаються з рядом практичних перешкод, які роблять неможливим створення практичного квантового ком’пютера в найближчому майбутньому, хоча квантова механіка і не забороняє побудову такого роду системи. Проте, сподіватись на те що його ніколи не буде створено також не варто - потенційна можливість залишається і вона несе багато загроз для сучасної криптографії, тому розробка нових постквантових криптосистем - надзвичайно актуальна задача сьогодні. Одним із найперспективніших напрямків сучасної постквантової криптографії - криптосистеми на основі ізогеній суперсингулярних еліптичних кривих, зокрема криптосистема SIKE на базі алгоритму SIDH потрапила в нещодавній конкурс NIST на роль постквантового алгоритму інкапсуляції ключів та є дуже важливим об’єктом для вивчення з боку математиків та криптографів, тому тематика нашого дослідження є надзвичайно актуальною.

В роботі було зроблено огляд вразливостей сучасної асиметричної криптографії до атак з застосуванням квантового комп’ютера, зроблено огляд математичного апарату, що використовується при побудові постквантового алгоритму ключового обміну SIDH.

Також в ході нашої роботи була досліджена можливість інтеграції еліптичних кривих в формі Едвардса з алгоритмом SIDH та було виявлено ряд проблем, зокрема при класичній реалізації протоколу SIDH
з використанням 3 та 4-ізогеній виникають особливі точки порядків 2 і 4, які необхідно правильно обробляти, що значно ускладнює ефективну реалізацію алгоритму.

В роботі було запропоновано використовувати натомість ізогенії порядку 3 та 5, щоб обійти цю проблему. В ході подальших досліджень можна здійснити пошук формул для обчислення ізогеній порядку 5 та

виконати повноформатну ефективну реалізацію алгоритму SIDH з використанням еліптичних кривих в формі Едвардса. Зокрема в ході нашої роботи було здійснено програмну реалізацію деяких компонентів алгоритму SIDH з кривими в формі Едвардса, що в подальшому може значно полегшити дослідження.

ПЕРЕЛІК ПОСИЛАНЬ

1. Shor P. W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer [Text] / P. W. Shor //S IAM J. Comput. - 1997. - 26 (5). - P. 1484-1509.

2. Grover L. K. A fast quantum mechanics algorithm for database search [Text] / L. K. Grover // Proceeding of the 28th ACM Symposium on Theory of Computation, New York: ACM Press. - 1996. - P. 212-219.

3. Feyman R. P. Quantum mechanical computers [Text] / R. P. Feyman // Opt. News - 1985. - February, 11. - P. 11-39.

4. Deutsch D. Rapid Solution of problems by quantum computation [Text] / Deutsch D., Jozsa R. // Proc. R. Soc. Lond. A. - 1992. - Vol. 439 (1907). - P. 553-558.

5. Bernstein, D. Post-quantum cryptography [Text] / D. Bernstein, J. Buchmann, E. Dahmen. - Berlin: Springer, 2009. - 246 p.

6. IEEE Std 1363. 1-2008. IEEE Standard Specification for Public Key Cryptographic Tehniques Based on Hard Problems over Lattice [Text]. 2009 - 4 - 10 - NY: The Institute of Electrical and Electronics Engineers, Inc. - 2009. - 69 p.

7. Main page of PQCrypto 2014 [Electronic resource] / University of Waterloo, Ontario, Canada Сайт конференції PQCrypto 2014 Режим доступа : URL:- https://pqcrypto2014.uwaterloo.ca. - 21.08.2014 р .

8. Stefan Heyse. Post quantum cryptography: implementing alternative public key schemes on embedded devices: dissertation for the degree of doktor-ingenieur: 10.2013 / Stefan Heyse. - Bochum, 2013. - 235 p. - Bibliogr.: P. 205-223.

9. Ю. І. Горбенко, Р. С. Ганзя. Аналіз розвитку криптографії після появи квантових комп’ютерів.

10. Алгоритм Шора - Вікіпедія, URL - https://goo.gl/Q5vqJ1

11. Luca De Feo: Mathematics of Isogeny Based Cryptography // Ecole

mathematique africaine - 2017

12. Jean-Marc Couvëignës: Hard Homogeneous Spaces - 2006

13. Michael Meyer, Steffen Reith, Fabio Campos: On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic - 2017

14. LUCA DE FEO, DAVID JAO, AND JERoME PLUT: TOWARDS QUANTUM-RESISTANT CRYPTOSYSTEMS FROM SUPERSINGULAR ELLIPTIC CURVE ISOGENIES - 2011

15. Craig Costello, Patrick Longa, and Michael Naehrig: Efficient algorithms for supersingular isogeny Diffie-Hellman // Microsoft Research - 2016

16. Suhri Kim, Kisoon Yoon, Jihoon Kwon, Seokhie Hong, Young-Ho Park: Efficient Isogeny Computations on Twisted Edwards Curves // Hindawi Security and Communication Networks Volume - 2018

17. Luca De Feo: Відповідь на запитання, електронний ресурс crypto.stackexchange.com; Режим доступу: https://crypto.stackexchange.com/questions/59376/why-do-we-need-to- use-supersingular-curve-on-sidh

18. Alexander Rostovtsev and Anton Stolbunov. Public-key cryptosystem based on isogenies, 2006. http://eprint.iacr.org/2006/145/

19. Andrew Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time, 2010. http://arxiv.org/abs/1012.4019/

20. Joseph H. Silverman. The arithmetic of elliptic curves, volume 106 of Graduate Texts in Mathematics. Springer-Verlag, New York, 1992. Corrected reprint of the 1986 original.

21. Бессалов А.В. ЭЛЛИПТИЧЕСКИЕ КРИВЫЕ В ФОРМЕ ЭДВАРДСА И КРИПТОГРАФИЯ, Киев, "Політехніка 2017

ДОДАТОК А ТЕКСТИ ПРОГРАМ

В додатку наведено програмну реалізацію деяких компонентів.

A.1 Елементи програмної реалізації протоколу SIDH з кривими Едвардса

Лістинг файлу ’gf.h’

```
#ifndef GF_H #define
GF_H

#define BASE_FIELD_LEN (751/64 + 1 + 1) //13 64bit words
#define FIELD.EXTENSION 2

#define ARCH 64
#define BYTES_IN_WORD (ARCH/8)

#define UNSUPPORTED_PARAM -1

#define INVALID_DATA -2

#define DEBUG 1

#include <string>


typedef  unsigned  long  l o ng u6
                          4
typedef  unsigned
                   n o n
typedef  unsigned  char  u8 ;


typedef u64 BaseEl[BASE_FIELD_LEN];
typedef u64 BigInt [2*BASE_FIELD_LEN] ;

typedef BaseEl GFElement[FIELD_EXTENSION];


class GaloisFieldException { int
code; public :
      GaloisFieldException(int c);
```

```
    std: :string What () ;
};

class GaloisField {
    int bitSize; int
    wordSize;
    BigInt characteristic; int
    extension;

    BigInt size ;
    BigInt halfSize; public:



    GaloisField();
    GaloisField(const BigInt characteristic, int ext, int bitSize);
    const BigInt* GetChar();
    const BigInt* GetSize();
    int GetWordSize();
    int GetBit Size () ;
    int GetExtension();

    static  const  GFElement  Zero;
    static  const  GFElement  Unity;
    static const GFElement I;

    static void InitFromString(BaseEl a, const char* str); static void
    InitFromString(GFElement a, const char* str); std::string Dump(const
    GFElement a); std::string Dump(const BaseEl a);

    bool IsQuadraticResidue(const GFElement a); bool
    IsQuadraticResidueBase(const BaseEl a);









    vo    Copy(GFElement a, c    o   GFElement b) ;
    id                           ns
    bo o  Equal(const GFEleme    nt  , const GFEleme   nt b) ;
    l                            a
    vo    Add(const   GFElement  a,  const GFElement   b, GFElement    c) ;
    id
    vo    S ub ( c o  GFElement  a,  const GFElement   b, GFElement    c) ;
    id    ns t
    vo    N e g ( c   GFElement  a,  GFElement c) ;
    id    o ns t
    vo    P o w ( c   GFElement  a,  const BigInt n,    int nlen, GFElement
    id    o ns t
    vo    P o w ( c   GFElement  a,  const BigInt n,    GFElement b)    ;
    id    o ns t
    vo    I nv ( c o  GFElement  a,  GFElement b) ;
    id    ns t
      3 o id GFDiv(c  onst GFEle   men t  a, const GFElement b, GFElement c)
    vo    M ul ( c o  GFElement  a,  const GFElement    b, GFElement    c) ;
    id    ns t
    vo    S qr ( c o  GFElement  a,  GFElement c) ;
    id    ns t
```

```
    bool Sqrt(const GFElement a, GFElement r);
    void MulBy2Power(const GFElement a, int pp, GFElement b); void
    MulByBase(const GFElement a, const BaseEl e, GFElement c);

    /* base field operations */
    void BaseCopy(BaseEl a, const BaseEl b);
    void BaseSqr(const BaseEl a, BaseEl c);
    void BaseMul(const      BaseEl a,const   BaseEl  b, BaseEl              c);
    void BaseAdd(const      BaseEl a,const   BaseEl  b, BaseEl             c) ;
    void BaseSub(const      BaseEl a,const   BaseEl  b, BaseEl              c);
    void BasePow(const      BaseEl a,const   BigInt  n, int nlen, BaseEl b);
    void BasePow(const      BaseEl a,const   BigInt  n, BaseEl              b);
    void BaseInv(const BaseEl a, BaseEl b);
    void BaseNeg(const BaseEl a, BaseEl c);
    void BaseSqrt(const BaseEl a, BaseEl b);
    int BaseCmp(const BaseEl a, const BaseEl b);

};

/* Common Arithmetics */

void shr(u64    n,const  u64*    a,u64*     res, u64    bits);
void shl(u64    n,const  u64*    a,u64*     res, u64    bits);
u64 get_bit(const u64* a, u64 num); void copy(u64* a, const
u64* b, int len); u64 add(u64 n, const u64* a, const u64* b,
u64* c); u64 sub(u64 n, const u64* a, const u64* b, u64* c);
void _mul_raw(u64 a, u64 b, u64* low, u64* high); u64
_add_raw(u64 a, u64 b, u64* c);
void mul_by_word(u64 n, const     u64*a, u64 d, u64* c);
void mul(u64    n,const  u64*    a,const u64* b,       u64* c);
void imul(u64 n, const u64* a, const u64* b, u64* c); void
sqr(u64 n, const u64* a, u64* res); int word_bit_len(u64 w);
int bigint_bit_len(u64 n, const u64* a);
u64 sub_word(u64 n, const u64* a, const u64 b, u64* c); u64
add_word(u64 n, const u64* a, const u64 b, u64* c);

void divide(u64 n, const u64* a, const u64* b, u64* quotient, u64* reminder); int cmp(u64
n, const u64* a, const u64* b);




void add_mod(u64     n, const  u64* a,   const  u64* b,  const  u64* m,  u64*  res);
void mul_mod(u64     n, const  u64* a,   const  u64* b,  const  u64* m,  u64*  res);
void exp_mod(u64     n, const  u64* a,   const  u64* b,  const  u64* m,  u64*  res);
```

```
//void inv_mod(u64 n, const u64* a, const u64* m, u64* res);


#endif /* GF_H */
```

