

СОДЕРЖАНИЕ

| | |
|---|----|
| Введение..... | 8 |
| 1 Анализ и разделение на виды актуальных алгоритмов цифровой подписи | 10 |
| 1.1 Алгоритм PKCS#1 v1.5 | 10 |
| 1.2 Алгоритм PKCS#1 PSS | 15 |
| 1.3 Алгоритмы EdDSA, HashEdDSA, PureEdDSA..... | 20 |
| 1.4 Алгоритм DSA | 24 |
| 1.5 Алгоритм ECDSA | 28 |
| Вывод по первой главе..... | 31 |
| 2 Методика оценки эффективности систем цифровой подписи на основе библиотеки PyCryptodome..... | 32 |
| 2.1 Общая информация о методике | 32 |
| 2.2 Внутренняя структура методики..... | 34 |
| 2.3 Программная реализация методики..... | 39 |
| 2.4 Оценка скорости работы алгоритмов электронной подписи PyCryptodome | 45 |
| Заключение..... | 48 |
| Список использованных источников..... | 49 |
| Приложение А..... | 51 |
| Приложение Б | 52 |

ВВЕДЕНИЕ

Современная информационная эпоха требует надежных и безопасных методов передачи данных. Одним из таких методов является использование систем цифровой подписи, которые позволяют обеспечить аутентичность и целостность информации. Однако, эффективность таких систем может быть ограничена различными факторами, такими как выбор алгоритма подписи, длина ключа и многими другими. В данном дипломном проекте рассматривается разработка интерактивной методики оценки эффективности систем цифровой подписи на основе библиотеки PyCryptodome. Эта методика позволит улучшить качество и безопасность передачи данных в современном информационном мире.

Актуальность дипломной работы заключается в том, что согласно ТЮВЕ на апрель 2023 года Python является самым популярным языком программирования[1]. Рейтинг самых популярных языков программирования расположен на [рисунке 1 \[1\]](#)



| Apr 2023 | Apr 2022 | Change | Programming Language | Ratings | Change |
|----------|----------|--------|--|---------|--------|
| 1 | 1 | |  Python | 14.51% | +0.59% |
| 2 | 2 | |  C | 14.41% | +1.71% |
| 3 | 3 | |  Java | 13.23% | +2.41% |
| 4 | 4 | |  C++ | 12.96% | +4.68% |
| 5 | 5 | |  C# | 8.21% | +1.39% |
| 6 | 6 | |  Visual Basic | 4.40% | -1.00% |
| 7 | 7 | |  JavaScript | 2.10% | -0.31% |
| 8 | 9 | ▲ |  SQL | 1.68% | -0.61% |
| 9 | 10 | ▲ |  PHP | 1.36% | -0.28% |

Рисунок 1 – Самые популярные языки программирования на апрель 2023 года

А PyCryptodome, в свою очередь, является одной из самых популярных библиотек для имплементации криптографических протоколов и алгоритмов, эти факторы в купе с поддерживаемыми алгоритмами электронной подписи обуславливают выбор данной библиотеки в качестве базиса создание методики, а её пакета Crypto.Signature в качестве объекта исследования данной выпускной квалификационной работы [2].

Цель работы – Повышение эффективности применения алгоритмов цифровой подписи путём рекомендации подходящего алгоритма в зависимости от условий применения

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Анализ и разделение на виды актуальных алгоритмов цифровой подписи.
2. Разработка критериев к методике оценки цифровых подписей.
3. Создание методики оценки эффективности цифровых подписей.
4. Разработка программной реализации методики.
5. Разработка рекомендаций к практическому применению методики.

1 АНАЛИЗ И РАЗДЕЛЕНИЕ НА ВИДЫ АКТУАЛЬНЫХ АЛГОРИТМОВ ЦИФРОВОЙ ПОДПИСИ

Согласно официальной документации Библиотека PyCryptodome поддерживает 3 алгоритма электронной подписи, каждый из которых имеет две вариации [\[3\]](#).

Алгоритмы:

1. RSA со следующими вариациями:
 - PKCS#1 v1.5
 - PKCS#1 PSS
2. EdDSA со следующими вариациями:
 - EdDSA
 - Pure EdDSA
3. DSA со следующими вариациями:
 - DSA
 - ECDSA

Ниже рассмотрены вышеперечисленные алгоритмы и их вариации в соответствии с их документацией.

1.1 Алгоритм PKCS#1 v1.5

Согласно RFC 8017 PKCS#1 v1.5, далее RSASSA-PKCS1-v15 объединяет примитивы RSASP1 и RSAVP1 (см пункт 5.2 RFC 8017 [\[4\]](#)) с методом кодирования EMSA-PKCS1-v15 (см пункт 9.2 RFC 8017 [\[4\]](#)) [\[4\]](#). Он совместим со схемой IFSSA, определенной в IEEE 13639, где примитивы подписи и проверки являются IFSP-RSA1 и IFVP-RSA1, а метод кодирования сообщения - EMSA-PKCS1-v15 [\[5\]](#).

Длина сообщений, на которых может работать RSASSA-PKCS1-v15, либо не ограничена, либо ограничена очень большим числом, в зависимости от хэш-функции, лежащей в основе метода кодирования EMSA-PKCS1-v15.

Предполагая, что вычисление e -ых корней по модулю n является невыполнимым и хэш-функция в EMSA-PKCS1-v15 обладает соответствующими свойствами, предполагается, что RSASSA-PKCS1-v15 обеспечивает безопасную подпись. Более точно, подделка подписей без знания закрытого ключа RSA предполагается вычислительно невозможной. Кроме того, в методе кодирования EMSA-PKCS1-v15 идентификатор хэш-функции встроен в кодирование. Из-за этой особенности злоумышленник, пытающийся найти сообщение с той же подписью, что и ранее подписанное сообщение, должен найти коллизии конкретной используемой хэш-функции; атака на другую хэш-функцию, отличную от выбранной подписантом, не является полезной для злоумышленника.

Примечание: как отмечено в PKCS #1 v1.5, метод кодирования EMSA-PKCS1-v15 имеет такое свойство, что закодированное сообщение, преобразованное в целочисленное представление сообщения, гарантированно большое и по крайней мере отчасти "случайное". Это предотвращает атаки, предложенные Десмедтом и Одлизко, где мультипликативные отношения между представителями сообщений разрабатываются путем факторизации представителей сообщений на набор малых значений (например, набор малых простых чисел) [8]. Корон, Накаш и Стерн показали, что более сильная форма этого типа атаки может быть довольно эффективной против некоторых случаев схемы подписи ISO/IEC 9796-2 [9]. Они также проанализировали сложность этого типа атаки на метод кодирования EMSA-PKCS1-v15 и пришли к выводу, что атака будет непрактичной и потребует больше операций, чем поиск коллизий в основной хэш-функции (т.е. более 2^{80} операций). Coppersmith, Halevi и Jutla впоследствии расширили атаку Корон и др. [10]. Чтобы сломать схему подписи ISO/IEC 9796-1 с восстановлением сообщения. Различные атаки иллюстрируют важность тщательного построения входных данных для примитива RSA-подписи, особенно в схеме подписи с восстановлением сообщения. Следовательно, метод кодирования

EMSA-PKCS-v15 явно включает хэш-операцию и не предназначен для схем подписи с восстановлением сообщения. Кроме того, хотя атака на метод кодирования EMSA-PKCS-v15 неизвестна, рекомендуется постепенный переход к EMSA-PSS (см пункт 9.2 RFC 8017 [\[4\]](#)) в качестве предосторожности против будущих разработок.

Операция генерации подписи алгоритма RSASSA-PKCS1-v15

$$RSASSA - PKCS1 - V1_5 - SIGN (K, M). \quad (1)$$

Вход:

К - закрытый ключ RSA подписанта

М - сообщение для подписи, строка октетов

Выход:

S подпись, строка октетов длиной k, где k - длина в октетах модуля RSA
n

Ошибки: "сообщение слишком длинное"; "модуль RSA слишком короткий"

Шаги:

1. Кодирование EMSA-PKCS1-v15: примените операцию кодирования EMSA-PKCS1-v15 к сообщению М, чтобы получить закодированное сообщение EM длиной k октетов:

$$EM = EMSA - PKCS1 - V15 - ENCODE (M, k). \quad (2)$$

Если операция кодирования выводит "сообщение слишком длинное", выведите "сообщение слишком длинное" и остановитесь. Если операция кодирования выводит "намеренная длина закодированного сообщения

слишком короткая", выведите "модуль RSA слишком короткий" и остановитесь.

2. RSA-подпись:

а. Преобразование закодированного сообщения EM в целочисленное представление сообщения m (см пункт 4.2 RFC 8017 [4]):

$$m = OS2IP(EM). \quad (3)$$

б. Применение примитива подписи RSASP1 (см пункт 5.2.1 RFC 8017 [4]) к закрытому ключу RSA K и целочисленному представлению сообщения m для получения целочисленного представителя подписи s :

$$s = RSASP1(K, m). \quad (4)$$

в. Преобразование целочисленного представителя подписи s в подпись S длиной k октетов (см пункт 4.1 RFC 8017 [4]):

$$S = I2OSP(s, k). \quad (5)$$

4. Вывод подписи S .

Операция проверки подписи алгоритма RSASSA-PKCS1-v15

$$RSASSA - PKCS1 - V1_5 - VERIFY((n, e), M, S). \quad (6)$$

Вход:

(n, e) - открытый ключ RSA подписанта

M - сообщение, подпись которого требуется проверить, октетовая строка

S - подпись, которую необходимо проверить, октетовая строка длиной k ,
где k - длина в октетах модуля RSA n

Выход: "верная подпись" или "неверная подпись"

Ошибки: "слишком длинное сообщение"; "слишком короткий модуль RSA".

Шаги:

1. Проверка длины: если длина подписи S не равна k октетам, вывести "неверная подпись" и остановиться.

2. Проверка RSA:

а. Преобразование подписи S в целочисленный представитель подписи s (см пункт 4.2 RFC 8017 [4]):

$$s = OS2IP(S). \quad (7)$$

б. Применение примитива проверки RSAVP1 (см пункт 5.2.2 RFC 8017 [4]) к открытому ключу RSA (n, e) и целочисленному представителю подписи s для получения целочисленного представителя сообщения m :

$$m = RSAVP1((n, e), s). \quad (8)$$

Если RSAVP1 выдает "целочисленный представитель подписи вне диапазона", вывести "неверная подпись" и остановиться.

с. Преобразование целочисленного представителя сообщения m в закодированное сообщение EM длиной k октетов (см пункт 4.1 RFC 8017 [4]):

$$EM = I2OSP(m, k). \quad (9)$$

Если I2OSP выдает "слишком большое целое число", вывести "неверная подпись" и остановиться.

3. Кодирование EMSA-PKCS1-v1_5: Применение операции кодирования EMSA-PKCS1-v1_5 (см пункт 9.2 RFC 8017 [4]) к сообщению M для получения второго закодированного сообщения EM' длиной k октетов:

$$EM' = EMSA - PKCS1 - V1_5 - ENCODE (M, k). \quad (10)$$

Если операция кодирования выдает "слишком длинное сообщение", вывести "слишком длинное сообщение" и остановиться. Если операция кодирования выдает "недостаточная длина закодированного сообщения", вывести "слишком короткий модуль RSA" и остановиться.

4. Сравнение закодированных сообщений EM и EM'. Если они совпадают, вывести "верная подпись"; в противном случае, вывести "неверная подпись".

Примечание: Другой способ реализации операции проверки подписи - применение "декодирования" (не указанного в данном документе) к закодированному сообщению для восстановления базового хеш-значения, а затем сравнение его с вновь вычисленным хеш-значением. Это имеет преимущество в том, что требуется меньше промежуточного хранения (два хеш-значения вместо двух закодированных сообщений), но недостаток в том, что требуется дополнительный код.

1.2 Алгоритм PKCS#1 PSS

Согласно RFC 8017, PKCS#1 PSS, далее RSASSA-PSS объединяет примитивы RSASP1 и RSAVP1 с методом кодирования EMSA-PSS (см пункт 9.2 RFC 8017 [4]) [4]. Он совместим с схемой подписи с приложением Integer Factorization Signature Scheme with Appendix (IFSSA), как это определено в IEEE 1363, где примитивы подписи и проверки - IFSP-RSA1 и IFVP-RSA1, а метод кодирования сообщения - EMSA4 [5]. EMSA4 немного более общий, чем EMSA-PSS, так как он действует на битовые строки, а не на октетные

строки. EMSA-PSS эквивалентен EMSA4, ограниченному случаю, когда операнды, а также значения хеша и соли являются октетными строками.

Длина сообщений, на которых может работать RSASSA-PSS, либо не ограничена, либо ограничена очень большим числом, в зависимости от хеш-функции, лежащей в основе метода кодирования EMSA-PSS.

