МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра математического моделирования и анализа данных

Дипломная работа

Криптография на основе функций хэширования: подписи без состояния

Болтач Антон Юрьевич
Студент 4 курса 9 группы
Научный руководитель
С. В. Агиевич

Минск, 2020 г.

Содержание

Введение

Цифровые подписи широко используются в Интернете, в частности, для аутентификации, проверки целостности и отказа от авторства. Алгоритмы цифровой подписи, наиболее часто используемые на практике — RSA, DSA и ECDSA, основаны на допущениях сложности задач теории чисел, а именно факторизации составного целого числа и вычислении дискретных логарифмов. В 1994 году Питер Шор показал, что эти вычислительные задачи с числами могут стать решаемыми при наличии квантовых компьютеров. Квантовые компьютеры могут решить их за полиномиальное время, ставя под угрозу безопасность схем цифровой подписи, используемых сегодня. Хотя квантовые компьютеры еще не доступны, их развитие происходит быстрыми темпами и поэтому представляет собой реальную угрозу в течение следующих десятилетий. К счастью, постквантовая криптография предоставляет множество квантовостойких альтернатив классическим схемам цифровой подписи.

Подписи на основе функций хэширования, как они также известны, являются одной из наиболее многообещающих из этих альтернатив. Мы говорим, о сравнительно новой криптографической платформе HBC (Hash-Based Cryptography). Это одна из популярных платформ конкурса NIST PostQuatum Crypto. В первом раунде было более 70-и заявок алгоритмов из класса HBC, из них 27 прошли во второй раунд.

1 Подписи на основе функций хэширования

1.1 Классификация

Формально схема цифровой подписи представляет собой тройку алгоритмов (Gen, Sign, Verify):

- Алгоритм генерации ключей Gen:
  - Вход: 1^n, где n — параметр безопасности.
  - Выход: Открытый ключ (pk) и соответствующий личный ключ (sk).

  Алгоритм Gen необходим для целостности цифровой подписи, поскольку мы получаем личный ключ и соответствующий открытый ключ.

- Алгоритм подписи Sign:
  - Вход: Сообщение m и личный ключ sk.
  - Выход: Подпись σ.

  Алгоритм Sign создает подпись на основе сообщения и личного ключа.

- Алгоритм проверки Verify:
  - Вход: Открытый ключ pk, сообщение m и подпись σ.
  - Выход: Проверка успешна (true) или отклонена (false).

  Алгоритм проверки проверяет подлинность подписи, на основе сообщения, открытого ключа и цифровой подписи.

В основном подписи на основе функций хэширования подразделяются на:

1. Подписи с состоянием Stateful:

   Подпись с состоянием означает, что алгоритм Gen, кроме sk (личного ключа) возвращает st (состояние), и st является и дополнительным входом, и дополнительным выходом алгоритма Sign.

   (a) Одноразовые подписи OTS (One-Time Signature).

   (b) Многоразовые подписи FTS (Few-Time Signature).

2. Деревья Меркля MSS (Merkle Signature Scheme).

3. Подписи без состояния Stateless.