Лістинг файлу ’gf.cpp’

```
# include "gf.h"
#include <stdlib.h>
#include <string.h>
# include < st dio . h>
# inc l ude < s s t re am >
#include <iomanip>
# inc l ude < io s t re am >

#define MAX_U64 OxFFFFFFFFFFFFFFFF #define
MSB_M                  0x8000000000000000
#define HEX_FORMAT "%.16llX"

static BigInt zero = { 0 };

u64 get_bit(const u64* a, u64 num) {
     return a[num/64] & ((u64)1 << (num % 64));
}

inline void copy(u64* a, const u64* b, int len) { for
(int i=0;i<len;i++) a[i] = b[i];
}

inline u64 add(u64 n, const u64* a, const u64* b, u64* c) { u64
     msb_a, msb_b, carry = 0;

     for (u32 i=0;i < n; i++) { msb_a = a[i] & MSB_M;
          msb_b = b[i] & MSB_M; c [i] = a[i] + b[i] +
          carry;
          carry = ( (msb_a && msb_b) || ((msb_a ^ msb_b) && !(MSB_M & c[i]))            );
     }
     return carry;
}

inline u64 sub(u64 n, const u64* a, const u64* b, u64* c) { u64
borrow = 0; for (int i=0; i<n; i++) { u64 t_a = a[i ] ;
```

```
          u64 t_b = b[i]; c [i] = t_a

          - t_b - borrow;

          borrow = (    (~t_a)  &  (c[i]  |  t_b)  |  (c [i] & t_b) )   >>  (63) ;
     }

     return borrow;

}


inline void _mul_raw(u64 a, u64 b, u64* low, u64* high) {

#ifdef _WIN64

     *low = _umul128(a, b, high);

#else

     __int128 r = (__int128)a * (__int128)b;

     *low = (unsigned long long) r;

     *high = r >> 64;
#endif // _WIN64 }


inline u64 _add_raw(u64 a, u64 b, u64* c) {

#ifdef _WIN64

     u64 msb_a, msb_b; msb_a = a & MSB_M; msb_b =

     b & MSB_M;

     *c = a + b;

     return ((msb_a && msb_b) || ((msb_a ~ msb_b) && !(MSB_M & *c))); #else

     __int128 r = (__int128)a + (__int128)b;

     *c = ( u64 ) r ; return r >> 64;
#e ndi f }


inline void mul_by_word(u64 n, const u64* a, u64 d, u64* c) { u64

carry = 0, carry_tmp; for (int i=0; i < n; i++) { carry_tmp = carry;

          _mul_raw(d, a[i], &(c[i]), &carry);

          carry += _add_raw(c[i], carry_tmp, &(c[i]));
     }

     c [n] = carry;

}


inline void mul(u64 n, const u64* a, const u64* b, u64* c) {

     BigInt tmp;

     mems et (c , 0 , 2* 8* n) ;

     for (u64 i=0; i < n; i++) {
```

```
            mul_by_word(n, a, b[i],       (u64*)tmp);
            add(n + 1, c + i, tmp, c + i);
        }
    }

    /*
        squaring in O(n*logn)
        (a[0] + B a[1] + B^2 a[2] + ...       + B^{n-1} a[n-1])^2 =    a[0]^2  + B^2a[1]^2 + ...      + B^{2(n
            -1)} a[n-1]^2 + 2 \Sum{i,j}       a[i]a[j] B^{i+j}
        (FF FF)^2 = FE 01 00 00 + 01 FC       02 00 + FE 01 = FF FE    00 01
    */
    inline void sqr(u64 n, const u64* a, u64* res) { u6 4
        c , c 2 ;
        memset(res, 0, 2*8*n);
        u64  mult [2] = {0,  0};
        u64  carry[2] = {0,  0};
        for  (int i=0; i<n;  i++) {
            _mul_raw(a[i], a[i], &c, &carry[0]); carry[0] +=
            _add_raw(res[2*i], c, &res[2*i]); carry [1] = 0;
            /* carry[0] - actual 64 bit      carry after squaring,    carry[1]  - still 0 */
            for (int j=i+1; j<n; j++) {
                _mul_raw(a[i], a[j], &mult[0], &mult [1]);

                c2 = _add_raw(mult[0], mult[0], &mult[0]); c =
                _add_raw(mult[1], mult[1], &mult [1]);

                mult[1] |=  c2;  /* mult  =2*a[i]*a[j]     */
                mult [1] += _add_raw(mult[0] ,    carry[0]  ,       &mult[0]) ;
                c += _add_raw(mult[1], carry[1], &mult[1]);         /*mult = mult + carry */

                mult[1] += _add_raw(mult[0], res[i + j], &res[i + j]);       /* res[i+j] += mult[0] */

                carry [0] = mult [1];
                carry [1] = c;
            }
            res[i+n+carry[1]] += _add_raw(res[i+n], carry[0], &res[i+n]) + carry[1]; /* some sort of
                hack, due to carry[1] might be only 0 or 1 */



        }
    }

    inline int word_bit_len(u64 n) {
        int c = 64;
        while (c) {
            if ((((u64)1 << (64-1)) & n) >> (64-1))
```

```
            return c; n <<= 1;
        --c;
    }
    return 0;
}

inline int bigint_bit_len(u64 nWords, const u64* a) { int bit_len =
    nWords * 64; int i=nWords -1; do {
        bit_len-=64;
    } while ((i>=0) && (a[i--] == 0));

    bit_len += word_bit_len(a[i+1] ) ;
    return bit_len;
}

inline void shl(u64 n, const u64* a, u64* res, u64 bits) { u64 buf
    = 0; int chk = bits / 64; bits = bits % 64; u6 4 c ur ;

    if (bit s) {
        for (int i = 0; i < n; i++) {
            cur = a[i] ;
            res[i+chk] = (cur << bits) ~ buf;
            buf = (cur & (MAX_U64 << ( 64-bits ))) >> ( 64-bits );
        }
    }
    else {
        for (int i = 0; i < n; i++) {
            cur = a[i] ; res [i + chk] =
            cur;
        }
    }

    for (int i = 0; i < chk; i++) { res
        [i] = 0;
    }
    res[n + chk] = buf ;
}

inline void shr(u64 n, const u64* a, u64* res, u64 bits) {
```

```
    u64 buf = 0; int chk = bits / 64; bits = bits % 64; u6 4
    c ur ;

    if (bit s) {
        for (int i = n-1+chk; i >= chk; i--) { cur
            = a[i] ;
            res[i-chk] = (cur >> bits) ~ buf;
            buf = (cur & (MAX_U64 >> ( 64-bits ))) << (64 - bits);
        }
    }
    else {
        for (int i = n-1+chk; i >= chk; i--) { cur
            = a[i] ; res [i- chk] = cur;
        }
    }
    for (int i = n; i<n+chk; i + + ) { res [i] = 0;
    }
}

static inline void dump(u64 n, const u64* a) { for
    (int i=n-1;        i>=0;  i--)
        printf(HEX_FORMAT , a[i]); printf("\n") ;
}

inline int cmp(u64 n, const u64* a, const u64* b) { for
    (int i = n - 1; i >= 0; i-- )

        if (a[i] > b[i]) return 1; else if (a[i]
                    < b[i]) return -1;
    }
    return 0;
}

void zero_int(u64 n, u64* a) {
    for (int i=0; i<n; i++) a[i] = 0;
}

void add_mod(u64 n, const u64* a, const u64* b, const u64* m, u64* res) { res [n] =
    add(n, a, b, res);
    if (cmp(n+1, res, m) != -1) sub(n+1, res, m, res);
```

```
void mul_mod(u64 n, const u64* a, const u64* b, const u64* m, u64* res) { /* new
     multiplication with reduction by division */
     BigInt d;
     mul ( n, a, b , d) ;
     divide(n, d, m, NULL, res);

     /* old multiplication using only additions */
     /*
     u64 b_len = bigint_bit_len(n, b);
     BigInt mm, r;
     zero_ int (n,
                  r)
     ;

     copy(mm, a, n);
     for (int i=0; i<b_len; i++)      {
          if (get_bit(b,    i))  add_mod(n,  r,  mm,  m, r);
          add_mod(n, mm, mm, m, mm);
     }
     copy(res, r, n);
     */
}

void sqr_mod(u64 n, const u64* a, const u64* m, u64* res) {
     BigInt d; sqr(n,
     a, d) ;
     divide(n, d, m, NULL, res);
}

void exp_mod(u64 n, const u64* a, const u64* p, const u64* m, u64* res) { u64 b_len =
     bigint_bit_len(n, p) ;
     BigInt mm, r;
     zero_int(n + 1, r) ;
     r[0]  = 1;
     copy(mm, a, n) ;
     for (int i=0; i<b_len; i++) {
          if (get_bit(p,    i)) mul_mod(n,    r,  mm,  m,r);
          mul_mod(n, mm,    mm, m, mm);
     }
     copy (res , r , n) ;
}

void imul(u64 n, const u64* a, const u64* b, u64* c) {
     /* c := a * b, where b    could besigned value     */
     int b_isneg = b[n]& MSB_M;
```

```
    BigInt bb; if (b_isneg) {
        sub(n + 1, zero, b, bb) ;
    }
    else {
        copy(bb, b, n+1) ;
    }

    mul(n+1, a, bb, c);

    if (b_isneg) {
        s ub ( 2* n, zero , c , c ) ;
    }
}

void inv_mod(u64 n, const BigInt a, const BigInt m, BigInt res) {
    /* Old realization of inversion using powering to p-2 */
    /*
    BigInt mm; copy(mm, m, n); mm[0]   -=      2;
    exp_mod(n, a, mm, m, res);
    */

    /* Compute a^{-1} mod m with Extended Euclidean Algorithm */
    BigInt q, tmp, r;
    BigInt t, newt, newr; zero_int(2*n, r); zero_int(n+1, t);   //      t
                        := 0
    zero_int(n + 1, newt);   newt [0] = 1;  //newt   := 1
    copy(r, m, n);    // r := m
    copy(newr, a, n);    //  newr     := a

    while(cmp(n, newr, zero) != 0) {
        divide(n, r, newr, q, tmp); // q := r div newr, tmp := r mod newr copy(r,
        newr, n);            // r := newr
        copy(newr, tmp, n);    //  newr := tmp

        imul(n, q, newt,     tmp);  // tmp  := q*newt

        sub(n+1, t, tmp,     tmp);  // tmp  := t -     tmp
        copy(t, newt, n+1); copy(newt, tmp, n+1);
```

```
     if (t[n] & MSB_M) {
          add(n, t, m, res);
     }
     else {
          c opy ( res , t , n) ;
     }
}

#define div2(n, a) shr((n), (a),       (a),  1)

void divide(u64 n, const u64* a, const u64* b, u64* quotient, u64* reminder) { BigInt q;
     BigInt tmp;
     BigInt r;

     copy(r,  a,  2*n)  ;
     zero_int(2*n,   q);
     zero_int(2*n,
     tmp);

     int k = bigint_bit_len(2*n, a);
     int t = bigint_bit_len(n, b) ;

     if (k < t) {
          if (quotient) copy(quotient, q, 2*n) ;
          copy(reminder, r, n); ret urn;
     }

     k = k-t;
     shl (n , b , tmp , k) ; while (k
     >= 0) {
          if (sub(2*n, r, tmp, r) == 0) {
               q[ k/64 ]    |= (u64) 1 << (k % 64) ;
          }
          else {
               add(2*n, r, tmp, r);
          }

          div2 (2*n, tmp);
          k-- ;
     }

     if (quotient) copy(quotient, q, 2*n);
     copy(reminder , r, n) ;
```

```
GaloisFieldException : :GaloisFieldException(int c) :       code(c)  {}
std::string GaloisFieldException::What() {
     std::stringstream ss; ss <<
     "GaloisFieldException␣#"; s s < < c o de ;
     return ss.str () ;
}

GaloisField::GaloisField() {

}

GaloisField::GaloisField(const BigInt characteristic, int ext, int bitSize) { this-
     >bitSize = bitSize;
     this->wordSize = (bitSize % ARCH == 0) ? bitSize / ARCH : bitSize / ARCH + 1;
     memset(this->characteristic, 0, 2*8*wordSize);
     copy( this->characteristic, characteristic, 2*wordSize);

     if (ext > 2) {
          throw GaloisFieldException(UNSUPPORTED_PARAM);