Предполагая, что вычисление e -й корней по модулю n невозможно и функции генерации хеша и маски в EMSA-PSS имеют соответствующие свойства, RSASSA-PSS обеспечивает безопасные подписи. Это заверение может быть доказано в том смысле, что сложность подделки подписей может быть непосредственно связана со сложностью инвертирования функции RSA, при условии, что функции генерации хеша и маски рассматриваются как черные ящики или случайные оракулы. Границы в доказательстве безопасности являются в основном "жесткими", что означает, что вероятность успеха и время выполнения для лучшего фальсификатора против RSASSA-PSS очень близки к соответствующим параметрам для лучшего алгоритма инвертирования RSA [\[11\]](#).

В отличие от схемы подписи RSASSA-PKCS1-v1_5 (см пункт 8.2 RFC 8017 [\[4\]](#)), идентификатор хеш-функции не встроен в закодированное сообщение EMSA-PSS, поэтому в теории возможно, чтобы злоумышленник заменил выбранную подписантом хеш-функцию на другую (и потенциально более слабую). Поэтому рекомендуется, чтобы функция генерации маски EMSA-PSS была основана на той же хеш-функции. Таким образом, вся закодированная сообщение будет зависеть от хеш-функции, и для злоумышленника будет трудно заменить другую хеш-функцию, чем ту, которую задумал подписант. Это соответствие хеш-функций необходимо только для предотвращения замены хеш-функций и не является необходимым, если замена хеш-функций решается другими средствами (например, проверяющий принимает только определенную хеш-функцию). См. для дальнейшего обсуждения этих вопросов. Доказуемая безопасность RSASSA-

PSS не зависит от того, является ли функция генерации маски хеш-функцией, применяемой к сообщению.

RSASSA-PSS отличается от других схем подписи на основе RSA тем, что он вероятностный, а не детерминированный, включая случайно генерируемое значение соли. Значение соли повышает безопасность схемы, предоставляя "более жесткое" доказательство безопасности, чем детерминированные альтернативы, такие как Full Domain Hashing (FDH) [11]. Однако случайность не является критической для безопасности. В ситуациях, когда случайная генерация невозможна, можно использовать фиксированное значение или номер последовательности, и результирующая доказуемая безопасность будет аналогична FDH.

Операция генерации подписи алгоритма RSASSA-PSS

$$RSASSA - PSS - SIGN(K, M). \quad (11)$$

Входные данные:

K - закрытый ключ RSA подписчика

M - сообщение, которое должно быть подписано, октетовая строка

Выходные данные:

S - подпись, октетовая строка длиной k, где k - длина в октетах модуля RSA n

Ошибки: "сообщение слишком длинное"; "ошибка кодирования"

Шаги:

1. Кодирование EMSA-PSS: применить операцию кодирования EMSA-PSS (см пункт 9.1.1 RFC 8017 [4]) к сообщению M, чтобы получить закодированное сообщение EM длиной $\text{ceil}((\text{modBits} - 1)/8)$ октетов так, чтобы длина битового представления целого числа OS2IP (EM) (см пункт 4.2 RFC

8017 [4]) была не больше $\text{modBits} - 1$, где modBits - длина в битах модуля RSA n :

$$EM = EMSA - PSS - ENCODE (M, \text{modBits} - 1). \quad (12)$$

Обратите внимание, что длина октетового представления EM будет на один меньше k , если $\text{modBits} - 1$ кратно 8, и равна k в противном случае.

Если операция кодирования выдает "сообщение слишком длинное", выдать "сообщение слишком длинное" и остановиться.

Если операция кодирования выдает "ошибка кодирования", выдать "ошибка кодирования" и остановиться.

2. RSA подпись:

a. Преобразование закодированного сообщения EM в представление целого числа m (см пункт 4.2 RFC 8017 [4])

$$m = OS2IP (EM). \quad (13)$$

b. Применение примитива подписи $RSASP1$ (см пункт 5.2.1 RFC 8017 [4]) к закрытому ключу RSA K

и представлению целого числа сообщения m , чтобы получить представление целого числа подписи s :

$$s = RSASP1 (K, m). \quad (14)$$

c. Преобразование представления целого числа подписи s в подпись S длиной k октетов (см пункт 4.1 RFC 8017 [4])

$$S = I2OSP (s, k). \quad (15)$$

3. Вывод подписи S.

Операция проверки подписи алгоритма RSASSA-PSS

$$RSASSA - PSS - VERIFY ((n, e), M, S). \quad (16)$$

Входные данные:

(n, e) - открытый ключ RSA подписчика

M - сообщение, подпись которого должна быть проверена, октетовая строка

S - подпись, которую нужно проверить, октетовая строка длиной k, где k - длина в октетах модуля RSA n

Выходные данные: "верная подпись" или "неверная подпись"

Шаги:

1. Проверка длины: если длина подписи S не равна k октетам, выдать "неверная подпись" и остановиться.

2. Проверка RSA подписи:

а. Преобразование подписи S в представление целого числа подписи s (см пункт 4.2 RFC 8017 [\[4\]](#)):

$$s = OS2IP(S). \quad (17)$$

б. Применение примитива верификации RSAVP1 (см пункт 5.2.2 RFC 8017 [\[4\]](#)) к открытому ключу RSA (n, e) и представлению целого числа подписи s, чтобы получить представление целого числа сообщения m:

$$m = RSAVP1((n, e), s). \quad (18)$$

Если RSAVP1 выдает "signature representative out of range", выдать "invalid signature" и остановиться.

с. Преобразование представления целого числа сообщения m в закодированное сообщение EM длиной $emLen = \lceil (modBits - 1)/8 \rceil$ октетов, где $modBits$ - длина в битах модуля RSA n (см пункт 4.1 RFC 8017 [4]):

$$EM = I2OSP(m, emLen). \quad (19)$$

Обратите внимание, что если $modBits - 1$ кратно 8, то $emLen$ будет на один меньше k , а в противном случае $emLen$ равен k . Если $I2OSP$ выдает "integer too large", выдать "invalid signature" и остановиться.

3. Проверка EMSA-PSS: применить операцию проверки EMSA-PSS (см пункт 9.1.2 RFC 8017 [4]) к сообщению M и закодированному сообщению EM , чтобы определить, соответствуют ли они друг другу:

$$Result = EMSA - PSS - VERIFY(M, EM, modBits - 1). \quad (20)$$

4. Если $Result = "consistent"$, выдать "valid signature". В противном случае выдать "invalid signature".

1.3 Алгоритмы EdDSA, HashEdDSA, PureEdDSA

Согласно RFC8032 EdDSA— это система цифровой подписи с 11 параметрами [7].

Общая система цифровой подписи EdDSA с ее 11 входными параметрами не предназначена для прямой реализации. Выбор параметров критичен для безопасной и эффективной работы. Вместо этого вы должны реализовать конкретный выбор параметров для EdDSA (например, Ed25519 или Ed448), иногда немного обобщенный для повторного использования кода для охвата Ed25519 и Ed448.

Поэтому точное объяснение общего EdDSA не особенно полезно для реализаторов. Для контекста и полноты здесь приводится краткое описание общего алгоритма EdDSA.

Определение некоторых параметров, таких как n и c , может помочь объяснить некоторые шаги алгоритма, которые неочевидны.

EdDSA имеет 11 параметров [\[12\]](#):

1. Нечетная простая степень p . EdDSA использует эллиптическую кривую над конечным полем $GF(p)$.

2. Целое число b с $2^{(b-1)} > p$. У EdDSA открытые ключи имеют ровно b бит, а подписи EdDSA имеют ровно $2b$ бит. Рекомендуется, чтобы b было кратно 8, так чтобы длины открытого ключа и подписи были целым числом октетов.

3. $(b-1)$ -битное кодирование элементов конечного поля $GF(p)$.

4. Криптографическая хеш-функция H , производящая выходной бит $2b$. Рекомендуются стойкие хэш-функции (то есть хеш-функции, где создание коллизий невозможно) и они не оказывают большого влияния на общую вычислительную сложность EdDSA.

5. Целое число c , которое равно 2 или 3. Секретные скаляры EdDSA являются кратными 2^c . Целое число c является двоичным логарифмом так называемого кофактора.

6. Целое число n с $c \leq n < b$. Секретные скаляры EdDSA имеют ровно $n + 1$ бит, с наивысшим битом (позиция 2^n) всегда установленным и нижними c битами всегда очищенными.

7. Не квадратичный элемент d из $GF(p)$. Обычное рекомендуемое значение — это ближайшее к нулю значение, которое дает приемлемую кривую.

8. Ненулевой квадратичный элемент a из $GF(p)$. Обычная рекомендация для лучшей производительности — это $a = -1$, если $p \bmod 4 = 1$, и $a = 1$, если $p \bmod 4 = 3$.

9. Элемент $B \neq (0,1)$ множества $E = \{ (x, y) \}$ является членом $GF(p)$ х $GF(p)$, таким что $a * x^2 + y^2 = 1 + d * x^2 * y^2$.

10. Нечетное простое число L такое, что $[L]B = 0$ и $2^c * L = \#E$. Число $\#E$ (количество точек на кривой) является частью стандартных данных, предоставленных для эллиптической кривой E , или его можно вычислить как кофактор порядок.

11. Функция "прехэширования" PH . $PureEdDSA$ означает $EdDSA$, где PH является идентификационной функцией, то есть $PH(M) = M$. $HashEdDSA$ означает $EdDSA$, где PH генерирует короткий вывод, независимо от того, насколько длинным является сообщение; например, $PH(M) = SHA-512(M)$.

Точки на кривой образуют группу при сложении, $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$, с формулами [21](#) и [22](#)

$$x_3 = \frac{x_1 * y_2 + x_2 * y_1}{1 + d * x_1 * x_2 * y_1 * y_2}; \quad (21)$$

$$y_3 = \frac{y_1 * y_2 - a * x_1 * x_2}{1 - d * x_1 * x_2 * y_1 * y_2}. \quad (22)$$

Нейтральный элемент в группе - $(0,1)$.

В отличие от многих других кривых, используемых для криптографических приложений, эти формулы являются "полными"; они действительны для всех точек на кривой без исключений. В частности, знаменатели ненулевые для всех входных точек.

Есть более эффективные формулы, которые все еще являются полными и используют однородные координаты, чтобы избежать дорогостоящих инверсий по модулю p .

Кодирование алгоритма EdDSA

Целое число $0 < S < L - 1$ кодируется в форме little-endian как строка из b битов $ENC(S)$.

Элемент $(x, y) \in E$ кодируется как строка из b битов, называемая $ENC(x, y)$, которая является кодированием $(b-1)$ -битов y , объединенных с одним битом, который равен 1, если x отрицательный, и 0, если x не отрицательный.

Кодирование $GF(p)$ используется для определения "отрицательных" элементов $GF(p)$: конкретно, x является отрицательным, если кодирование $(b-1)$ -битов x лексикографически больше кодирования $(b-1)$ -битов $-x$.

Ключи алгоритма EdDSA

Приватный ключ EdDSA представляет собой строку из b битов k . Пусть хэш $H(k) = (h_0, h_1, \dots, h_{(2b-1)})$ определяет целое число s , которое равно 2^n плюс сумме $m = 2^i * h_i$ для всех целых i , $s \leq i < n$. Пусть s определяет кратное $A = sB$. Публичный ключ EdDSA — это $ENC(A)$. Биты $h_b, \dots, h_{(2b-1)}$ используются при подписании.

Подпись алгоритма EdDSA

Подпись EdDSA сообщения M под приватным ключом k определяется как подпись PureEdDSA $PH(M)$. Другими словами, EdDSA просто использует PureEdDSA для подписи $PH(M)$.

Подпись PureEdDSA сообщения M под приватным ключом k - это строка из $2b$ битов $ENC(R) \parallel ENC(S)$. R и S определяются следующим образом. Сначала определим $r = H(h_b \parallel \dots \parallel h_{(2b-1)} \parallel M)$, интерпретируя строки из $2b$ битов в форме little-endian как целые числа в $\{0, 1, \dots, 2^{(2b)} - 1\}$. Пусть $R = [r]B$ и $S = (r + H(ENC(R) \parallel ENC(A) \parallel PH(M))) s \bmod L$.

Проверка подписи алгоритма EdDSA

Чтобы проверить подпись PureEdDSA $ENC(R) \parallel ENC(S)$ сообщения M под публичным ключом $ENC(A)$, выполните следующие действия. Разберите входные данные так, чтобы A и R были элементами E , а S был членом множества $\{0, 1, \dots, L-1\}$. Вычислите $h = H(ENC(R) \parallel ENC(A) \parallel M)$ и проверьте

групповое уравнение $2^c * S B = 2^c * R + [2^c * h] A$ в E . Подпись отклоняется, если разбор не удался (включая S , находящийся вне диапазона) или если групповое уравнение не выполняется.

Проверка EdDSA для сообщения M определяется как проверка PureEdDSA для $PH(M)$.