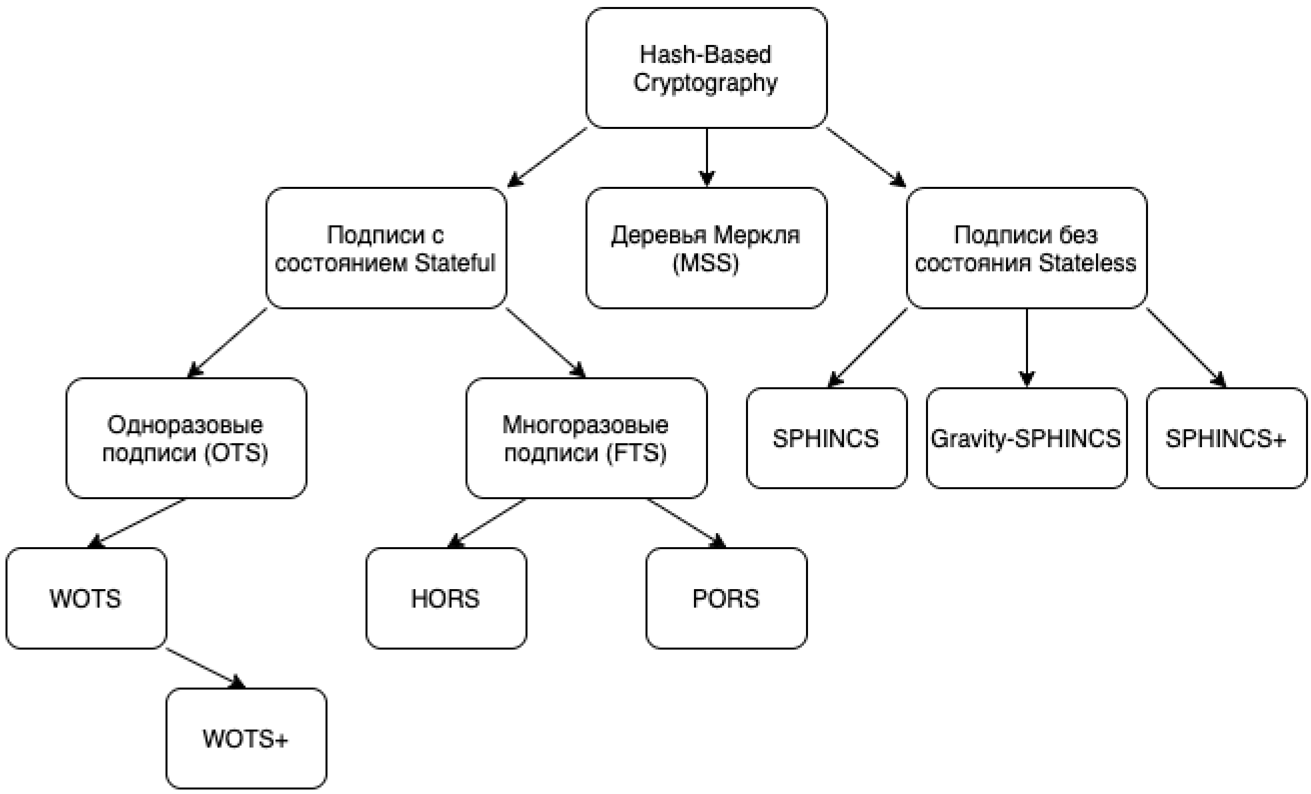

Рис. 1: Схема Hash-Based Cryptography

1.2 Одноразовые подписи

Одноразовые подписи OTS (One-Time Signature) называются одноразовыми, поскольку безопасность сообщения гарантируется только при однократном использовании. Однако, преимущества OTS заключаются в том, что они могут быть построены из любой односторонней функции, а алгоритмы подписи и проверки очень быстры и дёшевы в вычислении (по сравнению с другими подписями на основе функций хэширования).

1.2.1 Одноразовая подпись Винтерница

WOTS (Winternitz One-Time Signature) [2] использует хэш функцию $F : \{0,1\}^n \to \{0,1\}^n$. Она параметризуется длиной сообщения m и параметром $Winternitz$, $w \in N$, $w > 1$, который определяет компромисс между временем и памятью. Эти два параметра используются для вычисления

$$l_1 = \left\lceil \frac{m}{log(w)} \right\rceil, l_2 = \left\lfloor \frac{log(l_1(w-1))}{log(w)} \right\rfloor + 1, l = l_1 + l_2.$$

Схема использует $w-1$ итераций F на случайном входе. Мы определяем их как

$$F^a(x) = F(F^{a-1}(x))$$

и $F^0(x) = x$.

Теперь опишем три алгоритма подписи:

- **Алгоритм генерации ключей Gen:**

  Алгоритм генерации ключей выбирает l (n-битовых блоков) равномерно, случайным образом. Личный ключ $sk = (sk_1, ..., sk_l)$ состоит из этих l блоков случайных битовых строк. Открытый ключ проверки pk вычисляется как

  $$pk = (pk_1, ..., pk_l) = (F^{w-1}(sk_1), ..., F^{w-1}(sk_l))$$

- **Алгоритм подписи Sign:**

  Сообщение M^* длины m и личного ключа подписи sk, алгоритм подписи сначала вычисляет базовое w представление

  $$M^* : M^* = (M_1^*, ..., M_{l_1}^*), M_i^* \in \{0, ..., w-1\}$$

  и вычисляет его базовое w представление $C = (C_1, ..., C_{l_2})$. Длина базового w представления C не более l_2, так как $C \le l_1(w-1)$. Мы задаем $B = (B_1, ..., B_l) = M^*||C$. Подпись вычисляется как

  $$\sigma = (\sigma_1, ..., \sigma_l) = (F^{B_1}(sk_1), ..., F^{B_l}(sk_l))$$

- **Алгоритм проверки Verify:**

  Сообщение M^* длины m, подпись σ и открытый ключ проверки pk, алгоритм проверки сначала вычисляет B_i, $1 \leq i \leq l$, как описано выше. Затем он выполняет следующее сравнение:

$$pk = (pk_1, ..., pk_l) \stackrel{?}{=} (F^{w-1-B_1}(\sigma_1), ..., (F^{w-1-B_l}(\sigma_l))$$

  Если сравнение выполняется, оно возвращает $true$ или $false$ в противном случае.

Схема подписи без контрольной суммы может быть просто нарушена: после того, как подписавшая сторона выдаст действительную подпись для какого-либо сообщения, любой может легко создать подпись для сообщения $M^{'} = (M_1^{'}, ..., M_{l_1}^{'})$, если $M_i^{'} \leq M_i^*$ для всех i.