     }
     this->extension = ext;
     memset(size, 0, 2*8*wordSize);
     memset(halfSize, 0, 2*8*wordSize);
     sqr(wordSize, characteristic, this->size);
     shr(2*wordSize, this->size, this->halfSize,        1);
}

const BigInt* GaloisField::GetChar() { return
     &(this->characteristic) ;
}

int GaloisField::GetWordSize() { return
     wordSize;
}

int GaloisField::GetBitSize() {
     return bitSize;
}

int GaloisField::GetExtension() {
     return extension;
```

```
const GFElement GaloisField: :Zero = {{0}, {0}};
const GFElement GaloisField::Unity = {{1}, {0}};
const GFElement GaloisField::I = {{0}, {1}};

void GaloisField::BaseAdd(const BaseEl a, const BaseEl b,
                          BaseEl c) {
     add_mod(wordSize, a, b, characteristic, c);
}

void GaloisField::BaseSub(const BaseEl a, const BaseEl b,
                          BaseEl c) {
     u64 borrow = sub(wordSize, a, b, c);
     //std::cout << "From Sub\n";
     //std::cout << Dump(a) << "\n" << Dump(b) << "\n" <<
     //std::cout .flush();
                                                    Dump (c) << "\n";
     if (bitSize % ARCH != 0) {
          borrow = c[wordSize - 1] & ((u64)1 << (bitSize %
     }
                                                    ARCH) ) ;
     if ( borrow ! = 0) {
          add(wordSize, c, characteristic, c) ;
     }
}

void GaloisField::Add(const GFElement a, const GFElement
     for (int i=0; i<extension; i++) {
          BaseAdd(a[i], b[i], c[i]);
                                                   b, GFElement c) {
     }
}

int GaloisField::BaseCmp(const BaseEl a, const BaseEl b)
     return cmp(wordSize, a, b);
}

#define div2(n, a) shr((n), (a),       (a), 1)
                                                   {

// Tonelli -Shanks
void GaloisField::BaseSqrt(const BaseEl a, BaseEl r) {
     // p = Q*2^s
     BaseEl Q, pp, z, tmp, c, t, R, b;
     BaseCopy(z, Unity [0]);
     z [0]++ ;
     BaseCopy(Q, characteristic);
     u64 M, S = 1;
     div2(wordSize , Q) ;




     BaseCopy(pp , Q) ;
```

```
//p-1 = 2^s*Q

              while ((Q [0] & 1) == 0) { div2(wordSize , Q) ;
     S++;
}
//finding z - non quadratic residue
while (1) {
     BasePow (z , pp, tmp); if
     (BaseCmp(tmp, Unity [0])) break;
     else
          BaseAdd(z, Unity[0], z);
}

M = S;
BasePow(z, Q, c);
BasePow(a, Q, t);
BaseAdd(Q, Unity[0], tmp); div2(wordSize,
tmp); u6 4 i = 1 ;
BasePow(a, tmp, R);     // tmp = (Q+1)/2

while (1) {
     if (!BaseCmp(t, Unity[0])) {
          BaseCopy(r , R) ; break;
     }
     BaseCopy(tmp, t); //copying value of t, we'll need it later for
     (i=1; i<M; i++) {
          BaseSqr(t, t); if
          (!BaseCmp(t, Unity[0])) bre
          ak ;
     }
                              BaseCopy(b, c);
     for (u64 j=0; j<M-i-1; j++) {
          BaseSqr(b, b);
     }
     M = i;
                               BaseSqr(b, c);
     BaseMul(tmp, c, t); //original value of t multiplied by c^2
     BaseMul(R, b, R);
}
```

```
bool GaloisField::IsQuadraticResidue(const GFElement a) {
     GFElement b;
     Pow(a, halfSize, 2*bitSize, b); return
     Equal(b, Unity);
}

bool GaloisField::IsQuadraticResidueBase(const BaseEl a) { BaseEl
     b, c;
     shr(wordSize, characteristic, b, 1);
     BasePow(a, b, c);
     return BaseCmp(c, Unity[0]) == 0;
}

#define SQRT_ERROR -42

// Algorithm 9 in https://eprint.iacr.org/2012/685.pdf for p bool
GaloisField::Sqrt(const GFElement a, GFElement r) { BigInt p, p2;
                                                                    3 (mod 4)
     GFElement a1, x0, a0, alpha, b;
     BigInt c;
     BaseCopy(c, characteristic);
     c [0] -= 1;  // c = -1

     shr(wordSize, characteristic, p, 2); // p >>2
     shr(wordSize, characteristic, p2,       1);  //  p >> 1

     Pow(a, p, bitSize, a1); // a^{(p-3)/4}
     Mul(a1, a, x0) ; // a^{(p-3)/4 + 1}
     Mul(x0, a1, alpha); // a^{(p-3)/2 + 1}
     Pow(alpha, characteristic, bitSize, a0);       //
     Mul ( a0 , al pha , a0 ) ;
     if (! Equal ( a0 , Unity) )
     {
          return false; // check if quad. residue
     }
     if (BaseCmp(alpha[0], c) == 0) {
          Mul(x0, I, r);
     } else {
          Add(alpha, Unity, b);
          Pow(b, p2, bitSize, b);
          Mul ( x0 , b, r) ;
     }




     #ifdef DEBUG Sqr(r , alpha);
     if (!Equal(alpha, a)) throw GaloisFieldException(SQRT_ERROR);
```

```
    #endif
    return true;



void GaloisField::InitFromString(BaseEl a, const char* str) { u6 4 s _ l e
    n = s t r l e n ( s t r ) ; u6 4 t mp ;

    memset(a, 0, sizeof(BaseEl));

    for (int i = s_len-1; i >= 0; i--) {
        if ((str[i] >= '0') && (str[i] <= '9')) { // 0,1,2,... tmp =
            str [i]        - 48;
        }
        else if ((str[i] >= 'A') && (str[i] <= 'F')) { // A,B,... tmp =
            str [i]        - 55;
        }
        else {
            return;
        }
        a[(s_len - 1 - i) / 16] ^ = ( tmp << (((s_len - 1 - i) % 16)*4) );
    }
}

void GaloisField::InitFromString(GFElement a, const char* str) { std:
    :string s = str;
    size_t pos = s.find_first_of(":,+");

    if (pos != std : : str ing : : npos ) {
        std::string s1 = s.substr(0, pos); std::string s2 =
        s.substr(pos+1);
        InitFromString(a[0],      s1.c_str());
        InitFromString(a[1], s2.c_str());
    } else {
        throw GaloisFieldException(INVALID_DATA);
    }
}

std::string GaloisField::Dump(const GFElement a) { std::stringstream out;
    auto oldFlags = out.flags(); for (int i=wordSize-1; i>=0;   i--)    {
        out << std::uppercase << std: :setfill ('0')    <<std: :setw (16)   <<  std::hex << a[0][i];
    }
    out.flags(oldFlags);
```

```
    out << "u+u";
    for (int i=wordSize-1;     i>=0;  i--)  {
        out << std::uppercase << std: :setfill (' 0 ' )  <<  std :   :setw (16)  <<    std:    :hex    <<
        a[1][i];
    }
    out.flags(oldFlags); out
    << " j " ;

    return out.str () ;
}

std::string GaloisField::Dump(const BaseEl a) { std::stringstream out; for (int i=wordSize-
    1;                          i>=0;  i--)  {
        out << std::uppercase << std: :setfill (' 0 ' )    <<  std ::setw (16) <<  std::hex << a[i];
    }
    return out.str();
}

void GaloisField::Sub(const GFElement a, const GFElement b, GFElement c) { for (int i=0;
    i<extension; i++) {
        BaseSub(a[i], b[i], c[i]);
    }
}

void GaloisField::BaseNeg(const BaseEl a, BaseEl c) { sub(wordSize, characteristic, a, c);
}

void GaloisField::Neg(const GFElement a, GFElement c) { for (int i=0; i<extension; i++) {
        BaseNeg(a[i], c[i]);
    }
}

void GaloisField::BasePow(const BaseEl a, const BigInt n, int nlen, BaseEl b) {
    BaseEl tmp;
    BaseCopy(tmp, a);
    BaseCopy(b, Unity[0]); for (u64
    i=0; i<nlen; i++) { if
    (get_bit(n, i))
            B as eMul ( b , t mp , b) ;
        BaseSqr(tmp, tmp);
    }
```

```
void GaloisField::BasePow(const BaseEl a, const BigInt n, BaseEl b) {
     BasePow(a, n, bitSize, b);
}

void GaloisField::Pow(const GFElement a, const BigInt n, int nlen, GFElement b) { GFElement
     tmp;
     Copy(tmp , a) ;
     Copy(b , Unity); for (u64 i=0; i<nlen; i++) { if (get_bit(n,
     i))
               Mul (b , tmp, b);
          S qr ( t mp , t mp ) ;
     }
}

void GaloisField::Pow(const GFElement a, const BigInt n, GFElement b) {
     Pow(a, n, 2*bitSize, b);
}

void GaloisField::BaseInv(const BaseEl a, BaseEl b) {
     // via extended Euclidean
     inv_mod(wordSize, a, characteristic, b);
}

#define INV_INVALID -43

void GaloisField::Inv(const GFElement a, GFElement b) {
     // Thanks to YaSV for reminding basic complex analysis:
     // 1/(x+yi)   =  (x-yi)/(x^2 + y^2)
     #ifdef DEBUG GFElement t_1;
     Copy(t_1 , a);
     #endif
     BaseEl c, d;
     BaseSqr(a[0], c);
     BaseSqr(a[1], d);
     BaseAdd (c, d, c);
     BaseInv(c, d);

     Copy(b, a);
     BaseNeg(b[1], b[1]);
     MulByBase(b, d, b);

     #ifdef DEBUG




     Mul ( t_1 , b, t_1 ) ;
```

```
    if (!Equal(t_1, Unity)) throw GaloisFieldException(INV_INVALID);
    #endif
}

void GaloisField::BaseMul(const BaseEl a, const BaseEl b, BaseEl c) {
    mul_mod(wordSize, a, b, characteristic, c);
}

void GaloisField::Mul(const GFElement a, const GFElement b, GFElement c) {
    // (x1 + y1*i) *    (x2 + y2*i)   = x1*x2 - y1*y2 +    (y1*x2 + y2*x1)*i
    BaseEl d1, d2, d3, d4;
    BaseMul(a [0] , b[0] , d1);
    BaseMul (a [1] , b[1] , d2) ;

    BaseMul(a [1] , b [0] , d3) ;
    BaseMul(a[0], b[1], d4);

    BaseSub(d1, d2, c [0]) ;
    BaseAdd(d3, d4, c [1]) ;
}

      void GaloisField::BaseSqr(const BaseEl a, BaseEl c) { sqr_mod(wordSize, a,
                                  characteristic, c);
}

void GaloisField::Sqr(const GFElement a, GFElement c) {
    BaseEl d, b;
    BaseSqr(a[0] , d);
    BaseSqr(a[1], b);

    BaseMul (a [0] , a [1] , c[1]);

    BaseSub(d, b, c [0]) ;

    BaseAdd (c [1] , c[1], c[1]);
}

void GaloisField::BaseCopy(BaseEl a, const BaseEl b) { copy(a, b, wordSize);
}

void GaloisField::Copy(GFElement a, const GFElement b) { for (int i=0; i<extension;
    i++) {
        BaseCopy(a[i], b[i]);
```

```
void GaloisField::MulByBase(const GFElement a, const BaseEl e, GFElement c) { for (int
     i=0; i<extension; i++) {
          BaseMul(a[i], e, c[i]);
     }
}

void GaloisField::MulBy2Power(const GFElement a, int pp, GFElement b) {
     Copy(b , a);
     for (int i=0; i<pp; i++) {
          Add(b, b, b);
     }
     /*for (int i=0; i<extension; i++)      {
          BigInt d;
          BaseCopy(d, Zero[0]); shl(wordSize, a[i], d, pp);
          divide(wordSize , d, characteristic , NULL, b[i]);
     }*/
}

bool GaloisField::Equal(const GFElement a, const GFElement b) {
     return (BaseCmp(a[0], b[0]) == 0) && (BaseCmp(a[1], b[1]) == 0);
}

const BigInt* GaloisField::GetSize() { return &size;
}
```