PureEdDSA, HashEdDSA и именование

Один из параметров алгоритма EdDSA — это "prehash" функция. Она может быть идентичной функцией, что приведет к алгоритму с названием PureEdDSA, или функцией хэширования, такой как SHA-512, что приведет к алгоритму с названием HashEdDSA.

Выбор варианта зависит от того, какое свойство считается более важным между 1) устойчивостью к коллизиям и 2) интерфейсом однократной подписи. Свойство устойчивости к коллизиям означает, что EdDSA защищен даже в случае возможности вычисления коллизий для хэш-функции. Свойство интерфейса однократной подписи означает, что для создания подписи требуется только один проход по входному сообщению. PureEdDSA требует двух проходов по входу. Многие существующие API, протоколы и среды предполагают, что цифровые алгоритмы подписи требуют только один проход по входу и могут иметь проблемы с API или пропускной способностью при использовании других вариантов.

Обратите внимание, что однократная проверка невозможна с большинством использований подписей, независимо от выбранного алгоритма подписи. Это связано с тем, что большую часть времени сообщение нельзя обработать до того, как подпись будет проверена, что требует прохода по всему сообщению.

1.4 Алгоритм DSA

Ниже представлен анализ алгоритма DSA в соответствии с FIPS PUB 186-4 [\[6\]](#)

Параметры DSA

Цифровая подпись DSA вычисляется с помощью набора параметров области, закрытого ключа x , секретного числа k для каждого сообщения, данных для подписи и хэш-функции. Цифровая подпись проверяется с помощью тех же параметров области, открытого ключа y , который математически связан с закрытым ключом x , используемым для генерации цифровой подписи, данных для проверки и той же хэш-функции, которая использовалась при генерации подписи. Эти параметры определяются следующим образом:

p - простое модуль, где $2^{L-1} < p < 2^L$, а L - длина битов p .

q - простой делитель $(p-1)$, где $2^{N-1} < q < 2^N$, а N - длина битов q .

g - генератор подгруппы порядка q в мультипликативной группе $GF(p)$, такой что $1 < g < p$.

x - закрытый ключ, который должен оставаться секретным; x является случайно или псевдослучайно сгенерированным целым числом, таким что $0 < x < q$, т.е. x находится в диапазоне $1, q-1$.

y - открытый ключ, где $y = gx \bmod p$.

k - секретное число, уникальное для каждого сообщения; k является случайно или псевдослучайно сгенерированным целым числом, таким что $0 < k < q$, т.е. k находится в диапазоне $1, q-1$.

Выбор размеров параметров и хэш-функций для DSA

Этот стандарт определяет следующие выборы для пары L и N (длины битов p и q соответственно):

$L = 1024, N = 160$

$L = 2048, N = 224$

$L = 2048, N = 256$

$L = 3072, N = 256$

DSA уникальный Секретный Номер

Перед генерацией каждой цифровой подписи должен быть сгенерирован новый секретный случайный номер k для использования в процессе генерации подписи. Этот секретный номер должен быть защищен от несанкционированного раскрытия и модификации.

k^{-1} является мультипликативным обратным к k по умножению по модулю q ; т.е. $0 < k^{-1} < q$ и $1 = (k^{-1}k) \bmod q$. Этот обратный элемент требуется для процесса генерации подписи.

k и k^{-1} могут быть предварительно вычислены, так как для вычислений не требуется знание сообщения, которое будет подписано. Когда k и k^{-1} предварительно вычислены, их конфиденциальность и целостность должны быть защищены.

Пусть N - это длина q в битах. Пусть $\min(N, \text{outlen})$ обозначает минимум из положительных целых чисел N и outlen , где outlen - это длина блока вывода хэш-функции в битах.

Подпись сообщения M состоит из пары чисел r и s , которые вычисляются в соответствии с следующими уравнениями:

$$r = (gk \bmod p) \bmod q. \quad (23)$$

z - это первые $\min(N, \text{outlen})$ бит хэша (Hash) M .

$$s = (-1 \cdot k (z + xr)) \bmod q. \quad (24)$$

При вычислении s строка z , полученная из $\text{Hash}(M)$, должна быть преобразована в целое число.

Обратите внимание, что r может быть вычислен, когда k , p , q и g доступны, например, когда известны параметры области p , q и g , и k был предварительно вычислен, r также может быть предварительно вычислен, так как для вычисления r не требуется знание сообщения, которое будет

подписано. Значения предварительно вычисленных k , $k-1$ и r должны быть защищены так же, как и закрытый ключ x до тех пор, пока не будет вычислено s .

Значения r и s должны быть проверены для определения, равно ли $r = 0$ или $s = 0$. Если $r = 0$ или $s = 0$, должно быть сгенерировано новое значение k , и подпись должна быть пересчитана. Если подписи генерируются правильно, то крайне маловероятно, что $r = 0$ или $s = 0$.

Подпись (r, s) может быть передана вместе с сообщением верификатору.

Проверка и валидация подписи DSA

Проверка подписи может быть выполнена любой стороной (т.е. подписантом, предполагаемым получателем или любой другой стороной) с использованием открытого ключа подписанта. Подписант может захотеть проверить правильность вычисленной подписи, возможно, перед отправкой подписанного сообщения предполагаемому получателю. Предполагаемый получатель (или любая другая сторона) проверяет подпись, чтобы определить ее подлинность.

Перед проверкой подписи подписанного сообщения параметры области и утвержденный открытый ключ, и идентификатор предполагаемого подписанта должны быть предоставлены верификатору в аутентифицированном виде. Например, открытый ключ может быть получен в виде сертификата, подписанного доверенной стороной (например, ЦС) или на личной встрече с владельцем открытого ключа.

Пусть M' , r' и s' будут полученными версиями M , r и s соответственно; пусть u будет открытым ключом утвержденного подписанта; и пусть N - это длина q в битах. Также пусть $\min(N, \text{outlen})$ обозначает минимум из положительных целых чисел N и outlen , где outlen - это длина блока вывода хэш-функции в битах.

Процесс проверки подписи выглядит следующим образом:

1. Верификатор должен проверить, что $0 < r' < q$ и $0 < s' < q$; если одно из условий не выполняется, подпись должна быть отклонена как недействительная.

2. Если два условия в шаге 1 выполнены, верификатор вычисляет следующее:

$$w = (s')^{-1} \bmod q; \quad (25)$$

z = первые $\min(N, \text{outlen})$ бит хэша (Hash) M' .

$$u1 = (zw) \bmod q; \quad (26)$$

$$u2 = ((r')w) \bmod q; \quad (27)$$

$$v = (((g) u1 (y) u2) \bmod p) \bmod q. \quad (28)$$

1.5 Алгоритм ECDSA

Ниже представлен анализ алгоритма EcDSA в соответствии с FIPS PUB 186-4 [\[6\]](#).

Алгоритм цифровой подписи на эллиптических кривых (ECDSA)

Стандарт цифровой подписи на эллиптических кривых (ECDSA), был разработан для Американского национального института стандартов Аккредитованным комитетом по стандартам финансовых услуг X9.

ANS X9.62 определяет методы генерации и проверки цифровой подписи с использованием алгоритма цифровой подписи на эллиптических кривых (ECDSA). Также в ANS X9.62 включены спецификации для генерации параметров области, используемых при генерации и проверке цифровых подписей. ECDSA является эллиптической кривой аналогом DSA. Ключи

ECDSA не должны использоваться для каких-либо других целей (например, установки ключа).

Генерация параметров области

Данный стандарт определяет пять диапазонов для n (см. [таблицу 1](#)). Для каждого диапазона также указывается максимальный размер кофактора. Следует отметить, что спецификация кофактора h в наборе параметров области является необязательной в ANS X9.62, тогда как реализации, соответствующие FIPS 186-4, должны указывать кофактор h в наборе параметров области. [Таблица 1](#) предоставляет максимальные размеры для кофактора h .

Таблица 1 – Параметры безопасности ECDSA

| Длина n в битах | Максимальный кофактор (h) |
|-------------------|-------------------------------|
| 160-223 | 2^{10} |
| 224-255 | 2^{14} |
| 256-383 | 2^{16} |
| 384-511 | 2^{24} |
| ≥ 512 | 2^{32} |

ECDSA определен для двух арифметических полей: конечного поля GF_p и конечного поля GF_{m^2} . Для поля GF_p требуется, чтобы p было нечетным простым числом.

Предоставляются три типа кривых:

1. Кривые над простыми полями, которые идентифицируются как P-xxx,
 2. Кривые над бинарными полями, которые идентифицируются как B-xxx
 3. Кривые Коблица, которые идентифицируются как K-xxx,
- где xxx указывает битовую длину размера поля.

Приватные/публичные ключи

Пара ключей ECDSA состоит из приватного ключа d и публичного ключа Q , который связан с определенным набором параметров области ECDSA; d , Q и параметры области математически связаны друг с другом. Приватный ключ обычно используется в течение определенного периода времени (т.е. криптопериода); публичный ключ может продолжать использоваться так долго, как требуется проверка цифровых подписей, сгенерированных с использованием связанного приватного ключа (т.е. публичный ключ может продолжать использоваться после криптопериода связанного приватного ключа). Дополнительные руководства можно найти в SP 800-57.

Ключи ECDSA могут использоваться только для генерации и проверки цифровых подписей ECDSA.

Генерация пары ключей

Генерируется пара ключей цифровой подписи d и Q для набора параметров области $(q, FR, a, b \{, domain_paramete_rseed\}, G, n, h)$.

Генерация секретного числа

Перед генерацией каждой цифровой подписи должно быть сгенерировано новое секретное случайное число k для использования в процессе генерации подписи. Это секретное число должно быть защищено от несанкционированного раскрытия и модификации.

k^{-1} является мультипликативным обратным к k по умножению по модулю n ; т.е. $0 < k^{-1} < n$ и $1 = (k^{-1} k) \bmod n$. Этот обратный элемент необходим для процесса генерации подписи.

k и k^{-1} могут быть предварительно вычислены, поскольку знание сообщения, которое будет подписано, не требуется для вычислений. При предварительном вычислении k и k^{-1} их конфиденциальность и целостность должны быть защищены.

Генерация и проверка цифровой подписи ECDSA

Цифровая подпись ECDSA (r , s) должна быть сгенерирована в соответствии с ANS X9.62, используя:

1. Параметры области, правила генерации которых описаны выше
2. Личный ключ, правила генерации которого описаны выше
3. Секретное число для каждого сообщения, правила генерации которого описаны выше
4. Утвержденную хэш-функцию
5. Утвержденный генератор случайных битов.

Цифровая подпись ECDSA должна быть проверена в соответствии с ANS X9.62, используя те же параметры области и хэш-функцию, которые использовались при генерации подписи.

Вывод по первой главе

В результате написания данной главы дипломной работы были проанализированы основные алгоритмы цифровой подписи библиотеки PyCryptodome.

2 МЕТОДИКА ОЦЕНКИ ЭФФЕКТИВНОСТИ СИСТЕМ ЦИФРОВОЙ ПОДПИСИ НА ОСНОВЕ БИБЛИОТЕКИ PYCRYPTODOME

2.1 Общая информация о методике

Интерактивная методика оценки эффективности систем цифровой подписи на основе библиотеки PyCryptodome была разработана для того, чтобы оценить эффективность работы систем цифровой подписи и определить, насколько безопасна и затратна по времени подпись данных с использованием этих систем. Методика написана на языке Python версии 3.11 и использует библиотеку PyCryptodome версии 3.17.

PyCryptodome — это библиотека для языка программирования Python, которая предоставляет набор криптографических функций и алгоритмов, включая системы цифровой подписи. Используя эту библиотеку, можно создавать и проверять аутентичность цифровых подписей.

Функционал методики позволяет проводить тестирование различных алгоритмов цифровой подписи и оценивать их производительность и скорость работы. Также имеется возможность проводить сравнительный анализ различных алгоритмов и выбрать наиболее подходящий для конкретной задачи основываясь на бенчмарках и условиях использования подписи.

Таким образом, данная интерактивная методика является полезным инструментом для обеспечения безопасности передачи данных и защиты от их несанкционированного изменения или подделки отправителя сообщения.

При разработке методики учитывались такие параметры как время, занимаемое на разных этапах генерации подписи, размеры публичных и приватных ключей, а также самой подписи, кроме этого, было уделено внимание распространённости алгоритмов и частота их применения, поддержке существующими программами тех или иных алгоритмов. Также с помощью методики можно подписывать и проверять сообщения, используя 6 вариаций алгоритмов, содержащихся в библиотеке.

Рекомендации по применению методики

Данная методика разработана для пользователей, которым требуется реализовать систему цифровой подписи для своего проекта или его аспекта, в том числе для сетевых администраторов, для пользователей, которые занимаются настройкой средств обработки информации, для программистов, которым необходимо выбрать подходящие крипто примитивы из библиотеки PyCryptodome, а также для пользователей, которые далеки от криптографии, но при этом нуждаются в рекомендации подходящих алгоритмов электронной подписи которые позволят обеспечить необходимую скорость и/или защищённость. Вопросы методики интуитивно понятны и рассчитаны на широкую аудиторию, в следствии чего углублённое знание криптографии не требуется для её использования.