Идея исправления проблемы заключается в следующем, помимо подписания сообщения, мы так же должны подписать некоторую дополнительную строку, которую назовем контрольной суммой и вычислим следующем образом:

$$C = \sum_{i=1}^{l_1} (w - 1 - M_i^*)$$

1.2.2 Дополненная подпись Винтерница

Введем вариант подписи Винтерница WOTS$^+$ (Winternitz One-Time Signature$^+$) [4], который позволяет уменьшить размер подписи и достигает более высокого уровня безопасности. Как и все варианты $WOTS$, $WOTS^+$ параметризуется параметром безопасности $n \in N$, длиной сообщения m и параметром $w \in N$, $w > 1$, который определяет компромисс между временем и памятью. Последние два параметра используются для вычисления

$$l_1 = \left\lceil \frac{m}{log(w)} \right\rceil, l_2 = \left\lfloor \frac{log(l_1(w-1))}{log(w)} \right\rfloor + 1, l = l_1 + l_2.$$

Кроме того, $WOTS^+$ использует семейство функций $F_n : \{f_k : \{0,1\}^n \rightarrow \{0,1\}^n | k \in K_n\}$ с ключевым пространством K_n. Можно предположить как о криптографическом семействе хэш-функций, которое не сжимается. Используя F_n, мы определяем следующую функцию.

$c_k^i(x, r)$: На входе значения $x \in \{0,1\}^n$, счетчика итераций $i \in N$, ключа $k \in K$ и элементы случайности $r = (r_1, ..., r_j) \in \{0,1\}^{n \times j}$ при $j \geq i$, функция работает следующим образом:

- В случае $i = 0$, $c_k^i(x, r)$ возвращает $x(c_k^0(x, r) = x)$.

- Для $i > 0$ мы определяем $c_k^i(x, r)$ рекурсивно как

$$c_k^i(x, r) = f_k(c_k^{i-1}(x, r) \oplus r_i),$$

То есть в каждом раунде функция сначала принимает побитовый xor промежуточного значения и битовую маску r, затем оценивает f_k на результат. Мы пишем $r_{a,b}$ для подмножества $r_a, ..., r_b$ как r. В случае $b < a$ мы определяем $r_{a,b}$ как пустую строку. Будем считать, что параметры m, w и семейство функций F_n общеизвестны.

Теперь опишем три алгоритма подписи $WOTS^+$:

- **Алгоритм генерации ключа Gen:**

  При вводе параметра безопасности n унарно, алгоритм генерации ключа выбирает $l+w-1$ n-бит строки равномерно случайным образом. Личный ключ $sk = (sk_1, ..., sk_l)$ состоит из первых l случайных битовых строк. Оставшиеся $w-1$ бит строки используются в качестве элементов случайности $r = (r_1, ..., r_{w-1})$ для c. Далее, Kg выбирает функцию ключа $k \stackrel{\$}{\leftarrow} K$ равномерно случайным образом. Открытый ключ проверки pk вычисляется как

  $$pk = (pk_0, pk_1, ..., pk_l) = ((r, k), c_k^{w-1}(sk_1, r), ..., c_k^{w-1}(sk_l, r)).$$

- **Алгоритм подписи Sign:**

  На входе m битного сообщения M, личного ключа подписи sk и элементов случайности r, алгоритм подписи сначала вычисляет базовое w представление $M : M = (M_1...M_{l_1})$, $M_i \in \{0, ..., w-1\}$. Поэтому M рассматривается как двоичное представление натурального числа x, а затем вычисляется w бинарное представление x. Далее вычисляем контрольную сумму

  $$C = \sum_{i=1}^{l_1}(w - 1 - M_i)$$

  и его базовое w представление $C = (C_1, ..., C_{l_2})$. Длина базового w представления C не более l_2, так как $C \leq l_1(w-1)$. Мы задаем $B = (b_1, ..., b_l) = M||C$, конкатенация базовых w представлений M и C. Подпись вычисляется как

  $$\sigma = (\sigma_1, ..., \sigma_l) = (c_k^{b_1}(sk_1, r), ..., c_k^{b_l}(sk_l, r)).$$

  Обратите внимание, что контрольная сумма гарантирует, что с учетом $b_i, 0 < i \leq l$, соответствующего одному сообщению, b_i^* соответствующий любому другому сообщению включает по крайней мере один $b_i^* < b_i$.

- **Алгоритм проверки Verify:**

  На входе сообщение M двоичной длины m, подпись σ и открытый ключ pk. Алгоритм проверки сначала вычисляет $b_i, 1 \leq i \leq l$, как описано выше. Затем он выполняет следующее сравнение:

  $$pk = (pk_0, pk_1, ..., pk_l) \stackrel{?}{=} ((r, k), c_k^{w-1-b_1}(\sigma_1, r_{b_1+1,w-1}, ..., c_k^{w-1-b_l}(\sigma_l, r_{b_l+1,w-1}))$$

Если сравнение выполняется, оно возвращает $true$ или $false$ в противном случае.

Время выполнения всех трех алгоритмов ограничено l и w оценками f_k. Размер подписи и личного ключа составляет $|\sigma| = |sk| = l * n$ бит. Размер открытого ключа равен $(l+w-1)n+|k|$ бит, где $|k|$ обозначает количество бит, необходимых для представления любого элемента K.

1.3 Деревья Меркля

Первый способ создать схему многократной подписи MSS (Merkle Signature Scheme) из схемы одноразовой подписи — использовать конструкцию, предложенную Мерклом в 1989 году. Учитывая целые числа n, h и хэш-функцию $H : \{0,1\}^{2n} \to \{0,1\}^n$, так называемое дерево Меркля представляет собой двоичное дерево высоты h, узлы которого помечены значением $x \in \{0,1\}^n$, таким образом, что значение каждого внутреннего узла вычисляется как $x = H(y||z)$, где y и z — значения левых и правых дочерних элементов.