Лістинг файлу ’ec.h’

```
#ifndef EC_H #define EC_H

#def ine NORMAL_POINT 1
#def ine INFINITY_POINT 0

#include <string>
#inc l ude " gf . h"
/*
     In SIDH we use prime p = 2^372 * 3^239 - 1 and Field F_p^2 for 124-bit quantum security level p =
     3 (mod 4)
     Resulting supersingular elliptic curve is E/F_{p^2} : y^2 = x^3 + x; #E = (2^237 * 3^239) ^2
```

```
     For all x in F_p^2 x = a + i*b, where i is square root of QNR in F_p
*/

typedef struct {
     GFElement X;
     GFElement Y;
} EcPoint;

typedef struct {
     GFElement X;
     GFElement Y;
     GFElement Z;
} EcPointProj;

#define PRNG_STATE_LEN 256

/* Dummy realization of PRNG, new stronger implementations must be derived from this */
class PseudoRandomGenerator {
protected:
     unsigned char state[PRNG_STATE_LEN]; virtual void Run();
     virtual void DeriveRandomFromState(unsigned char* ptr, int byteCnt); publ ic :
     PseudoRandomGenerator();
     PseudoRandomGenerator(unsigned char* seed, int len);
     virtual void GenerateSequence(int bit_len, unsigned char* dest);
     virtual void GenerateBaseEl(int bit_len, BaseEl a);
};

#define   WEIERSTRASS   0
#define   MONTHOMERRY   1
#define EDWARDS 2

#define NOT_SUPPORTED -1

class EllipticCurveException {
int c ode ; publ ic :
     EllipticCurveException(int c); s t d : : s t ring What ( ) ;
};

/*
Elliptic curve in Edwards:
     x ^2 + y ^2 = 1 + d*x^2*y^2
```

```
                                or in Weierstrass form: y"2 = x"3 + a*x + b

Scalar Multiplications(all in the projective coordinates):
     - AddAndDouble naive:
     Result: Q = kP
     Q <- 0P
     H <- 1P
                                        for i from 0 to m do if k[i] = 1
               Q <- Q + H H
          <- 2H

     - Montgomery constant-time:
     Result: Q = kP
     Q <- 0P
     H <- 1P
                                        for i from 0 to m do if k[i] = 0
               H <- H + Q Q
               <- 2Q
          else
               Q <- H + Q H
               <- 2H
     - Windowed constant time for Edwards:
     Precomputation: 0P,    1P,  ...  (2^w - 1)P
     Result: Q = kP
     Q <- 0P
     for i from m/w downto 0 do Q
          <-    (2"w)Q
                                                Q <- Q + k[i]P

     - wNAF(not tested)
*/

class EllipticCurve ;

typedef void (EllipticCurve::*TScalarMul)(const EcPointProj*, const BigInt , EcPointProj*, int) ;

class EllipticCurve { u8 form;
     bool isSupersingular; bool
     isBasePointPresent;
```

```
    PseudoRandomGenerator* prng;
    EcPointProj* T; // for precomputations

    TScalarMul scalarMulEngine;

    void AcquireEdwardsForm();
    void ScalarMulNaive(const EcPointProj*, const BigInt, EcPointProj*, int bitLen=0); void
ScalarMulMontgomery(const EcPointProj*, const BigInt, EcPointProj*, int bitLen=0); publ ic :
    GFElement d, a, b; // d for Edwards form; a,b for Weierstrass form BigInt n;
    GaloisField* GF;
    EcPoint UnityPoint;
    EcPointProj UnityPointProj;
    EcPoint BasePoint;

    EllipticCurve();
    EllipticCurve(PseudoRandomGenerator*);
    ~EllipticCurve();
    void InitAsWeierstrass(GaloisField* GF, const BigInt cardinality, const GFElement a, const GFElement
b, const EcPoint* BP = NULL) ; void InitAsEdwards(GaloisField* GF, const BigInt cardinality, const
GFElement d, const EcPoint* BP = NULL);


    void SetPseudoRandomProvider(PseudoRandomGenerator* p) ;

    bool CheckSupersingularity();
    void GenerateBasePoint();
    void GetJInvariant(GFElement J);

    bool IsPointOnCurve(const EcPoint* P);
    bool CheckPointTorsion(const EcPoint* P, const BigInt order);

    bool PointEqual(const EcPoint* A, const EcPoint* B);
    bool PointEqual(const EcPointProj* A, const EcPointProj* B);

    void PointCopy(EcPoint* dst, const EcPoint* src); void PointCopy(EcPointProj* dst, const
    EcPointProj* B);

    std::string PointDump(const EcPoint* X); std::string PointDump(const EcPointProj* X);

    void ToProjective(const EcPoint* src, EcPointProj* dst); void ToAffine(const EcPointProj* src,
    EcPoint* dst);
```

```
    void Add(const EcPoint* A, const EcPoint* B, EcPoint* C);
    void Add(const EcPointProj* A, const EcPointProj* B, EcPointProj* C);

    void Dbl(const EcPoint* A, EcPoint* C);
    void Dbl(const EcPointProj* A, EcPointProj* C);

    void SetNaiveScalarMulEngine(); // DoubleAndAdd algorithm
    void SetMontgomeryScalarMulEngine(); // Suitable for cryptologic usage
    //void SetScalarMulWindowedEngine(const EcPoint* A, int windowSize); // Very fast and still
        suitable for cryptology in Edwards form

    void ApplyDistortionMap(const EcPoint* P, EcPoint* Q);
    void ScalarMul(const EcPoint* P, const BigInt k, EcPoint* Q, int bitLen=0);
    void ScalarMul(const EcPointProj* P, const BigInt k, EcPointProj* Q, int bitLen=0);

};


#endif /* EC_H */
```