Аналоги

1. Методика выбора ключевой информации для алгоритма блочного шифрования. [\[13\]](#)

Эта методика также программно реализована, но посвящена блочным шифрам, которые не подходят для систем асимметричного шифрования и алгоритмов криптографии с публичным ключом, которые необходимы для систем электронной подписи.

2. Cryptoy. [\[14\]](#)

Приложение представляет собой набор самых популярных алгоритмов шифрования, которые позволяют шифровать сообщения и делиться полученными шифрами. Приложение написано под устройства на android и не подразумевает использование алгоритмов шифрования вне приложения, также оно не позволяет подписывать сообщения и проверять цифровые подписи

3. Cryptool 2 [\[15\]](#).

Cryptool - инструмент, позволяющий визуализировать внутренние механизмы криптографических алгоритмов и атак на них. Однако в нём реализовано только 2 шаблона подписей, остальные посвящены симметричному шифрованию.

2.2 Внутренняя структура методики

Упрощённая внутренняя структура методики в виде блок-схемы представлена на [рисунке 2](#). Полная версия блок схемы приведена в [приложении А](#).

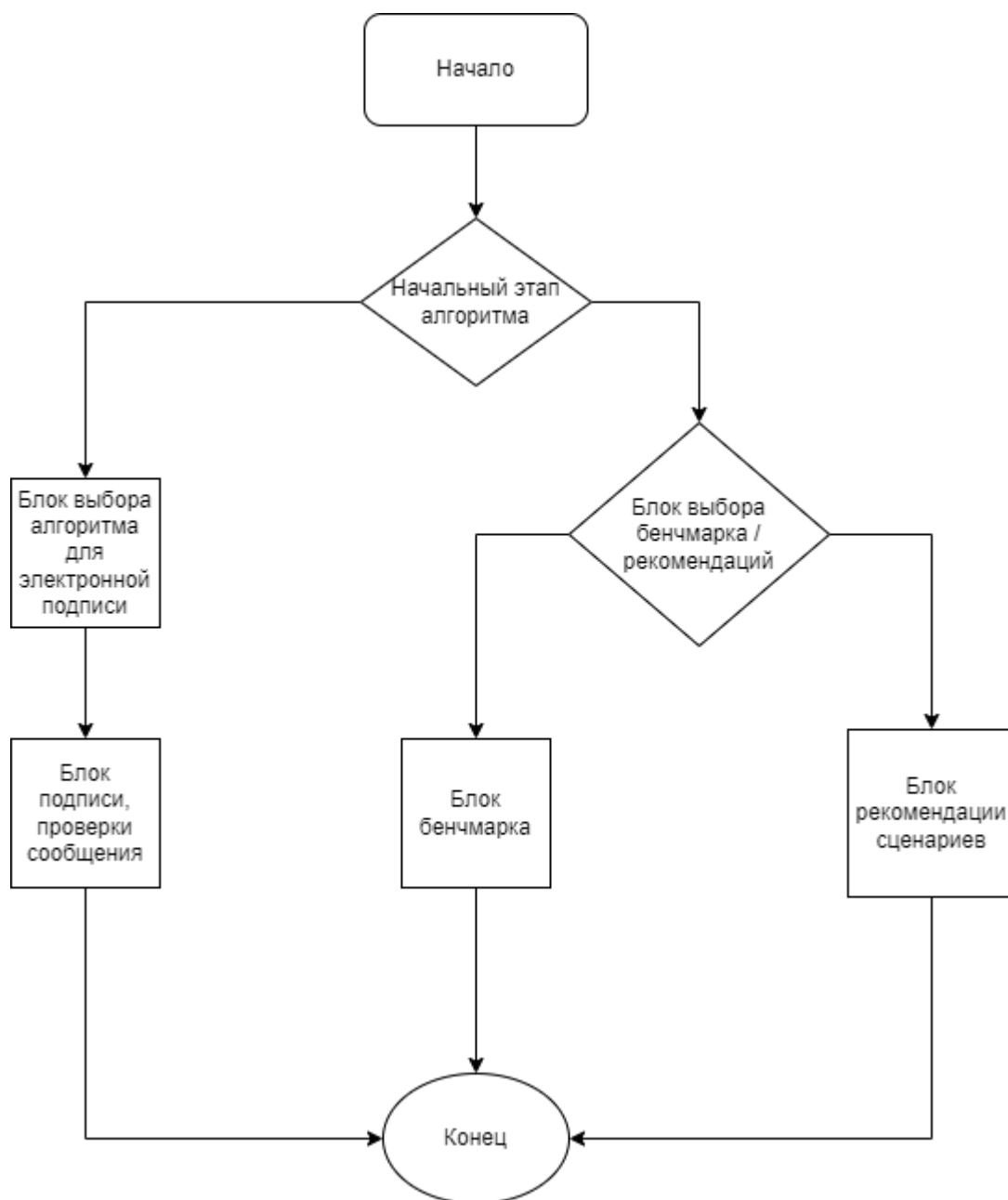


Рисунок 2 – Внутренняя структура методики

Далее детально рассмотрены блоки упрощенной схемы:

После запуска программы ([рисунок 3](#)) в случае отсутствия файлов бенчмарка пользователю предлагается его провести, бенчмарк создаст файлы ключей, что также позволит подписывать сообщения без необходимости их генерации. После чего пользователю предлагается подписать/проверить подпись сообщения или подобрать алгоритм подписи.

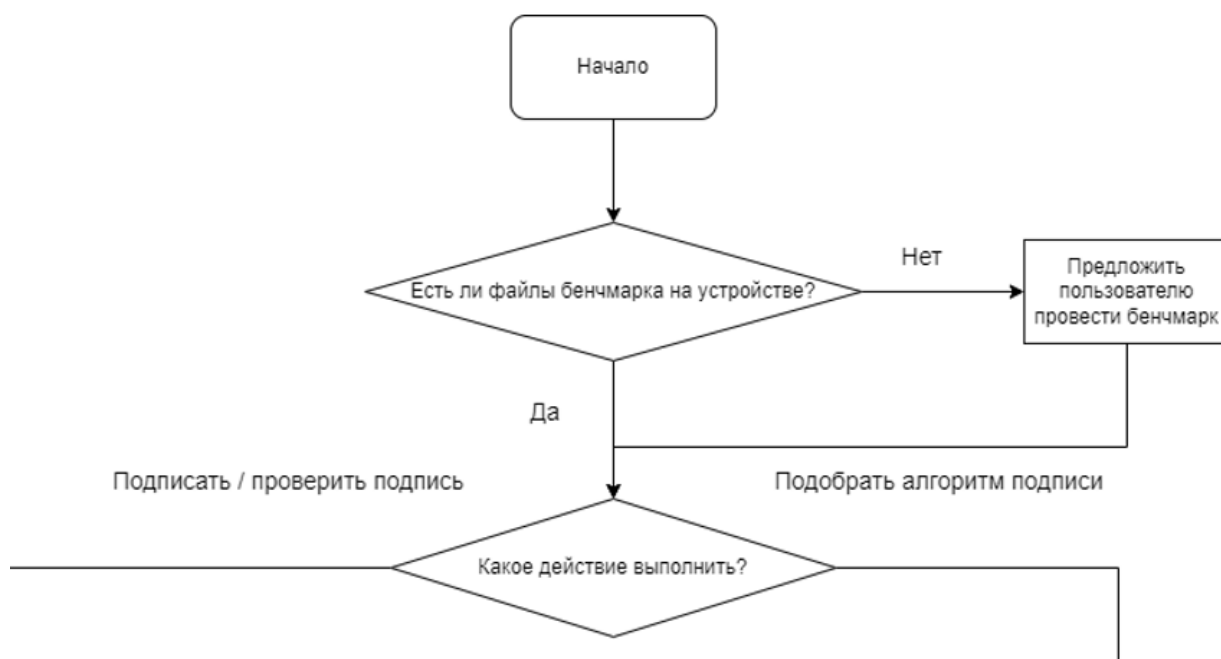


Рисунок 3 – Начальный этап алгоритма

В случае выбора «Подписать/проверить подпись» ([рисунок 4](#)) пользователю предлагается выбрать алгоритмы электронной подписи и алгоритм хэширования. В случае выбора HashedEdDSA или PureEdDSA алгоритм хэширования выбрать невозможно, так как PureEdDSA не использует хэш, а HashedEdDSA использует SHA512 или SHAKE256, который не используется другими алгоритмами подписи и, в следствии этого, не включён в методику.

При выборе оставшихся 4 алгоритмов пользователю предлагается выбрать один из трёх алгоритмов хэширования: SHA256, SHA384, SHA512.

После чего пользователю предлагается ввести сообщение, которое необходимо подписать или проверить.

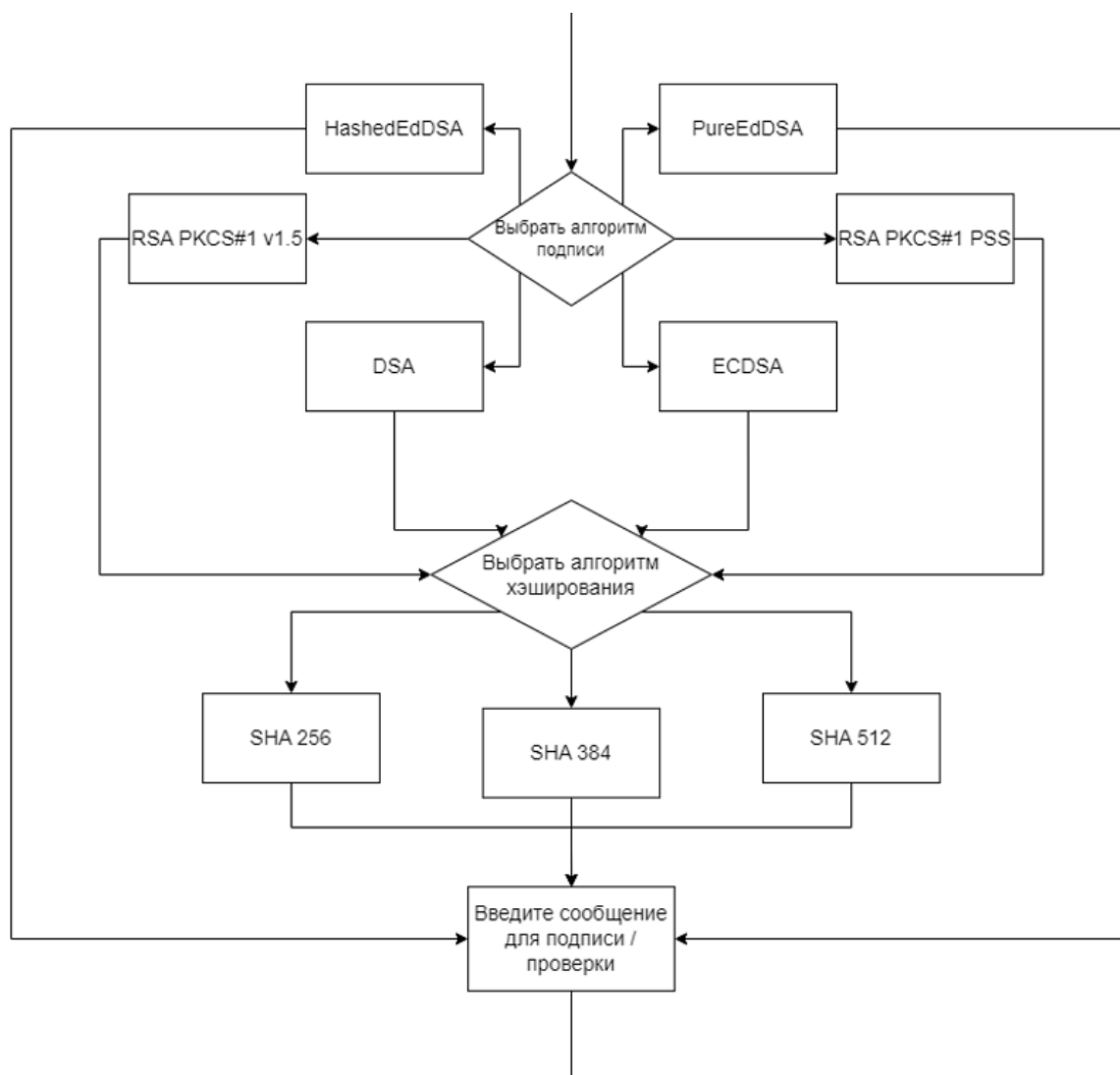
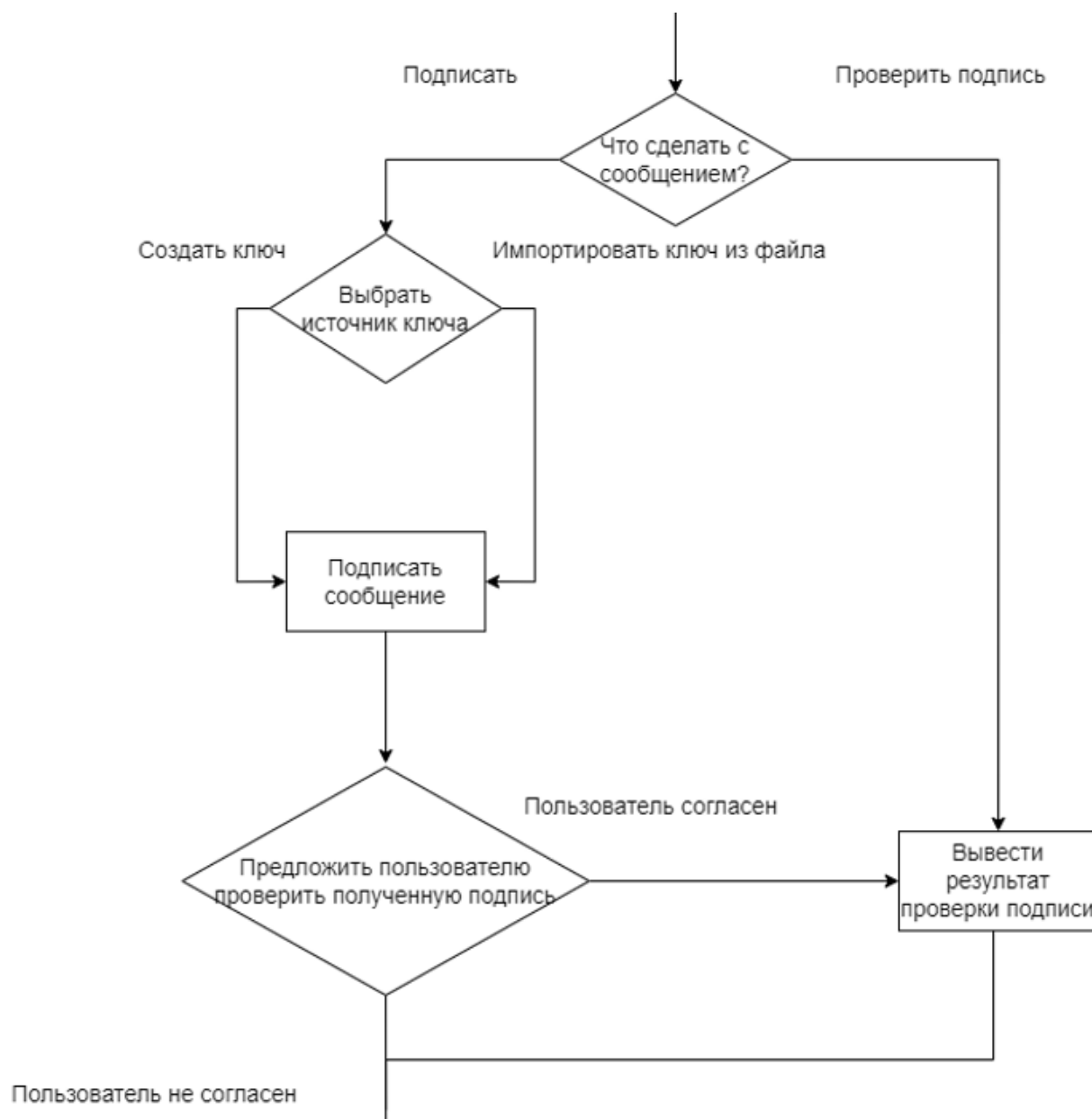