Эта конструкция позволяет превратить схему одноразовой подписи в схему многократной подписи следующим образом. Учитывая 2^h экземпляров OTS, подписывающий создает дерево Меркля, каждое листовое значение которого являются открытым ключом экземпляра OTS. Общий открытый ключ — это корневое значение. i-я подпись содержит подпись, сгенерированную i-м экземпляром OTS, а также путь аутентификации i (см. Рис. 2).

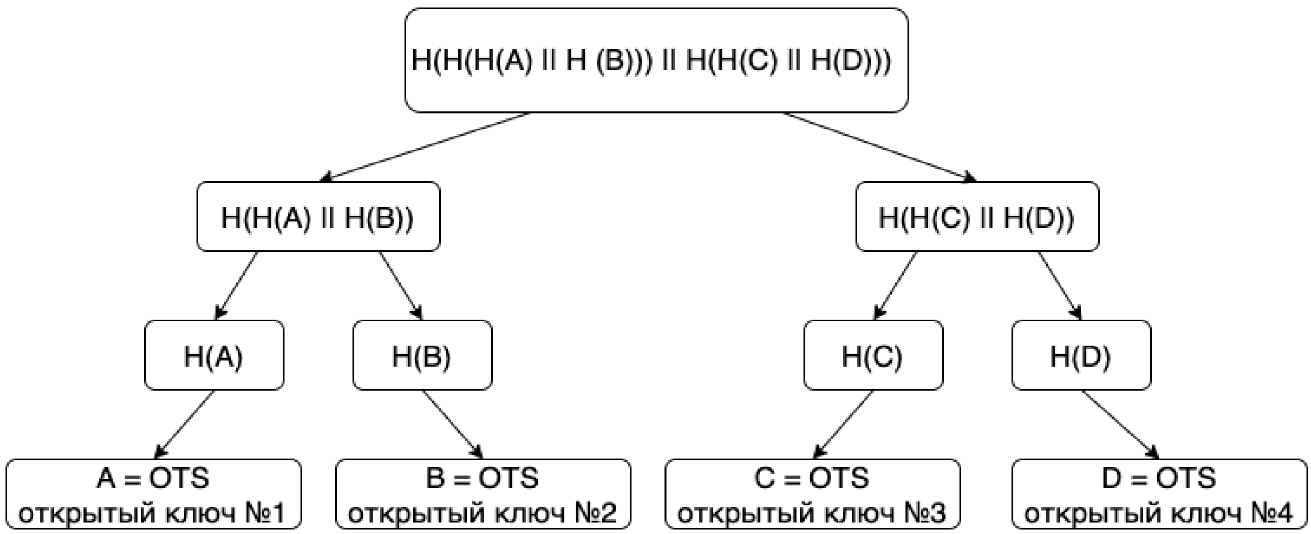

Рис. 2: Пример MSS.

Исходя из примера, чтобы подписать сообщение: используем первый публичный ключ OTS (A), а затем никогда больше не используем его, чтобы не нарушить безопасность. Затем используем открытый ключ OTS (B), затем OTS (C), и, наконец, OTS (D). Таким образом, мы можем подписать 4 сообщения в общей сложности с помощью нашего дерева. Следовательно, более большое дерево позволит нам подписывать больше сообщений.

Привлекательной идеей здесь является то, что открытый ключ состоит только из корня дерева, и каждый раз, когда мы подписываем сообщение, наша подпись состоит только из нескольких хэшей и назовем это путь аутентификации (см. Рис. 3).

В нашем примере подпись с первым ключом OTS (A) была бы набором элементов (1, σ, pk (A), $auth$):

- 1 — это индекс листа подписи. Мы должны иметь это в виду: мы не можем повторно использовать OTS этого листа. Это делает нашу схему статичной.

- σ — это подпись OTS.

- pk (A) — это открытый ключ OTS.

- $auth$ — это список узлов (так называемый список хэшей), который позволяет нам вычислить корень (наш основной открытый ключ).