Лістинг файлу ’ec.cpp’

```
#include "ec.h"
#include "gf.h"
#include <stdlib.h>
#include <string.h>
#include <time.h>
#inc l ude < s s t re am >
#inc l ude < io s t re am >

PseudoRandomGenerator::PseudoRandomGenerator(unsigned char* seed, int len) {
    srand(time(0));
    memcpy(this->state, seed, len); this->state[0] = 0;
}

PseudoRandomGenerator::PseudoRandomGenerator() {

}

void PseudoRandomGenerator::Run() {
    //int a = rand();
    //*((int*)state) = a;
    //this->state[0];
```

```
void PseudoRandomGenerator::DeriveRandomFromState(unsigned char* ptr, int byteCnt) { //memcpy(ptr,
state, byteCnt); for (int i=0; i<byteCnt; i++) { ptr [i] = 0x55;
    }
}

void PseudoRandomGenerator::GenerateSequence(int bitLen, unsigned char* dest) { int byteCnt = bitLen / 8;
    int restBits = bitLen % 8; for (int i=0; i<byteCnt; i++) {
        Run();
        DeriveRandomFromState(dest + i, 1);
    }
}

void PseudoRandomGenerator::GenerateBaseEl(int bit_len, BaseEl a) { a[bit_len/ARCH] = 0;
    GenerateSequence(bit_len, (unsigned char*)a);
}

EllipticCurveException::EllipticCurveException(int c):          code(c) {}
std::string EllipticCurveException::What() { std::stringstream ss; ss << "EllipticCurveException#"; s s <
    < c o de ; return ss . str () ;
}

EllipticCurve::EllipticCurve()      :  isBasePointPresent(false), scalarMulEngine(&EllipticCurve::
    ScalarMulMontgomery) {}

EllipticCurve::EllipticCurve(PseudoRandomGenerator* p): prng(p), isBasePointPresent(false),
    scalarMulEngine(&EllipticCurve::ScalarMulMontgomery) {}

void EllipticCurve::InitAsWeierstrass(GaloisField* GF, const BigInt cardinality, const GFElement a, const
    GFElement b, const EcPoint* BP) { this ->GF = GF ; this->form = WEIERSTRASS;
    memcpy(this->n, cardinality, 2*BYTES_IN_WORD*GF->GetWordSize()); this-
    >GF->Copy(this->a, a); this->GF->Copy(this->b, b); if (BP != NULL) {
        PointCopy(&(this->BasePoint) , BP);
    } else {
```

```
          GenerateBasePoint();


     this->isSupersingular = CheckSupersingularity(); this->UnityPoint = {


     {{0}, {0}}, {{0}, {0}} };
     this->UnityPointProj = { {{0}, {0}}, {{1}, {0}}, {{0}, {0}} };
}


void EllipticCurve::PointCopy(EcPoint* dst, const EcPoint* src) { this->GF->Copy(dst->X, src->X); this-
     >GF->Copy(dst->Y, src->Y);
}


void EllipticCurve::PointCopy(EcPointProj* dst, const EcPointProj* src) { this->GF->Copy(dst->X, src->X);
     this->GF->Copy(dst->Y, src->Y); this->GF->Copy(dst->Z, src->Z);
}


bool EllipticCurve::PointEqual(const EcPoint* A, const EcPoint* B) { return GF->Equal(A->X, B->X) && GF-
     >Equal(A->Y, B->Y);
}


void EllipticCurve::ToProjective(const EcPoint* src, EcPointProj* dst) {
     GF->Copy(dst->Z, GF->Zero);
     prng->GenerateSequence(GF->GetBitSize()-1, (unsigned char*)dst->Z[0]);              // Z from GF(p)
     GF->MulByBase(src->X, dst->Z[0], dst->X);
     GF->MulByBase(src->Y, dst->Z[0], dst->Y);
}


#define UNDEFINED_POINT -45


void EllipticCurve::ToAffine(const EcPointProj* src, EcPoint* dst) {
     GFElement z_inv;
     #ifdef DEBUG
     if (GF->Equal(src->Z, GF->Zero)) {
          std::cout << "UNDEFINED␣POINT:␣\n" << PointDump(src);
     }
     #endif
     GF->Inv(src->Z, z_inv);
     GF->Mul(src->X, z_inv, dst->X);
     GF->Mul(src->Y, z_inv, dst->Y);
}









void EllipticCurve::InitAsEdwards(GaloisField* GF, const BigInt cardinality, const GFElement
```

```
     d, const EcPoint* BP) { this ->GF = GF; this->form = EDWARDS;
     memcpy(this->n, cardinality, 2*BYTES_IN_WORD*GF->GetWordSize()); this-
     >GF->Copy(this->d, d); if (BP != NULL) {
          PointCopy(&(this->BasePoint) , BP);
     } else {
          GenerateBasePoint () ;
     }

     this->isSupersingular = CheckSupersingularity () ; this->UnityPoint = {

     {{0}, {0}}, {{1}, {0}} };
     this->UnityPointProj = { {{0}, {0}}, {{1}, {0}}, {{1}, {0}} };
}

void EllipticCurve::SetPseudoRandomProvider(PseudoRandomGenerator* p) { this -
     >prng = p;
}

bool EllipticCurve::CheckSupersingularity() {
     BigInt r;
     divide(GF->GetWordSize(), this->n, *(this->GF->GetChar()), NULL, r);            // n = 1  (mod p)
     return GF->BaseCmp(r, GF->Unity[0]) == 0;
}

void EllipticCurve::GetJInvariant(GFElement J) {
     GFElement t,g,c, e; switch (form) {
          case EDWARDS: // 16(1 + 14*d + d^2)^3 / d(1-d)^4

               this->GF->Copy(t, GF->Zero); t
               [0] [0] = 14;
               GF->Mul(t, d, t);
               GF->Add(t, GF->Unity, t);
               GF - > Sqr (d , g) ;
               GF - > Add (t , g , t) ;
               GF->Sqr(t, e);
               GF->Mul(t, e , t ) ;
               GF->MulBy2Power(t, 4, t);

               GF->Sub(GF->Unity, d, c);
               GF-> Sqr(c , c) ;
               GF-> Sqr(c , c) ;
```

```
            GF->Inv(c , e) ;

            GF->Mul(e , t, J) ;
            break;
             case WEIERSTRASS: // 6912 a^3 / (4 a^3 + 27 b^2)

            GF -> Sqr(a, c) ;
            GF->Mul(c, a, c); // a ^ 3
            e[0][0] = 6912;
            GF - > Mul ( e , c , e ) ;

            GF->MulBy2Power(c, 2, c); t
            [0] [0] = 27;
            GF - > Sqr (b , g) ;
            GF->Mul(g , t , g) ;
            GF->Add(g , c , g) ;
            GF->Inv(g , t) ;

            GF->Mul (e , t, J) ; break;
      }
}
```

```
bool EllipticCurve::IsPointOnCurve(const EcPoint* P) {
GFElement r,t,f,g; switch (form) { case EDWARDS:
```

```
            GF->Sqr(P->X , r); GF->Sqr(P->Y , t);
            GF ->Add(r, t, f);
            GF->Mul(r, t, g);
            GF->Mul(g, d, g);
            GF - > Sub ( f , g, f);

            return GF->Equal(f, GF->Unity);
        case WEIERSTRASS :

            GF->Sqr(P->Y , r); GF->Sqr(P->X ,
            t);
            GF->Mul(t, P->X, t);
            GF->Mul(P->X, a, f);
            GF->Add(t, f, t);
```

```
        return GF->Equal(r, b);

}

bool EllipticCurve::CheckPointTorsion(const EcPoint* P, const BigInt order) { EcPoint Z;
    ScalarMul(P, order, &Z);
    return PointEqual(&Z, &UnityPoint);
}

void EllipticCurve::AcquireEdwardsForm() {
    if (form != EDWARDS) throw EllipticCurveException(NOT_SUPPORTED);
}

// In my world (0,1) is an Edwards unity point
void EllipticCurve::Add(const EcPoint* A, const EcPoint* B, EcPoint* C) {