Рисунок 4 – Блок выбора алгоритмов для электронной подписи

Далее введённое сообщение можно подписать или проверить его подпись ([рисунок 5](#)). При подписи сообщения пользователь может выбрать источник ключа: либо использовать заранее сгенерированный ключ в формате PEM, либо создать новый ключ и использовать его для создания подписи сообщения. После подписания сообщения пользователю предлагается проверить подпись сообщения чтобы убедиться в том, что сообщение было подписано корректно и подпись возможно проверить публичным ключом. После подписи / проверки подписи сообщения программа завершается.



В случае выбора «Подобрать алгоритм подписи» ([рисунок 6](#)) пользователю предлагается показать/провести бенчмарк или получить рекомендацию алгоритма.



Рисунок 6 – Блок выбора бенчмарка / рекомендаций

В случае выбора «Показать / Провести бенчмарк» ([рисунок 7](#)) пользователю предлагается вывести в консоль результаты прошлого бенчмарка в виде таблицы либо провести новый бенчмарк, после чего вывести его в консоль. После этого программа завершается. Бенчмарк представляет собой две таблицы в которых обозначено временные характеристики различных алгоритмов.



Рисунок 7 – Блок бенчмарка

В случае выбора «Получить рекомендацию алгоритма» ([рисунок 8](#)) пользователю предлагается указать сценарий, для которого он будет использовать электронную подпись, на основе чего будут выведены рекомендованные и не рекомендованные алгоритмы электронной подписи для каждого сценария соответственно. После получения рекомендации пользователю предлагается вывести бенчмарк для наглядного понимания времени, необходимого для каждого этапа алгоритма электронной подписи с учётом генерации или импорта ключа. После чего программа завершается.



Рисунок 8 – Блок рекомендации сценариев

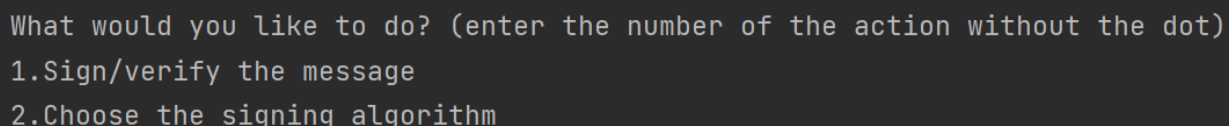
2.3 Программная реализация методики

Методика состоит из трех файлов:

- main.py
- Benchmark.py
- Signer.py

Полный листинг файлов указан в [приложении Б](#).

Файл main.py является главным исполняемым файлом, взаимодействие с методикой происходит через его запуск пользователем. При запуске программы запускается интерактивный сеанс взаимодействия с пользователем в консоли ([рисунок 9](#)).



```
What would you like to do? (enter the number of the action without the dot)
1.Sign/verify the message
2.Choose the signing algorithm
```

Рисунок 9 – консольный интерфейс взаимодействия с пользователем.

Взаимодействие с пользователем реализовано по средствам использования конструкций if – else и match/case, которые, при использовании функционала подписания/проверки подписи методики, задают различные значения переменных, в зависимости от выбора пользователя, после чего переменные передаются в качестве аргументов функциям, которые используют указанный пользователем алгоритм и его параметры, вместе с сообщением для проверки подписи или подписывают сообщение.

В процессе использования методики создаются и используются ключи криптографии с открытым ключом и подписи, полученные в результате работы алгоритмов. Для каждого алгоритма электронной подписи создана отдельная папка ([рисунок 10](#)), в которой хранятся публичный и приватный ключи в формате PEM ([рисунок 11](#)), сами подписи хранятся в общей папке в виде текстовых файлов, где название каждого текстового файла соответствует алгоритму электронной подписи ([рисунок 12](#)).

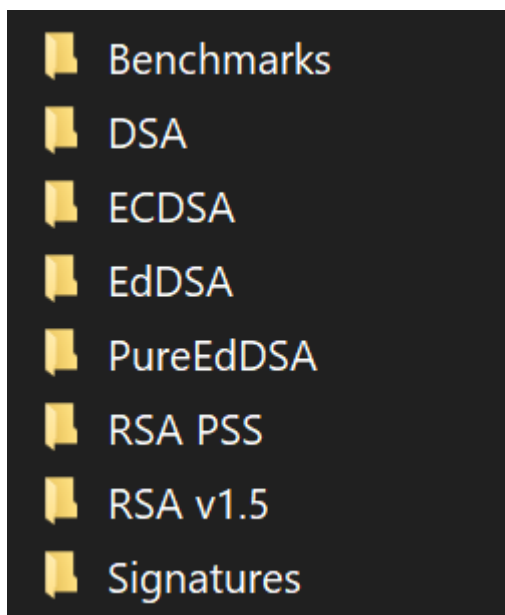


Рисунок 10 – Файловая структура хранения файлов методики

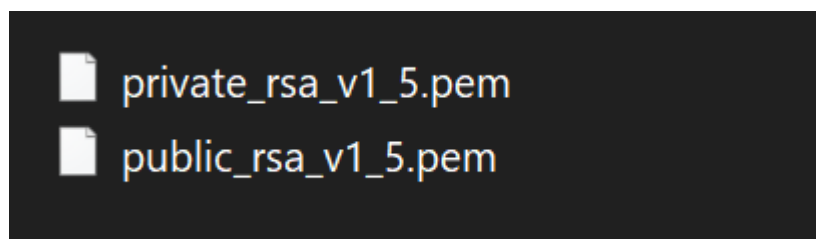


Рисунок 11 – Пример приватного и публичного ключей, используемых методикой



Рисунок 12 – Подписи, полученные в результате использования методики

При использовании функционала бенчмарка методика позволяет провести и / или вывести бенчмарк, в том числе и проведенный до этого, на экран.

На [рисунке 13](#) изображён бенчмарк при импортировании заранее созданных ключей для проверки подписи и подписания сообщения:

```
import benchmark
```

| algorithm | hash | sign | verify | total |
|------------|--------|---------|---------|---------|
| rsa_v1_5 | SHA256 | 0.03400 | 0.00351 | 0.03752 |
| rsa_v1_5 | SHA384 | 0.03355 | 0.00300 | 0.03655 |
| rsa_v1_5 | SHA512 | 0.03350 | 0.00200 | 0.03551 |
| rsa_pss | SHA256 | 0.03275 | 0.00300 | 0.03575 |
| rsa_pss | SHA384 | 0.03227 | 0.00400 | 0.03627 |
| rsa_pss | SHA512 | 0.03233 | 0.00255 | 0.03488 |
| dsa | SHA256 | 0.08068 | 0.08228 | 0.16296 |
| dsa | SHA384 | 0.08088 | 0.08126 | 0.16214 |
| dsa | SHA512 | 0.07934 | 0.08100 | 0.16034 |
| ecdsa | SHA256 | 0.00200 | 0.00651 | 0.00852 |
| ecdsa | SHA384 | 0.00200 | 0.00554 | 0.00755 |
| ecdsa | SHA512 | 0.00201 | 0.01552 | 0.01753 |
| eddsa | SHA512 | 0.00101 | 0.00200 | 0.00301 |
| pure_eddsa | - | 0.00000 | 0.00200 | 0.00200 |

Рисунок 13 – бенчмарк с импортом ключей

На [рисунке 14](#) изображён бенчмарк при создании ключей на этапе подписания сообщения:

```
generate benchmark
```

| algorithm | hash | sign | verify | total |
|------------|--------|----------|---------|----------|
| rsa_v1_5 | SHA256 | 1.25953 | 0.00652 | 1.26605 |
| rsa_v1_5 | SHA384 | 0.24042 | 0.00600 | 0.24643 |
| rsa_v1_5 | SHA512 | 0.85727 | 0.00651 | 0.86378 |
| rsa_pss | SHA256 | 0.64162 | 0.00600 | 0.64762 |
| rsa_pss | SHA384 | 1.04822 | 0.00552 | 1.05374 |
| rsa_pss | SHA512 | 0.77940 | 0.00601 | 0.78541 |
| dsa | SHA256 | 11.39181 | 0.08569 | 11.47750 |
| dsa | SHA384 | 7.66034 | 0.08461 | 7.74495 |
| dsa | SHA512 | 0.20279 | 0.08482 | 0.28761 |
| ecdsa | SHA256 | 0.00251 | 0.00898 | 0.01149 |
| ecdsa | SHA384 | 0.00201 | 0.00851 | 0.01052 |
| ecdsa | SHA512 | 0.00251 | 0.00851 | 0.01102 |
| eddsa | SHA512 | 0.00100 | 0.00601 | 0.00701 |
| pure_eddsa | - | 0.00150 | 0.00601 | 0.00752 |

Рисунок 14 – бенчмарк с созданием ключей

Также стоит отметить, что бенчмарк храниться на устройстве пользователя в двух файлах с расширением .csv, в одном файле находится часть бенчмарка с импортом ключей, а в другом с созданием ([рисунок 15](#)).