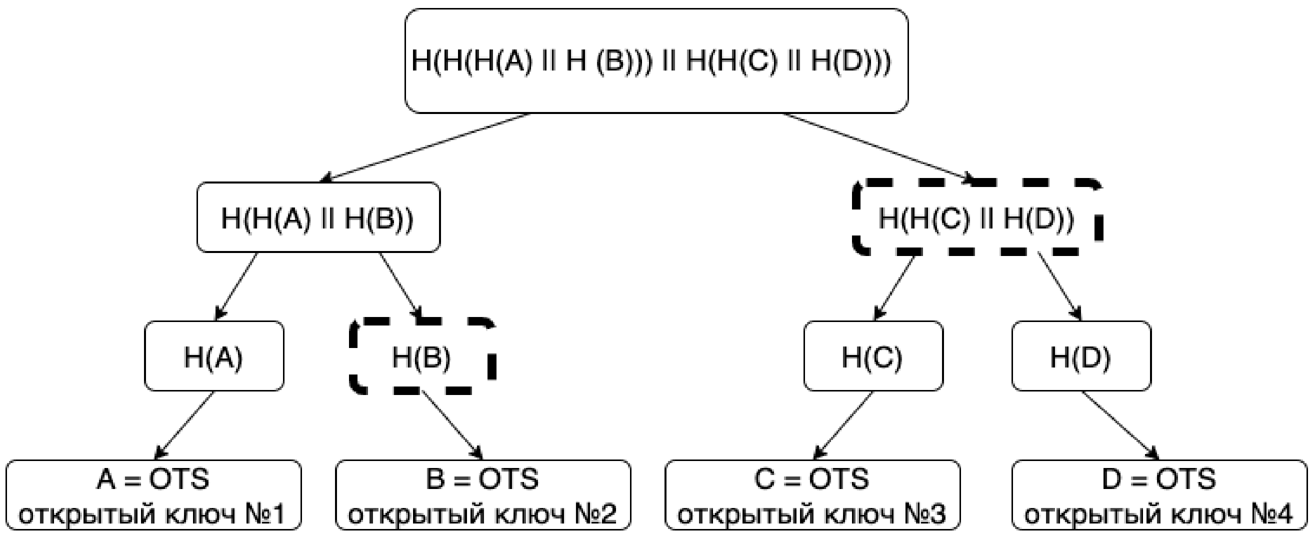

Рис. 3: Пример MSS с выделенными блоками пути аутентификации.

Мы видим, что с помощью нашего открытого ключа OTS и двух наших хэшей (соседние узлы всех узлов на пути от нашего подписывающего листа до корня) мы можем вычислить основной открытый ключ. И таким образом мы можем проверить, что это действительно была подпись, которая была получена от этого основного публичного ключа.

Благодаря этой схеме мы не знаем всех открытых ключей OTS для проверки основного открытого ключа. Следовательно, схема MSS экономит память и вычисления.

Время генерации ключа экспоненциально h, потому что на этом этапе необходимо вычислить полное дерево Меркля. Например, $h = 20$ возможно, но может быть недостаточно для всех подписывающих. Кроме того, подписывающий должен отслеживать индексы i, которые уже были использованы, поэтому схема является $stateful$.

1.4 Многоразовые подписи

В то время как одноразовые подписи обеспечивают удовлетворительную криптографическую безопасность для подписания и проверки транзакций, для них характерен существенный недостаток — их можно использовать безопасно только один раз. Поэтому существуют схемы подписи FTS (Few-Time Signature), позволяющей подписывать несколько раз одним и тем же личным ключем.

1.4.1 HORS

HORS (Hash to Obtain Random Subset) — это многоразовая схема подписи. Пусть f — односторонняя функция, а H — хэш-функция, которая выводит случайный размер подмножества $\{1, 2, ..., t\}$, где k и t - параметры, влияющие на безопасность с помощью $k < t$. Параметры k и t также влияют на то, что длина моего личного ключа (и, следовательно, открытого ключа) будет t, в то время как мои подписи будут длиной t. Ключ подписи — это случайный кортеж $(s_1, ..., s_t)$, а открытым ключом является $(f(s_1), ..., f(s_t))$.

Теперь опишем три алгоритма подписи HORS:

- **Алгоритм генерации ключа Gen:**
  - Вход: Параметры l, k, t.
    Шаги:
    1. Генерируем t случайных l-битовых строк $s_1, s_2, ..., s_t$.
    2. Пусть $v_i = f(s_i)$ для $1 \leq i \leq t$.
  - Выход: Открытый ключ $PK = (k, v_1, v_2, ..., v_t)$ и личный ключ $SK = (k, s_1, s_2, ..., s_t)$.

- **Алгоритм подписи Sign:**
  - Вход: Сообщение m и личный ключ $SK = (k, s_1, s_2, ..., s_t)$.
    Шаги:
    1. Пусть $h = Hash(m)$
    2. Разбиваем h на k подстрок $h_1, h_2, ..., h_k$ длины $log_2 t$ бит каждый.
    3. Представим каждое h_j как целое i_j для $1 \leq j \leq k$.
  - Выход: Подпись $\sigma = (s_{i_1}, s_{i_2}, ..., s_{i_k})$.

- **Алгоритм проверки Verify:**
  - Вход: Сообщение m, подпись $\sigma = (s_1^{'}, s_2^{'}, ..., s_k^{'})$ и открытый ключ $PK = (k, v_1, v_2, ..., v_t)$.
    Шаги:

1. Пусть $h = Hash(m)$
2. Разбиваем h на k подстрок $h_1, h_2, ..., h_k$ длины $log_2 t$ бит каждый.
3. Представим каждое h_j как целое i_j для $1 \le j \le k$.

– Выход: Успешно, если для каждого j, $1 \le j \le k$, $f(s_j^{'}) = v^{i_j}$; Отклонено, в ином случае.

1.4.2 PORS

PORS (PRNG to Obtain Random Subset, где PRNG генератор псевдослучайных чисел), где используется PRNG для получения случайного подмножества. Алгоритмы генерации ключей, подписи и проверки аналогичны HORS, но в отличии от HORS, где используется хэш-функция, мы используем PRNG для сообщения и запрашиваем его до тех пор, пока не получим k различных индексов. Расходы в этом случае на вычисления минимальны, но значительно повышает безопасность.