    AcquireEdwardsForm();
    GFElement z1, z2, z3, z4, z5, z6, z7;
    GF->Mul(A->X,      B->Y,z1); //   z1 = x1 *x2
    GF->Mul(A->Y,      B->Y,z2); //   z2 = y1 *y2
    GF->Mul(z1, z2, z3);
    GF->Mul(z3, d, z3); // z3 = d * x1 * x2 * y1 * y2

    GF->Neg(z3, z4); // z4 = - z3 GF-
    >Add(z3, GF->Unity, z3);

    GF->Inv(z3, z3);


    GF->Add(z4, GF->Unity, z4);


    GF->Inv(z4, z4);
    GF->Mul(A->X,      B->Y,z5); //   z5 = x1 *y2
    GF->Mul(A->Y,      B->X,z6); //   z6 = x2 *y1
    GF->Add(z5, z6, z5);
    GF->Sub(z2, z1, z2);

    GF->Mul(z5, z3, C->X);
    GF->Mul(z2, z4, C->Y);
}

void EllipticCurve::Dbl(const EcPoint* A, EcPoint* B) {
    AcquireEdwardsForm();
    GFElement z1, z2, z3, z4, z5;
```

```
    GF ->Sqr (A->Y , z2) ;
    GF->Mul(A ->X, A ->Y, z3) ;
    GF->Add(z3, z3, z3);
    GF->Mul(z1, z2, z4) ;
    GF->Mul(z4, d, z4) ;
    GF->Neg(z4, z5);
    GF->Add(z4, GF->Unity, z4); GF-
    >Inv(z4, z4) ;
    GF->Add(z5, GF->Unity, z5); GF-
    >Inv(z5 , z5) ;
    GF->Sub(z2, z1, z2) ;

    GF->Mul(z4, z3, B->X); GF-
    >Mul(z2, z5 , B->Y) ;
}

void EllipticCurve : :Add(const EcPointProj* P1 , const EcPointProj* P2 , EcPointProj* P3) {
    AcquireEdwardsForm () ;
    #ifdef DEBUG
    EcPointProj P1_copy, P2_copy;
    PointCopy(&P1_copy , P1);
    PointCopy(&P2_copy , P2) ;
    #endif

    /* 10M + 1S */
    GFElement A, B, C, D, E, F, G, T;
    GF->Mul(P1->Z, P2->Z, A); // A = Z1Z2 GF -
    > Sqr (A , B) ; // B = A^2 GF - >Mul (P1 -
    >Y , P2 - >Y , C); // C = Y1Y2 GF ->Mul
    (P1 - >X , P2 ->X , D); // D = X1X2 GF-
    >Mul(C , D, E) ;
    GF - >Mul (E, d, E); // E = dCD
    GF->Sub(B, E, F); // F = B-E GF-
    >Add(B, E, G); // G = B+E

    GF->Add(P1->Y, P1->X, T);
    GF->Add(P2->Y, P2->X, P3->X);
    GF->Mul(P3->X, T, P3->X);
    GF-> Sub(P3->X, C, P3->X) ;
    GF->Sub(P3->X, D, P3->X) ;
    GF->Mul(P3->X, A, P3->X);
    GF->Mul(P3->X, F, P3->X); // X3 = AF((X1+Y1)(X2+Y2)-C-D)

    GF -> Sub(C , D, P3->Y) ;
```

```
    GF - >Mul (P3 - >Y , G, P3 - >Y ) ; // Y3 =

    AG(C-D) GF - >Mul (F , G, P3 - >Z) ; // Z3 =

    #ifdef DEBUG
    if (GF->Equal(P3->Z, GF->Zero)) {
        std : :cout << "!ADD,uZ=0:u\n" << PointDump(&P1_copy) << "\n" << PointDump(&P2_copy) << "\n" <<
            PointDump(P3);
    }
    #endif
}



void EllipticCurve::Dbl(const EcPointProj* P, EcPointProj* P2) {
    AcquireEdwardsForm() ;
    #ifdef DEBUG EcPointProj
    P_copy;
    PointCopy(&P_copy , P) ;
    #endif
    /* 3M + 4S */
    GFElement A, B, C, D, E, F, G;

    GF - > Sqr (P - >Y , A); //A =
    Y^2
    GF->Sqr(P->X,    B);  // B =  X^2
    GF->Sqr(P->Z,    C);  // C =  Z^2
    GF->Add(A, B,    D);  // D =  A+B
    GF->Sub(A, B,    E) ; // E =  A-B
    GF->Add(C, C, F);                          B
   GF->Sub(F, D, F); // F = 2C - A
        GF->Add(P->Y, P->X, G);
    GF ->Sqr(G, G);

    GF - >Mul (F ,G,P2 - >X) ;     //   X2       =       FG
    GF - >Mul (D ,E,P2 - >Y) ;     //   Y2       =       DE
    GF->Mul(D,     F, P2->Z); //    Z3    =       DF
    #ifdef DEBUG
    if (GF->Equal(P2->Z, GF->Zero)) {
        std : : c   << "!DBL   uZ 0:u\n" << PointDump (&P_
            o ut    ;                =
        std : : c   << "  ! u  << GF->Dump(D)    <<  " \ n"
            o ut        D:  "                          ;
        std : : c   << "  ! u  << GF->Dump(F)    <<  " \ n"
            o ut        F:  "                          ;
        std : : c   << "  ! u  << GF->Dump(A)    <<  " \ n"
            o ut        A:  "                          ;
        std : : c   << "  ! u  << GF->Dump(B)    <<  " \ n"
            o ut        B:  "                          ;
        GF- >Add(C  C,  C)
                        ;
        std : : c   << "  ! u  << GF->Dump(C)    <<  " \ n"
            o ut        C:  "                          ;
    }
```

```
#endif


void EllipticCurve::ScalarMulNaive(const EcPointProj* P, const BigInt k, EcPointProj* Q,
    bit Len) {
    EcPointProj H;
    PointCopy(&H, P); // H := A
    PointCopy(Q, &UnityPointProj);
    bitLen = (bitLen != 0) ? bitLen : GF->GetBitSize()*GF->GetExtension();
    for (u32 i=0; i<bitLen; i++) {
    if (get_bit(k, i)) {
            Add(Q, &H, Q);
        }
        Dbl (&H, &H);
    }
}
void EllipticCurve::ScalarMul(const EcPoint* P, const BigInt k, EcPoint* Q, int bitLen) {
    EcPointProj H;
    ToProjective(P, &H);
    ScalarMul(&H, k, &H, bitLen);
    ToAf f ine(&H, Q);
}

void EllipticCurve::ScalarMul(const EcPointProj* P, const BigInt k, EcPointProj* Q, int
    bitLen) {
    (this ->*scalarMulEngine) (P, k, Q, bitLen) ;
}

void EllipticCurve::ScalarMulMontgomery(const EcPointProj* P, const BigInt k, EcPointProj
    , int bitLen) {
    EcPointProj H;
    PointCopy(&H, P); // H := A, H = P1
    PointCopy(Q, &UnityPointProj);
    bitLen = (bitLen != 0) ? bitLen : GF->GetBitSize()*GF->GetExtension();

                                    for (int i=bitLen-1; i>=0; i--) {
                                        if (get_bit(k, i) == 0) {
            Add(Q, &H, &H);
            Dbl (Q , Q) ;
        }
        else {
            Add (Q , &H, Q);
            Dbl (&H, &H);
        }
```

```
// ksi(x,y) = (xi, y"{-1})
void EllipticCurve::ApplyDistortionMap(const EcPoint* P, EcPoint* Q) {
     GF->Mul(P->X, GF->I , Q->X) ;
     GF->Inv(P->X, Q->Y);
}

EllipticCurve::~EllipticCurve() {

}

#define NOT_QUAD_RESIDUE -44

void EllipticCurve::GenerateBasePoint() {
     BaseEl x2;
     GFElement y;
     GF->Copy(y, GF->Zero);

     prng->GenerateBaseEl(GF->GetBitSize()-1, BasePoint.X[0]);
     GF->BaseCopy(BasePoint.X[1] , GF->Zero [1]) ;

     GF->BaseSqr(BasePoint.X[0], x2);
     GF->BaseSub(x2, GF->Unity[0], y[0]);

     GF->BaseMul(x2, d[0], x2);
     GF->BaseSub(x2, GF->Unity[0], x2);
     GF->BaseInv(x2, x2);
     GF->BaseMul(x2, y[0], y[0]);
     #ifdef DEBUG
     if (!GF->IsQuadraticResidue(y)) throw GaloisFieldException(NOT_QUAD_RESIDUE);
     #endif
     GF->Sqrt(y, BasePoint.Y);

}

std::string EllipticCurve::PointDump(const EcPoint* X) {
     return "x:u" + GF->Dump(X->X) + "\n" + "y:u" + GF->Dump(X->Y) + "\n";
}

std::string EllipticCurve::PointDump(const EcPointProj* X) {
     return "X:u" + GF->Dump(X->X) + "\n" + "Y:u" + GF->Dump(X->Y) + "\n" + "Z:u" + GF->Dump (X->Z) + "\n";
```