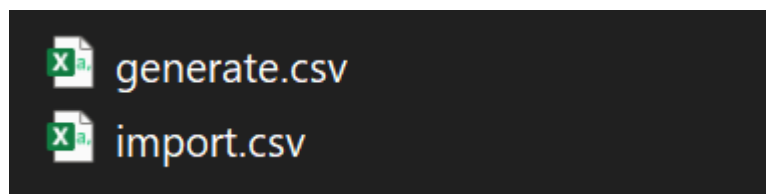


Рисунок 15 – файлы бенчмарка

Для проверки подписи / подписи сообщения используется файл Signer.py, в котором содержатся функции, для обработки выбранным алгоритмом параметров, указанных пользователем, также файл содержит функции для создания и импорта ключей для каждого из 6 алгоритмов. Все вышеперечисленные функции кроме функции импорта имеют два режима: один режим интерактивный, предназначенный для вызова функции в процессе взаимодействия с пользователем, когда включен этот режим функции предлагают пользователю сохранить ключ и подпись или выводят сообщение о результате проверки аутентичности подписи. Второй режим не предполагает консольного вывода и предназначен для вызова функции в процессе проведения бенчмарка.

При использовании функционала рекомендации алгоритма согласно сценарию использования пользователь указывает сценарий, в котором он намеревается использовать систему цифровой подписи и получает рекомендацию алгоритмов, которые стоит и не стоит использовать в зависимости от своего выбора. Ниже приведён список сценариев использования алгоритмов электронной подписи и алгоритмов, которые методика рекомендует и не рекомендует использовать в соответствующих сценариях:

1. Протоколы аутентификации

Рекомендованные алгоритмы: PureEdDSA

Не рекомендованные алгоритмы: DSA, RSA v1.5

2. Сертификаты

Рекомендованные алгоритмы: DSA, RSA v1.5, EdDSA, ECDSA

Не рекомендованные алгоритмы: -

3. Блокчейн

Рекомендованные алгоритмы: ECDSA, EdDSA

Не рекомендованные алгоритмы: DSA, RSA v1.5, RSA PSS

4. Документооборот

Рекомендованные алгоритмы: ECDSA, EdDSA

Не рекомендованные алгоритмы: DSA, RSA v1.5, RSA PSS

5. Аутентификация сообщений

Рекомендованные алгоритмы: PureEdDSA, ECDSA, EdDSA, RSA v1.5, RSA PSS

Не рекомендованные алгоритмы: DSA

Приведённые рекомендации основаны на анализе алгоритмов, приведённых в начале данной выпускной квалификационной работы и бенчмарках, созданных методикой. Они опираются на такие параметры как: уязвимость к различного рода атакам, скорость работы алгоритма, длина подписи, размер ключа и распространённость алгоритмов.

2.4 Оценка скорости работы алгоритмов электронной подписи PyCryptodome

Модуль бенчмарка разработанной методики позволяет создавать таблицы формата .csv, содержащие время выполнения различных этапов алгоритмов цифровой подписи. Далее будет выполнен анализ бенчмарка ([таблица 2](#) и [таблица 3](#)), проведённого на компьютере со следующими характеристиками:

- Процессор: Ryzen 7 5700x
- Оперативная память: AMD R9S48G3206U2S (3200 МГц)

Таблица 2 – Бенмчарк с созданием ключа

| generate | | | | |
|------------|--------|----------|---------|----------|
| algorithm | hash | sign | verify | total |
| rsa_v1_5 | SHA256 | 1.25953 | 0.00652 | 1.26605 |
| rsa_v1_5 | SHA384 | 0.24042 | 0.00600 | 0.24643 |
| rsa_v1_5 | SHA512 | 0.85727 | 0.00651 | 0.86378 |
| rsa_pss | SHA256 | 0.64162 | 0.00600 | 0.64762 |
| rsa_pss | SHA384 | 1.04822 | 0.00552 | 1.05374 |
| rsa_pss | SHA512 | 0.77940 | 0.00601 | 0.78541 |
| dsa | SHA256 | 11.39181 | 0.08569 | 11.47750 |
| dsa | SHA384 | 7.66034 | 0.08461 | 7.74495 |
| dsa | SHA512 | 0.20279 | 0.08482 | 0.28761 |
| ecdsa | SHA256 | 0.00251 | 0.00898 | 0.01149 |
| ecdsa | SHA384 | 0.00201 | 0.00851 | 0.01052 |
| ecdsa | SHA512 | 0.00251 | 0.00851 | 0.01102 |
| eddsa | SHA512 | 0.00100 | 0.00601 | 0.00701 |
| pure_eddsa | - | 0.00150 | 0.00601 | 0.00752 |

Таблица 3 – Бенмчарк с импортом ключа

| import | | | | |
|------------|--------|---------|---------|---------|
| algorithm | hash | sign | verify | total |
| rsa_v1_5 | SHA256 | 0.03400 | 0.00351 | 0.03752 |
| rsa_v1_5 | SHA384 | 0.03355 | 0.00300 | 0.03655 |
| rsa_v1_5 | SHA512 | 0.03350 | 0.00200 | 0.03551 |
| rsa_pss | SHA256 | 0.03275 | 0.00300 | 0.03575 |
| rsa_pss | SHA384 | 0.03227 | 0.00400 | 0.03627 |
| rsa_pss | SHA512 | 0.03233 | 0.00255 | 0.03488 |
| dsa | SHA256 | 0.08068 | 0.08228 | 0.16296 |
| dsa | SHA384 | 0.08088 | 0.08126 | 0.16214 |
| dsa | SHA512 | 0.07934 | 0.08100 | 0.16034 |
| ecdsa | SHA256 | 0.00200 | 0.00651 | 0.00852 |
| ecdsa | SHA384 | 0.00200 | 0.00554 | 0.00755 |
| ecdsa | SHA512 | 0.00201 | 0.01552 | 0.01753 |
| eddsa | SHA512 | 0.00101 | 0.00200 | 0.00301 |
| pure_eddsa | - | 0.00000 | 0.00200 | 0.00200 |

Далее для наглядности результатов представлены гистограммы со сравнением общего времени работы алгоритмов библиотеки с разным хэшем ([рисунки 16](#) и [рисунки 17](#)).

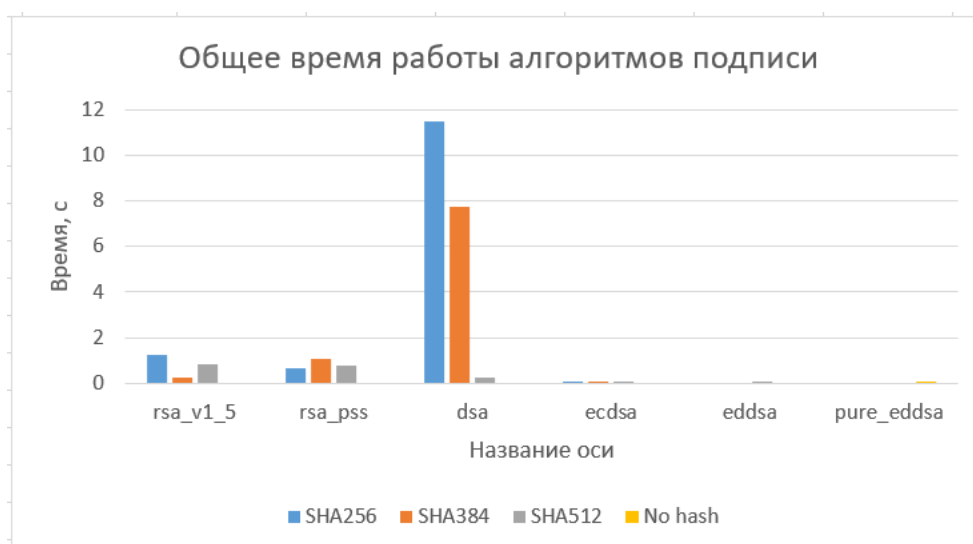


Рисунок 16 - Общее время работы алгоритмов подписи с созданием ключа

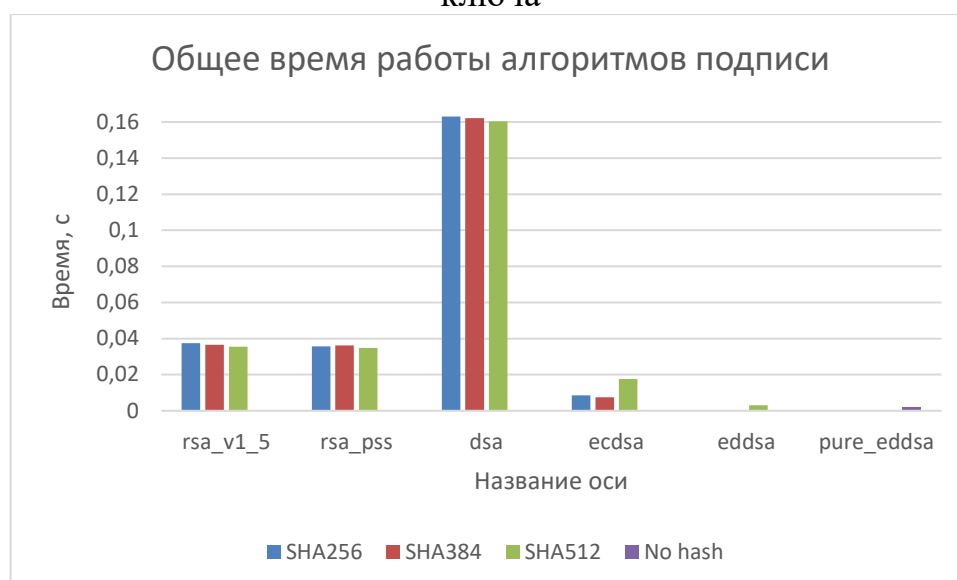


Рисунок 17 - Общее время работы алгоритмов подписи с импортом ключа

Таким образом самым быстрым из алгоритмов, реализованных в библиотеке, оказался eddsa, при этом при наложении подписи на хэш требуется больше времени чем при подписи сообщения на прямую. Из хэшей, используемых в методике при использовании этого алгоритма можно использовать только SHA512. Самым же медленным оказался dsa, он примерно в 4 раза медленнее второго самого медленного алгоритма – RSA v1.5 при использовании заранее созданных ключей и в 9,5 раз при создании ключа на этапе подписи.

ЗАКЛЮЧЕНИЕ

В заключение диплома можно отметить, что разработка интерактивной методики оценки эффективности систем цифровой подписи на основе библиотеки PyCryptodome является актуальной и важной задачей в современном информационном обществе. Безопасность передачи данных становится все более критической, и использование систем цифровой подписи является необходимым условием для защиты информации от несанкционированного изменения.

Разработанная методика позволяет проводить тестирование различных алгоритмов цифровой подписи, определять их производительность и скорость работы. Таким образом, пользователь может выбрать наиболее подходящий алгоритм для конкретной задачи и обеспечить максимальную защищённость передачи данных.

Использование библиотеки PyCryptodome в данной методике обеспечивает высокую точность и надежность результатов, а также удобство в использовании. Разработанная методика может быть применена в различных областях, где требуется обеспечение целостности и аутентичности передачи данных, включая банковскую сферу, государственные и медицинские учреждения и другие.