2 Подписи без состояния

2.1 SPHINCS

SPHINCS [5] - подпись, сочетающая в себе большое количество достижений в области HBC (Hash-Based Cryptography). Данная подпись обеспечивает нам самое главное избавление от состояния. Это означает, что нам больше не нужно сохранять и обновлять состояние подписи и поэтому SPHINCS называется Stateless (без состояния) подписью. Представим основные идеи SPHINCS, описав его как комбинацию четырех типов деревьев. Ниже перечислены четыре типа деревьев (см. Рис. 4):

1. Главное Гипердерево, высотой h (60 в SPHINCS-256). Корень этого дерева является частью открытого ключа. Листья этого дерева экземпляры HORST. Это Гипердерево делится на d слоев($d = 12$ в SPHINCS-256).

2. Поддеревья, которые являются деревьями Меркля высоты h/d ($60/12 = 5$ в SPHINCS-256). Листья этих деревьев являются корнями деревьев; указанные корни являются сжатыми открытыми ключами экземпляров WOTS, которые соединяются с деревом на следующем уровне.

3. Открытый ключ WOTS это деревья сжатия, которые являются L-деревьями, высоты $\lceil log_2 l \rceil$.Листья этого дерева являются компонентами WOTS открытого ключа (67 значений по 256 бит каждое в SPHINCS-256). Связанный экземпляр WOTS подписывает корень дерева на следующем уровне.

4. В нижней части гипердерева, открытый ключ HORST — деревья сжатия это деревья Меркля высоты $\tau = log_2 t$, где t номер элементов открытого ключа HORST(2^{16} в SPHINCS-256).

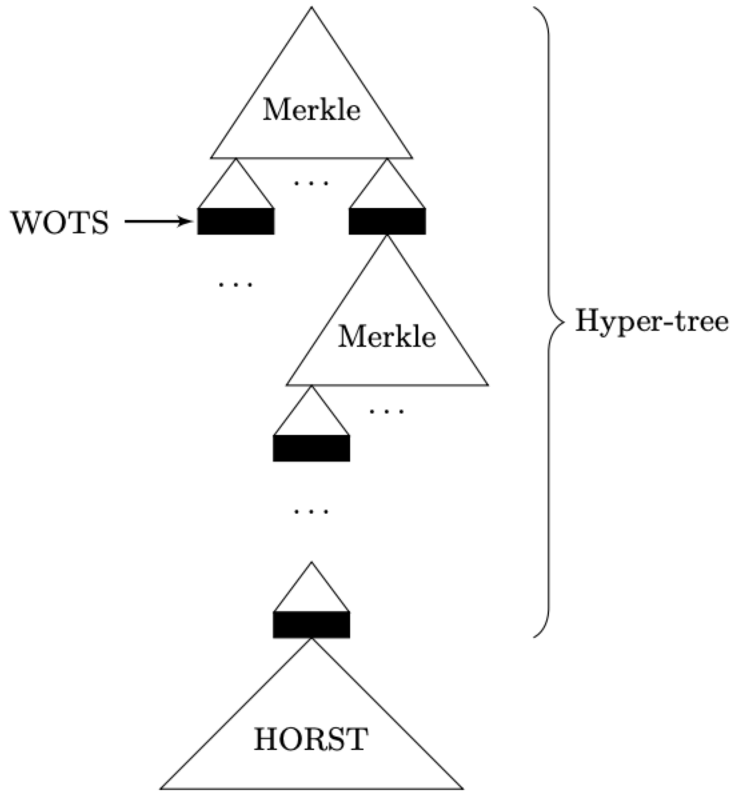

Рис. 4: Пример SPHINCS. Гипердерево состоит из d слоев дерева Меркля и соединены WOTS. Внизу дерево HORS(или HORST) соединяется с подписанным сообщением.

Алгоритм Sign в SPHINCS формально можно описать:

1. Извлекается листовой индекс из сообщения и личного ключа. Этот индекс определяет один из экземпляров 2^h HORST (относительно основного гипердерева), который будет использоваться для подписи сообщения.

2. Создайте экземпляр HORST, который является производным от личного ключа и конечного индекса, и подпишите сообщение этим экземпляром HORST. Подпись HORST включает k ключей и их соответствующие пути аутентификации и является частью подписи SPHINCS. Итого получаем сжатый в дереве HORST открытый ключ p.

3. Для каждого слоя гипердерева подпишите открытый ключ p(полученный из нижнего слоя), используя правильный экземпляр WOTS(полученный из листового индекса); добавьте эту подпись WOTS и связанный с ней путь аутентификации к подписи SPHINCS. Вычислите путь аутентификации этого экземпляра WOTS в поддереве. Добавьте этот путь к подписи SPHINCS и p-корень поддерева.