Лістинг файлу ’isogeny.h’

```
#ifndef ISOGENY_H #define
ISOGENY_H

#include "ec.h"


#define ALICE_KEY 3
#define BOB_KEY 4

typedef struct {
     BigInt d; // param of Edwards curve
     EcPoint P_img;
     EcPoint Q_img;
} SIDHPublicKey;

typedef struct {
     BigInt r;
     EcPoint R;
     int type; // Alice or Bob }
SIDHPrivateKey;

class SIDHEngine {
     /*
          Parameters identifying group structure of E over field GF(p"2), p =            (2"12 * 3"13 * f
               ) - 1 pri me ;
          E - supersingular jEj = (2"12 * 3"13 * f)"2
          Alice must choose secret subgroup E_A = E[2"12] from E as E[2"12] = <[m_A]P_A + Q_A >, m_A -
          secret, P_A, Q_A - public Bob does mutatis mutandis.
     */
     int l2, l3, f;




     void Generate4LTorsionPoint(EcPoint* P4L); void
     Generate3LTorsionPoint(EcPoint* P3L);

     GaloisField* GF;
     PseudoRandomGenerator* prng; publ ic :
     EllipticCurve* BaseCurve; // think usual form ' d be y"2 = x"3 + x, need to check Edwards equiv.
     // Edwards equiv. would be x"2 + y"2 = 1        -x"2y"2  (d  = -1)  or   isomorphic   a=2,d=-2
     EcPoint P_A, Q_A, P_B, Q_B;
     BigInt L3, L2;




     SIDHEngine () ;
```

```
    SIDHEngine(int l2, int l3, int f);
    SIDHEngine(int l2, int l3, int f, BigInt p, int bit_size);
    ~ SIDHEngine() ;

    void GenerateBasePoints();
    void SetBasePoints(const EcPoint* p_A, const EcPoint* q_A, const EcPoint* p_B, const EcPoint *
        q_B) ;

    void GeneratePrivateKey(int type, SIDHPrivateKey* priv); // Alice or Bob void
    GeneratePublicKey(const SIDHPrivateKey* priv, SIDHPublicKey* pub ); void
    DeriveSharedSecret(const SIDHPrivateKey* priv, const SIDHPublicKey* pub_other, GFElement*
    sharedSecret);

    /* via Velu formulas, small isogenies of degree 3 and 4 */
    void Compute3Isogeny(const EllipticCurve* E, const EcPointProj* R, GFElement dImg);                //
        E_dImg <- E_d/<R>
    void Compute4Isogeny(const EllipticCurve* E, const EcPointProj* R, GFElement dImg);

    void Evaluate3Isogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, const
    EcPointProj* P, EcPointProj* PImage); void Evaluate4Isogeny(const EllipticCurve* E, const
    EcPointProj* kernelPoint, const EcPointProj* P, EcPointProj* PImage);

    /*
        Natural isogeny from E to E/<P>, P belongs to E_A or E_B
        via smooth degree isogeny algorithm(perhaps multiplication strategy), isogenies of degree 3^1,
        4^1, phi(E) = F Need to check optimal strategies(I suppose they might be vulnarable to timing
        at tacks)
    */
    //void Compute3LIsogeny(const EllipticCurve* E, const EcPointProj* kernelPoint,
        EllipticCurve* F, int l);
    //void Compute4LIsogeny(const EllipticCurve* E, const EcPointProj* kernelPoint,
        EllipticCurve* F, int l);


    void ComputeAndEvaluate3LIsogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, const
        EcPointProj* P, const EcPointProj* Q, EcPointProj* PImage, EcPointProj*
        QImage, EllipticCurve* F); void ComputeAndEvaluate4LIsogeny(const EllipticCurve* E, const
    EcPointProj* kernelPoint, const EcPointProj* P, const EcPointProj* Q, EcPointProj* PImage,
    EcPointProj* QImage, EllipticCurve* F);
};
```

```
#e ndi f
```