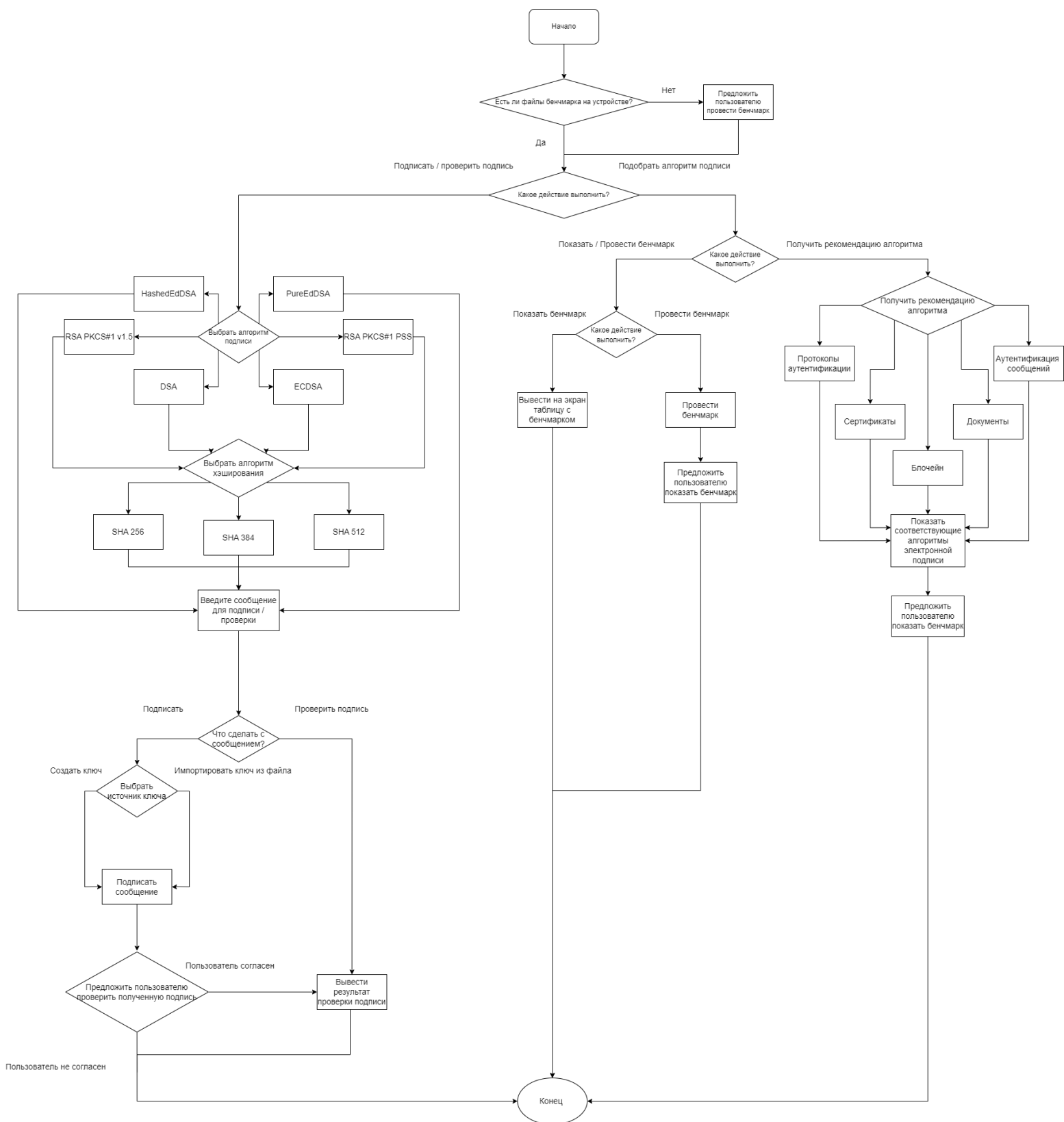
Таким образом, разработанная интерактивная методика является полезным инструментом для обеспечения защищённости передачи данных и защиты от их несанкционированной модификации и подделки их авторства.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. TIOBE Index for April 2023 // Tiobe [Электронный ресурс]. – 2023. – URL: <https://www.cryptool.org/en/ct2/> (дата обращения: 20.02.2023).
2. 3 Best Python Encryption Libraries in 2023 // TLe Apps [Электронный ресурс]. – 2023. – URL: <https://tleapps.com/best-python-encryption-libraries/> (дата обращения: 20.02.2023).
3. Welcome to PyCryptodome's documentation // PyCryptodome [Электронный ресурс]. – 2023. – URL: <https://www.pycryptodome.org> (дата обращения: 20.02.2023).
4. RFC 8017 // Datatracker [Электронный ресурс]. – 2016. – URL: <https://datatracker.ietf.org/doc/html/rfc8017> (дата обращения: 10.03.2023).
5. 1363-2000 - IEEE Standard Specifications for Public-Key Cryptography // IEEE Xplore [Электронный ресурс]. – 2000. – URL: <https://ieeexplore.ieee.org/document/891000> (дата обращения: 10.03.2023).
6. Digital Signature Standard (DSS) // NIST Technical Series Publications [Электронный ресурс]. – 2013. – URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf> (дата обращения: 20.03.2023).
7. RFC 8032 // Datatracker [Электронный ресурс]. – 2017. – URL: <https://datatracker.ietf.org/doc/html/rfc8032> (дата обращения: 20.03.2023).
8. Desmedt, Y., A. Odlyzko A Chosen Text Attack on the RSA Cryptosystem and Some Discrete Logarithm Schemes // Lecture Notes in Computer Science. - Heverlee: 1985. - С. 516-522.
9. Coron, J., Naccache, D., J. Stern On the Security of RSA Padding // Lecture Notes in Computer Science. - Heverlee: 1999. - С. 1-18.
10. Coppersmith, D., Halevi, S., C. Jutla ISO 9796-1 and the new forgery strategy // Rump session of Crypto. - IBM, 1999. - С. 1-17.

11. Bellare, M., P. Rogaway The Exact Security of Digital Signatures - How to Sign with RSA and Rabin // Lecture Notes in Computer Science. - Herverlee: 1996. - С. 399-416.
12. Bernstein, D., Duif, N., Lange, T., Schwabe, P., B. Yang High-speed high-security signatures // Journal of Cryptographic Engineering. - 2012. - №2. - С. 77-89.
13. Жданов О. Методика выбора ключевой информации для алгоритма блочного шифрования. - Инфра-М, 2013. - 88 с.
14. Появилось приложение для интерактивного обучения основам криптографии // Naked science [Электронный ресурс]. – 2014. – URL: <https://naked-science.ru/article/hi-tech/cryptography-is-fun> (дата обращения: 30.04.2023).
15. About CrypTool 2 // Cryptool [Электронный ресурс]. – 2023. – URL: <https://www.cryptool.org/en/ct2/> (дата обращения: 01.05.2023).

ПРИЛОЖЕНИЕ А



ПРИЛОЖЕНИЕ Б

main.py:

```
from Signer import sign_rsa_v1_5, sign_rsa_pss, sign_eddsa, sign_dsa, sign_ecdsa,
sign_pure_eddsa
from Signer import verify_rsa_v1_5, verify_rsa_pss, verify_eddsa, verify_dsa, verify_ecdsa,
verify_pure_eddsa
import sys
from Benchmark import bench, show_bench
import os.path
```

```
def signer_func(alg, hash_chosen, key_source, message):
```

```
    signers = {
        '1': sign_rsa_v1_5,
        '2': sign_rsa_pss,
        '3': sign_dsa,
        '4': sign_ecdsa,
        '5': sign_eddsa,
        '6': sign_pure_eddsa
    }
```

```
    return signers[alg](hash_chosen, key_source, message)
```

```
def verifier_func(alg, hash_chosen, key, signature, message):
```

```
    verifiers = {
        '1': verify_rsa_v1_5,
        '2': verify_rsa_pss,
        '3': verify_dsa,
        '4': verify_ecdsa,
        '5': verify_eddsa,
        '6': verify_pure_eddsa
    }
```

```
    return verifiers[alg](hash_chosen, key, signature, message)
```

```

if __name__ == "__main__":
    if not (os.path.isfile("Benchmarks\generate.csv") and
os.path.isfile("Benchmarks\import.csv")):
        print("No benchmarks have been found, would you like to perform a benchmark?\n"
            "It is recommended to do so as it will automatically create key pares for all signature
algorithms\n"
            "Warning! This will overwrite all saved keys and signatures"
            "y/n")
        if input() == 'y':
            bench()

print("What would you like to do? (enter the number of the action without the dot) \n"
    "1.Sign/verify the message \n"
    "2.Choose the signing algorithm \n")

match input():
    case '1':
        print("Choose an algorithm:"
            "\n 1. Rsa PKCS#1 v1.5"
            "\n 2. Rsa PKCS#1 PSS"
            "\n 3. DSA"
            "\n 4. ECDSA"
            "\n 5. HashedEdDSA"
            "\n 6. PureEdDSA")
        a = input()
        if a != "1" and a != "2" and a != "3" and a != "4" and a != "5" and a != "6":
            print("Unsupported algorithm")
            sys.exit()
        if a != '5' and a != '6':
            print("Select a Hash algorithm:"
                "\n1. SHA256"
                "\n2. SHA384"
                "\n3. Sha512")

```

```

h = input()
if h != "1" and h != "2" and h != "3":
    print("Unsupported hash algorithm")
    sys.exit()
else:
    h = None
print("Enter the message to be signed or verified")
m = input().encode('utf-8')
print("Would you like to: \n"
      "1.Sign\n"
      "2.Verify\n"
      "the message?")
action = input()
if action == '2':
    try:
        verifier_func(a, h, None, None, m)
    except Exception as e:
        print(e)
        sys.exit()
elif action == '1':
    print("Choose the key source:"
          "\n1. Generate the key"
          "\n2. Import the key from file")
    s = input()
    if s == "1":
        k = 'generate'
    else:
        if s == "2":
            k = 'import'
        else:
            print("Wrong key source")
            sys.exit()
    try:
        key, signature = signer_func(a, h, k, m)

```



```

except Exception as e:
    print(e)
    sys.exit()
if input("Would you like to verify the signature? y/n\n") == 'y':
    try:
        verifier_func(a, h, key, signature, m)
    except Exception as e:
        print(e)
        sys.exit()
else:
    print("Wrong action")
    sys.exit()

case '2':
    print("Would you like to: \n"
          "1. Perform/see benchmarks of all available algorithms on this device\n"
          "2. Get a recommendation for which algorithm to use in yours use case\n")
    b = input()
    if b == '1':
        b_1 = input("Would you like to:\n"
                    "1. Perform a benchmark\n"
                    "2. See previous benchmarks\n")
        if b_1 == '1':
            bench()
            print("Would you like to see the results of the benchmark?\n"
                  "y/n\n")
            if input() == 'y':
                show_bench()

        elif b_1 == '2':
            show_bench()
        else:
            print("Wrong action")
            sys.exit()

```

```

elif b == '2':
    print("What are you going to use the signature for?"
          "1. Authentication protocols\n"
          "2. Certificates\n"
          "3. Blockchain\n"
          "4. Documents\n"
          "5. Message authentication\n")
match input():
    case '1':
        print("Recompounded algorithms: PureEdDSA\n"
              "Not recommended algorithms: DSA, RSA v1.5\n")
    case '2':
        print("Recompounded algorithms: DSA, RSA v1.5, EdDSA, ECDSA\n"
              "Not recommended algorithms: - \n")
    case '3':
        print("Recompounded algorithms: ECDSA, EdDSA\n"
              "Not recommended algorithms: DSA, RSA v1.5, RSA PSS\n")
    case '4':
        print("Recompounded algorithms: ECDSA, EdDSA\n"
              "Not recommended algorithms: DSA, RSA v1.5, RSA PSS\n")
    case '5':
        print("Recompounded algorithms: PureEdDSA, ECDSA, EdDSA, RSA v1.5,
RSA PSS\n"
              "Not recommended algorithms: DSA\n")
    case _:
        print("wrong action")
        sys.exit()
if input("Would you like to see the benchmark of all supported signature
algorithms?\n")
    "y/n") == 'y':
    try:
        show_bench()
    except Exception as e:
        print(e)

```

```
        sys.exit()
    else:
        print("Wrong action")
        sys.exit()

case _:
    print("Wrong action")
    sys.exit()
```

Benchmark.py:

```
from Signer import sign_rsa_v1_5, sign_rsa_pss, sign_eddsa, sign_dsa, sign_ecdsa,
sign_pure_eddsa
from Signer import verify_rsa_v1_5, verify_rsa_pss, verify_eddsa, verify_dsa, verify_ecdsa,
verify_pure_eddsa
import time
import csv
from prettytable import from_csv

def bench():
    signers = [sign_rsa_v1_5, sign_rsa_pss, sign_dsa, sign_ecdsa]
    verifiers = [verify_rsa_v1_5, verify_rsa_pss, verify_dsa, verify_ecdsa]
    n_h_s = [sign_eddsa, sign_pure_eddsa]
    n_h_v = [verify_eddsa, verify_pure_eddsa]
    hashes = ['1', '2', '3']
    result = [['algorithm', 'hash', 'sign', 'verify', 'total']]
    alg = []
    modes = ['import', 'generate']
    for mode in modes:
        print("Performing ", mode, "benchamrk")
        for id, sign in enumerate(signers):
            for hash in hashes:
                alg.append(verifiers[id].__name__.replace('verify_', ''))
                alg.append((lambda hash: "SHA256" if hash == '1' else "SHA384" if hash == '2' else
"SHA512")(hash))
                start_t = time.time()
                sign(hash, mode, b'To be signed', 'm')
                sign_t = time.time() - start_t
                pre_ver = time.time()
                verifiers[id](hash, None, None, b'To be signed', 'm')
                verify_time = time.time() - pre_ver
                total_time = time.time() - start_t
```

```

        alg.append(f'{sign_t:.5f}')
        alg.append(f'{verify_time:.5f}')
        alg.append(f'{total_time:.5f}')
        result.append(alg)
        alg = []
    for id, sign in enumerate(n_h_s):
        alg.append(n_h_v[id].__name__.replace('verify_', ''))
        if n_h_v[id].__name__.replace('verify_', '') == 'eddsa':
            alg.append('SHA512')
        else:
            alg.append('-')
        start_t = time.time()
        sign("", mode, b'To be signed', 'm')
        sign_t = time.time() - start_t
        pre_ver = time.time()
        n_h_v[id]("", None, None, b'To be signed', 'm')
        verify_time = time.time() - pre_ver
        total_time = time.time() - start_t
        alg.append(f'{sign_t:.5f}')
        alg.append(f'{verify_time:.5f}')
        alg.append(f'{total_time:.5f}')
        result.append(alg)
        alg = []
    with open('Benchmarks\\{0}.csv'.format(mode), 'w', newline='') as myFile:
        writer = csv.writer(myFile, delimiter=';')
        writer.writerows(result)
        result = [['algorithm', 'hash', 'sign', 'verify', 'total']]
        print("done")

```

```

def show_bench():
    print("import benchmark")
    with open("Benchmarks\\import.csv") as fp:
        table = from_csv(fp)