2.2 Gravity-SPHINCS

Gravity-SPHINCS [5], схема подписи на основе SPHINCS с более короткими ключами (32 и 64 байта вместо $\approx$ 1 КБ), более короткими подписями ($\approx$ 30 КБ вместо 41 КБ) и более быстрыми алгоритмами Sign и Verify. Gravity-SPHINCS наследует некоторые параметры от SPHINCS (длина хэша, глубина WOTS и др.), а так же имеет новые. В приведенном ниже списке h обозначает высоту поддеревьев(в отличие от высоты основного дерева в SPHINCS), а $B_n = \{0, 1\}^n$ обозначает набор n-битовых строк.

Параметры являются следующими:

- Хэш-выход длина бита n, положительное целое число.
- Глубина WOTS w, степень 2-ки такой, что $w \geq 2$ и $log_2 w$ делит n.
- Размер множества PORS t, положительное, степень двойки.
- Размер подмножества PORS k, положительное целое такое, что $k \leq t$.
- Высота дерева Меркля h, положительное целое.
- Количество внутренних деревьев Меркля d, неотрицательное целое.
- Высота кэша c, неотрицательное целое.
- Высота b, неотрицательное целое.
- Пространство сообщения M, обычно подмножество битовых строк $\{0, 1\}^*$.

Из этих параметров получены:

- Размер WOTS $l = \mu + \lfloor log_2(\mu(w-1))/log_2 w \rfloor + 1$, где $\mu = n/log_2 w$.
- Множество PORS, $T = \{0, ..., t-1\}$.
- Адресное пространство $A = \{0, ..., d\} \times \{0, ..., 2^{c+dh} - 1\} \times \{0, ..., max(l, t) - 1\}$.
- Пространство открытых ключей $PK = B_n$.
- Пространство личных ключей $SK = B_n^2$.
- Пространство подписи $SG = B_n \times B_n^k \times B_n^{\leq k(log_2 t - \lfloor log_2 k \rfloor)} \times (B_n^l \times B_n^h)^d \times B_n^c$.
- $SG_B = B_n^b \times \{0, ..., 2^b - 1\} \times SG$
- Размер открытого ключа n бит.
- Размер личного ключа, $2n$ бит.

- Максимальный размер подписи

$$sigsz = (1 + k + k(log_2 t - \lfloor log_2 k \rfloor) + d(l + h) + c)n$$

Алгоритм подписи $Sign$ одного сообщения и проверка $Verify$ в Gravity-SPHINCS очень похожа на SPHINCS.

Опишем три алгоритма подписи Gravity-SPHINCS:

- **Алгоритм генерации ключей Gen:**

  KG получает на вход $2n$ случайных бит и на выходе получаем личный ключ $sk \in B_n^2$, и открытый ключ $pk \in B_n$.

  1. Генерация личного ключа из $2n$ случайных бит:

  $$sk = (seed, salt) \xleftarrow{\$} B_n^2$$

  2. Для $0 \leq i < 2^{c+h}$ генерируется $Winternitz$ открытый ключ:

  $$x_i \leftarrow WOTS, \text{используя } genpk(seed, make - addr(0, i))$$

  3. Генерация открытого ключа:

  $$pk \leftarrow Merkle, \text{используя } root_{c+h}(x_0, ..., x_{2^{c+h}-1})$$

- **Алгоритм подписи Sign:**

  S на вход принимает хэш $m \in B_n$ и личный ключ $sk = (seed, salt)$, и на выходе получаем подпись.

  1. Вычисляем $s \leftarrow H(salt, m)$.
  2. Вычисляем гипердерева индекс и случайное подмножество как

  $$j, (x_1, ..., x_k) \leftarrow PORS(s, m)$$

  3. Вычисляем PORST подпись и открытый ключ:

  $$(\sigma_d, oct, p), \text{используя } sign(seed, make - addr(d, j), x_1, ..., x_k)$$

  4. Для $i \in \{d - 1, ..., 0\}$ выполняется:

     (a) Вычисляем WOTS подпись:

     $$\sigma_i \leftarrow WOTS, \text{используя } sign(seed, make - addr(i, j), p)$$

     (b) Вычисляем $p \leftarrow WOTS$, используя $extractpk(p, \sigma_i)$.

     (c) $j^* \leftarrow \lfloor j/2^h \rfloor$.

(d) Для $u \in \{0, ..., 2^h - 1\}$ вычислим WOTS открытый ключ:

$$p_u \leftarrow WOTS, \text{используя } genpk(seed, make-addr(i, 2^h, j^* + u))$$

(e) Вычислим Меркля аутентификацию:

$$A_i \leftarrow Merkle, \text{используя } auth_h(p_0, ..., p_{2^h-1}, j - 2^h j^*)$$

(f) $j \leftarrow j^*$.

5. Для $0 \leq u < 2^{c+h}$ вычислим WOTS открытый ключ:

$$p_u \leftarrow WOTS, \text{используя } genpk(seed, make-addr(0, u))$$

6. Вычислим Меркля аутентификацию:

$$(a_1, ..., a_{h+c}) \leftarrow Merkle, \text{используя } auth_{h+c}(p_0, ..., p_{2^{h+c}-1}, 2^h j)$$

7. $A_c \leftarrow (a_{h+1}, ..., a_{h+c})$.
8. Получаем подпись $(s, \sigma_d, oct, \sigma_{d-1}, A_{d-1}, ..., \sigma_0, A_0, A_c)$.