Лістинг файлу ’isogeny.cpp’

```
#include "isogeny.h"

SIDHEngine::SIDHEngine(int l2,       int l3, int  f) : l2(l2),  l3(l3),              f(f) {
}

SIDHEngine::SIDHEngine(int l2,       int l3, int  f, BigInt   p, intbit_size) :      l2(l2),
                                                                                        l3(l3) , f (f)
     GF = new GaloisField(p, 2,      bit_size);
     unsigned char seed [4] = { 0x05, 0x05, 0x05,       0x05  };
     prng = new PseudoRandomGenerator((unsigned char*) seed, 4) ;
     BaseCurve = new EllipticCurve(prng);
     BigInt cardinality;
     add(GF->GetWordSize(), *(GF->GetChar()), GF->Unity[0], cardinality);
     sqr(2*GF->GetWordSize(), cardinality, cardinality);
     GFElement d;
     GF->Neg(GF->Unity, d);

     BaseCurve->InitAsEdwards(GF, cardinality, d, NULL);

     shl(GF->GetWordSize(), GF->Unity[0], L2, l2);

     BaseEl a3;
     GF->BaseCopy(L3, GF->Unity[0]);
     for (int i=0; i<l3; i++) {
          GF->BaseCopy(a3, L3);
          shl(GF->GetWordSize(), a3, L3, 1);
          add(GF->GetWordSize(), L3, a3, L3);
     }
}

SIDHEngine::~SIDHEngine() {
     delete prng;
     delete GF;
     delete BaseCurve;
}

// 1 - x"2y"2 = x "2 + y "2
void SIDHEngine::Generate4LTorsionPoint(EcPoint* P4L) {
     BaseEl x, x2, y, y2;
     GF->Copy (P4L->X, GF->Unity);
     P4L - >X [0] [0] =2; // X = 2
     GF->Copy(P4L->Y, GF->Zero); // Y = 0
     for (; ;) {
          GF->BaseSqr(P4L->X[0] , x2) ;
```

```
          GF->BaseMul(x2, BaseCurve->d[0], x2);
          GF->BaseSub(x2, GF->Unity[0], x2);
          GF->BaseInv(x2, x2);
          GF->BaseMul(x2, y, y); if (GF-
          >IsQuadraticResidueBase(y)) {
               EcPoint J ;
               GF->BaseSqrt(y, P4L->Y [0]);
               BaseCurve->ScalarMul(P4L, L3, &J);
               if (BaseCurve->PointEqual(&J, &BaseCurve->UnityPoint)) { }
          }
          P4L->X [0] [0] ++;




     }
}

void SIDHEngine::Generate3LTorsionPoint(EcPoint* P3L) {


}

void SIDHEngine::GenerateBasePoints() {
     Generate3LTorsionPoint(&P_A);
     Generate4LTorsionPoint(&P_B);
     BaseCurve->ApplyDistortionMap(&P_A, &Q_A);
     BaseCurve->ApplyDistortionMap(&P_B, &Q_B);
}

void SIDHEngine::SetBasePoints(const EcPoint* p_A, const EcPoint* q_A, const EcPoint* p_B. const
     EcPoint* q_B) {
     BaseCurve->PointCopy(&P_A, p_A);
     BaseCurve->PointCopy(&Q_A, q_A);
     BaseCurve->PointCopy(&P_B, p_B);
     BaseCurve->PointCopy(&Q_B, q_B);
}

void SIDHEngine::Compute3Isogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, GFElement D) {
     GFElement c0, c1, c2, c3, t0, t1, C;
     E->GF->Add(kernelPoint->Y, kernelPoint->Z, c0);
     E->GF->Sqr(c0, c0);
     E->GF->Sqr(kernelPoint->Y, c1);
     E->GF->Sqr(kernelPoint->Z, c2);
     E->GF->Sub(c0, c1, c3);
     E->GF->Sub(c3, c2, c3);
     E->GF->Add(c1, c1, t0);
```

```
    E->GF->Add(t0,    c1 ,
                      , t0)
    E->GF->Add(c0,    c3   t1
                      ,    )
    E->GF->Add(t1,    t0   t1
                      ,    )
    E->GF->Add(c2,    c3 ,
                      ,  t0)
    E->GF->Mul(t0,    t1   D);
                      ,

    E->GF-> Sub(c1,   c2 ,
                      ,  t0)

    E->GF->Add(c3,    c3   t1
                      ,    )
    E->GF->Sub(c0 ,   t1   t1
                      ,    )
    E->GF->Mul(t0 ,   t1 ,
                      ,  t0)
    E->GF->Add(t0,    D,   C)
                           ;


    E->GF->Inv(C , C);

    E->GF->Mul(C, D, D);

}


void SIDHEngine::Evaluate3Isogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, const

    EcPointProj* P, EcPointProj* PImage) {

    GFElement t0, t1, t2, t3;

    E->GF->Mul(kernelPoint->Y, P->Z, t0);

    E->GF->Mul(kernelPoint->Z, P->Y, t1);

    E->GF->Add(t0, t1, t2);

    E->GF->Add(P->Y, P->Z, t3);

    E->GF->Sqr(t2, t2);

    E->GF->Mul(t2, t3 , t2);

    E->GF->Sub(t0, t1, t0);

    E->GF->Sqr(t0, t0);

    E->GF->Sub(P->Y, P->Z, t1);

    E->GF->Mul(t0 , t1, t0) ;

    E->GF->Add(t0, t2, PImage->Y);

    E->GF->Sub(t2, t0, PImage->Z);
}


void SIDHEngine::Compute4Isogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, GFElement D)

    {

    GFElement c0, C;

    E->GF->Add(kernelPoint->Y, kernelPoint->Z, c0);

    E->GF->Sqr(c0, c0);

    E->GF->Sqr(c0, C);

    E->GF->Sub(kernelPoint->Y, kernelPoint->Z, c0);

    E->GF->Sqr(c0, c0);

    E->GF->Sqr(c0, c0);

    E->GF->Sub(C, c0, D);




    E->GF->Inv(C, C);
```

```
    E->GF->Mul(C, D, D);


void SIDHEngine::Evaluate4Isogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, const
    EcPointProj* P, EcPointProj* PImage) {
    GFElement t0, t1, t2, t3, t4, t5, c0 ;
    E->GF->Mul(P->Z, kernelPoint->Y, t0);
    E->GF->Mul(P->Y, kernelPoint->Z, t1);
    E->GF->Mul(t0, t1, t2);
    E->GF->Add(t2 , t2 , t2);
    E->GF->Add(t0 , t1, t3);
    E->GF->Sqr(t3, t3);
    E->GF->Sub(t3 , t2 , t5);
    E->GF->Mul(P->Y, P->Z, t4);
    E->GF->Add(kernelPoint->Y, kernelPoint->Z, c0);
    E->GF->Sqr(c0 , c0);
    E->GF->Mul(t4 , c0, t4) ;
    E->GF->Sub(t4, t2, t4);
    E->GF->Add(t4, t5, PImage->Y);
    E->GF->Sub(t4, t5, PImage->Z);
    E->GF->Mul(PImage->Y, t3, t0);
    E->GF->Sub(t5 , t2 , t3);
    E->GF->Mul(PImage->Z, t3, t1);
    E->GF->Add(t0, t1, PImage->Y);
    E->GF->Sub(t0, t1, PImage->Z);
}

void ComputeAndEvaluate3LIsogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, const
    EcPointProj* P, const EcPointProj* Q, EcPointProj* PImage, EcPointProj* QImage, EllipticCurve* F)
    {

}

void ComputeAndEvaluate4LIsogeny(const EllipticCurve* E, const EcPointProj* kernelPoint, const
    EcPointProj* P, const EcPointProj* Q, EcPointProj* PImage, EcPointProj* QImage, EllipticCurve* F)
    {


}
```