```

```
    print(table)
print("generate benchmark")
with open("Benchmarks\generate.csv") as fp:
    table = from_csv(fp)
    print(table)

if __name__ == "__main__":
    show_bench()
```

Signer.py:

```
from Crypto.Signature import pkcs1_15, DSS, pss, eddsa
from Crypto.Hash import SHA256, SHA384, SHA512
from Crypto.PublicKey import RSA, ECC, DSA
```

```
hashes = {
    '1': SHA256.new,
    '2': SHA384.new,
    '3': SHA512.new
}
```

```
def import_rsa_v1_5_key(a):
    with open('RSA v1.5/private_rsa_v1_5.pem', 'r') as f:
        key = RSA.import_key(f.read())
    return key
```

```
def generate_rsa_v1_5_key(a=""):
    key = RSA.generate(2048)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the key on your device?\n"
                  "(Warning! this will erase the previous key of that type in the keys folder)\n"
                  "press y to save / n to skip\n")
    if a == 'y':
        with open('RSA v1.5/private_rsa_v1_5.pem', 'wb') as f:
            f.write(key.export_key('PEM'))
        with open('RSA v1.5/public_rsa_v1_5.pem', 'wb') as f:
            f.write(key.publickey().export_key('PEM'))
    return key
```

```

def import_rsa_pss_key(a):
    with open('RSA PSS/private_rsa_pss.pem', 'r') as f:
        key = RSA.import_key(f.read())
    return key

def generate_rsa_pss_key(a=""):
    key = RSA.generate(2048)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the key on your device?\n"
                  "(Warning! this will erase the previous key of that type in the keys folder)\n"
                  "press y to save / n to skip\n")
    if a == 'y':
        with open('RSA PSS/private_rsa_pss.pem', 'wb') as f:
            f.write(key.export_key('PEM'))
        with open('RSA PSS/public_rsa_pss.pem', 'wb') as f:
            f.write(key.publickey().export_key('PEM'))
    return key

def import_eddsa_key(a):
    with open('EdDSA/private_eddsa.pem', 'rt') as f:
        key = ECC.import_key(f.read())
    return key

def generate_eddsa_key(a=""):
    key = ECC.generate(curve='ed25519')
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the key on your device?\n")

```



```

        "(Warning! this will erase the previous key of that type in the keys folder)\n"
        "press y to save / n to skip\n")
    if a == 'y':
        with open('EdDSA/private_eddsa.pem', 'wt') as f:
            f.write(key.export_key(format='PEM'))
        with open('EdDSA/public_eddsa.pem', 'wt') as f:
            f.write(key.public_key().export_key(format='PEM'))
    return key

def import_pure_eddsa_key(a):
    with open('PureEdDSA/private_pure_eddsa.pem', 'rt') as f:
        key = ECC.import_key(f.read())
    return key

def generate_pure_eddsa_key(a=""):
    key = ECC.generate(curve='ed25519')
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the key on your device?\n"
            "(Warning! this will erase the previous key of that type in the keys folder)\n"
            "press y to save / n to skip\n")
    if a == 'y':
        with open('PureEdDSA/private_pure_eddsa.pem', 'wt') as f:
            f.write(key.export_key(format='PEM'))
        with open('PureEdDSA/public_pure_eddsa.pem', 'wt') as f:
            f.write(key.public_key().export_key(format='PEM'))
    return key

def import_dsa_key(a):
    with open('DSA/private_dsa.pem', 'rt') as f:

```

```

    key = DSA.import_key(f.read())
    return key

def generate_dsa_key(a=""):
    key = DSA.generate(2048)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the key on your device?\n"
                  "(Warning! this will erase the previous key of that type in the keys folder)\n"
                  "press y to save / n to skip\n")
    if a == 'y':
        with open('DSA/private_dsa.pem', 'wb') as f:
            f.write(key.export_key('PEM'))
        with open('DSA/public_dsa.pem', 'wb') as f:
            f.write(key.publickey().export_key('PEM'))
    return key

def import_ecdsa_key(a):
    with open('ECDSA/private_ecdsa.pem', 'rt') as f:
        key = ECC.import_key(f.read())
    return key

def generate_ecdsa_key(a=""):
    key = ECC.generate(curve='P-521')
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the key on your device?\n"
                  "(Warning! this will erase the previous key of that type in the keys folder)\n"
                  "press y to save / n to skip\n")

```

```

if a == 'y':
    with open('ECDSA/private_ecdsa.pem', 'wt') as f:
        f.write(key.export_key(format='PEM'))
    with open('ECDSA/public_ecdsa.pem', 'wt') as f:
        f.write(key.public_key().export_key(format='PEM'))
return key

def sign_rsa_v1_5(hash_chosen, key_source, message=b'To be signed', a="): # signer for Rsa
PKCS#1 v1.5
    keys = {
        'generate': generate_rsa_v1_5_key,
        'import': import_rsa_v1_5_key
    }
    key = keys[key_source](a)
    h = hashes[hash_chosen](message)
    signature = pkcs1_15.new(key).sign(h)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the signature on your device?\n"
            "(Warning! this will erase the previous signature of that type in the signatures
folder)\n"
            "press y to save / n to skip\n")
    if a == 'y':
        with open('Signatures/signature_v1_5.txt', 'wb') as f:
            f.write(signature)
    return key, signature

def verify_rsa_v1_5(hash_chosen, key=None, signature=None, message=b'To be signed', a = ""):
    try:
        h = hashes[hash_chosen](message)
        if key is None and signature is None:

```

```

        with open('RSA v1.5/public_rsa_v1_5.pem', 'r') as f:
            key = RSA.import_key(f.read())
        with open('Signatures/signature_v1_5.txt', 'rb') as f:
            signature = f.read()
        pkcs1_15.new(key).verify(h, signature)
        if a == "":
            print("The signature is valid.")
    except (ValueError, TypeError):
        print("The signature is not valid.")

def sign_rsa_pss(hash_chosen, key_source, message=b'To be signed', a = ""): # signer for Rsa
    PKCS#1 PSS
    keys = {
        'generate': generate_rsa_pss_key,
        'import': import_rsa_pss_key
    }
    key = keys[key_source](a)
    h = hashes[hash_chosen](message)
    signature = pss.new(key).sign(h)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the signature on your device?\n"
            "(Warning! this will erase the previous signature of that type in the signatures
            folder)\n"
            "press y to save / n to skip\n")
    if a == 'y':
        with open('Signatures/signature_pss.txt', 'wb') as f:
            f.write(signature)
    return key, signature

def verify_rsa_pss(hash_chosen, key = None, signature = None, message=b'To be signed', a = ""):

```

```

try:
    h = hashes[hash_chosen](message)
    if key is None and signature is None:
        with open('RSA PSS/public_rsa_pss.pem', 'r') as f:
            key = RSA.import_key(f.read())
        with open('Signatures/signature_pss.txt', 'rb') as f:
            signature = f.read()
    pss.new(key).verify(h, signature)
    if a == "":
        print("The signature is valid.")
except (ValueError, TypeError):
    print("The signature is not valid.")

```

```

def sign_eddsa(hash_chosen, key_source, message=b'To be signed', a = ""): # signer for eddsa
    keys = {
        'generate': generate_eddsa_key,
        'import': import_eddsa_key
    }
    key = keys[key_source](a)
    h = SHA512.new(message)
    signature = eddsa.new(key, 'rfc8032').sign(h)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the signature on your device?\n"
            "(Warning! this will erase the previous signature of that type in the signatures"
            "folder)\n"
            "press y to save / n to skip\n")
    if a == 'y':
        with open('Signatures/signature_eddsa.txt', 'wb') as f:
            f.write(signature)

    return key, signature

```

```

def verify_eddsa(hash_chosen, key = None, signature = None, message=b'To be signed', a = ""):
    try:
        h = SHA512.new(message)
        if key is None and signature is None:
            with open('EdDSA/public_eddsa.pem', 'r') as f:
                key = ECC.import_key(f.read())
            with open('Signatures/signature_eddsa.txt', 'rb') as f:
                signature = f.read()
            eddsa.new(key, 'rfc8032').verify(h, signature)
        if a == "":
            print("The signature is valid.")
    except (ValueError, TypeError):
        print("The signature is not valid.")

```

```

def sign_pure_eddsa(hash_chosen, key_source, message=b'To be signed', a = ""):
    keys = {
        'generate': generate_pure_eddsa_key,
        'import': import_pure_eddsa_key
    }
    key = keys[key_source](a)
    signature = eddsa.new(key, 'rfc8032').sign(message)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the signature on your device?\n"
            "(Warning! this will erase the previous signature of that type in the signatures"
            "folder)\n"
            "press y to save / n to skip\n")
    if a == 'y':
        with open('Signatures/signature_pure_eddsa.txt', 'wb') as f:
            f.write(signature)

```

```
return key, signature
```

```
def verify_pure_eddsa(hash_chosen, key = None, signature = None, message=b'To be signed', a
= "):
    try:
        if key is None and signature is None:
            with open('PureEdDSA/public_pure_eddsa.pem', 'r') as f:
                key = ECC.import_key(f.read())
            with open('Signatures/signature_pure_eddsa.txt', 'rb') as f:
                signature = f.read()
            eddsa.new(key, 'rfc8032').verify(message, signature)
        if a == "":
            print("The signature is valid.")
    except (ValueError, TypeError):
        print("The signature is not valid.")
```

```
def sign_dsa(hash_chosen, key_source, message=b'To be signed', a = ""): # signer for dsa
    keys = {
        'generate': generate_dsa_key,
        'import': import_dsa_key
    }
    key = keys[key_source](a)
    h = hashes[hash_chosen](message)
    signature = DSS.new(key, 'fips-186-3').sign(h)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the signature on your device?\n"
            "(Warning! this will erase the previous signature of that type in the signatures
            folder)\n"
            "press y to save / n to skip\n")
```

```

if a == 'y':
    with open('Signatures/signature_dsa.txt', 'wb') as f:
        f.write(signature)
    return key, signature

def verify_dsa(hash_chosen, key=None, signature=None, message=b'To be signed', a = ""):
    try:
        h = hashes[hash_chosen](message)
        if key is None and signature is None:
            with open('DSA/public_dsa.pem', 'r') as f:
                key = DSA.import_key(f.read())
            with open('Signatures/signature_dsa.txt', 'rb') as f:
                signature = f.read()
            DSS.new(key, 'fips-186-3').verify(h, signature)
        if a == "":
            print("The signature is valid.")
        except (ValueError, TypeError):
            print("The signature is not valid.")

def sign_ecdsa(hash_chosen, key_source, message=b'To be signed', a = ""): # signer for eddsa
    keys = {
        'generate': generate_ecdsa_key,
        'import': import_ecdsa_key
    }
    key = keys[key_source](a)
    h = hashes[hash_chosen](message)
    signature = DSS.new(key, 'fips-186-3').sign(h)
    if a != "":
        a = 'y'
    else:
        a = input("Would you like to save the signature on your device?\n")

```



```
        "(Warning! this will erase the previous signature of that type in the signatures
folder)\n"
```

```
        "press y to save / n to skip\n")
```

```
if a == 'y':
```

```
    with open('Signatures/signature_ecdsa.txt', 'wb') as f:
```

```
        f.write(signature)
```

```
    return key, signature
```

```
def verify_ecdsa(hash_chosen, key=None, signature=None, message=b'To be signed', a = ""):
```

```
    try:
```

```
        h = hashes[hash_chosen](message)
```

```
        if key is None and signature is None:
```

```
            with open('ECDSA/public_ecdsa.pem', 'r') as f:
```

```
                key = ECC.import_key(f.read())
```

```
            with open('Signatures/signature_ecdsa.txt', 'rb') as f:
```

```
                signature = f.read()
```

```
            DSS.new(key, 'fips-186-3').verify(h, signature)
```

```
            if a == ":
```

```
                print("The signature is valid.")
```

```
    except (ValueError, TypeError) as e:
```

```
        print("The signature is not valid.", e)
```

```
if __name__ == "__main__":
```

```
    a = "0"
```

```
    match a:
```

```
        case "0":
```

```
            key, signature = sign_rsa_v1_5('1', 'import') # import
```

```
            verify_rsa_v1_5('1') # verify_pkcs1('1', signature, key)
```

```
        case "1":
```

```
            key, signature = sign_rsa_pss('1', 'generate')
```

```
            verify_rsa_pss('1', key, signature)
```

```
        case "2":
```

```
key, signature = sign_eddsa('null', 'generate') # no hash selection
verify_eddsa('null', key, signature)
case "3":
    key, signature = sign_dsa('1', 'generate')
    verify_dsa('1', key, signature)
case "4":
    key, signature = sign_ecdsa('2', 'generate')
    verify_ecdsa('2', key, signature)
case "5":
    key, signature = sign_pure_eddsa('null', 'generate') # no hash selection
    verify_pure_eddsa('null', key, signature)
```