- **Алгоритм проверки подписи Verify:**

V получает на вход хэш $m \in B_n$, открытый ключ $pk \in B_n$ и подпись

$$(s, \sigma_d, oct, \sigma_{d-1}, A_{d-1}, ..., \sigma_0, A_0, A_c)$$

и проверяет это следующим образом:

1. Вычислим индекс гипердерева и случайное подмножество

$$j, (x_1, ..., x_k) \leftarrow PORS(s, m)$$

2. Вычислим открытый ключ PORST,

$$p \leftarrow PORST, \text{используя } extractpk(x_1, ..., x_k, \sigma_d, oct).$$

3. Если $p = \perp$, затем прерываем и возвращаем 0.
4. Для $i \in \{d - 1, ..., 0\}$ выполняем следующее:

(a) Вычислим открытый ключ WOTS:

$$p \leftarrow WOTS, \text{используя } extractpk(p, \sigma_i)$$

(b) $j^* \leftarrow \lfloor j/2^h \rfloor$.

(c) Вычислим корень дерева Меркля:

$$p \leftarrow Merkle, \text{используя } extract_h(p, j - 2^h j^*, A_i)$$

(d) $j \leftarrow j^*$.

5. Вычислим корень дерева Меркля:

$$p \leftarrow Merkle, \text{используя } extract_c(p, j, A_c)$$

6. В результате 1, если $p = pk$ и 0 в ином случае.

2.3 SPHINCS$^+$

Хотя в практическом плане размер подписи и скорость SPHINCS далеки от того, к чему мы привыкли, от подписей RSA или ECDSA. В данной работе представлен алгоритм SPHINCS$^+$ [5], который улучшает SPHINCS с точки зрения скорости алгоритмов схемы и размера подписи.

SPHINCS$^+$ использует псевдослучайную функцию PRF для генерации ключей, $PRF : \{0,1\}^n \times \{0,1\}^{256} \rightarrow \{0,1\}^n$, и псевдослучайную функцию PRF_{msg} для генерации случайного сжатия сообщения: $PRF_{msg} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \rightarrow \{0,1\}^n$. Для сжатия подписываемого сообщения мы используем дополнительную хэш-функцию H_{msg}, которая может обрабатывать сообщения произвольной длины:

$$H_{msg} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \rightarrow \{0,1\}^m$$

SPHINCS$^+$ Личный и открытый ключ:

- Открытый ключ состоит из двух n-битных значений: корневого узла из трех верхних в гипердереве и случайного открытого начального значения PK.

- Личный ключ состоит еще из двух n-битных случайных: SK, чтобы генерировать WOTS$^+$ и $FORS$ личные ключи, и $SK.prf$, используемый ниже для случайного хэша сообщения.

SPHINCS$^+$ Подпись сообщения.

Как не удивительно, что подпись состоит из FORS подписи для хэша сообщения, WOTS$^+$ подпись соответствующих открытых ключей FORS, ряда каналов аутентификации для подтверждения того, что WOTS$^+$ является открытым ключом. Чтобы проверить эту цепочку путей и подписей, проверка итеративно восстанавливает открытые ключи и корневые узлы до тех пор, пока не будет достигнут корневой узел в верхней части гипердерева SPHINCS$^+$.

Расмотрим два момента из алгоритма:

- Вычисление хэша сообщения.

- Выбор листа.

Здесь SPHINCS$^+$ отличается от оригинальных SPHINCS тонкими, но важными деталями.

Во-первых, мы псевдо случайным образом генерируем случайные числа R, основанные на сообщении и $SK.prf$. R может быть дополнительно сконструирован недетерминированным путем добавления дополнительной случайности $OptRand$. Это может противодействовать атакам бокового канала, которые полагаются на сбор нескольких следов для одного и того же вычисления. Обратите внимание, что установка этого значения в нулевую строку (или использование значения с низкой энтропией) не оказывает отрицательного влияния на псевдослучайность R. Формально, мы полагем, что $R = PRF(SK.prf, OptRand, M)$. R часть подписи. Используя R, мы затем получаем индекс конечного узла, который должен использоваться, а также получаем хэш сообщения $(MD||idx) = H_{msg}(R, PK, PK.root, M)$.

В отличие от SPHINCS, этот метод выбора индекса является публично проверяемым, не позволяя злоумышленнику свободно выбирать кажущийся случайным индекс и комбинировать его с сообщением по своему выбору. Критически важно, что это противодействует многоцелевым атакам на схему подписи FTS. Поскольку индекс теперь может быть вычислен верификатором, он больше не включается в подпись.

3 Програмная реализация

В распределенной базе данных блокчейн используются подпись на эллиптических кривых ECDSA (Elliptic Curve Digital Signature Algorithm) с кривой $secp256k1$ (см. Рис. 5) для подписания транзакций и отправки их в сеть. Реализованы подписи на основе функций хэширования без состояния, (SPHINCS, SPHINCS^{+}, Gravity-SPHINCS) на языке Python. А так же интегрированы в современную блокчейн архитектуру под названием Bitshares, заменив подпись на эллиптической кривой.

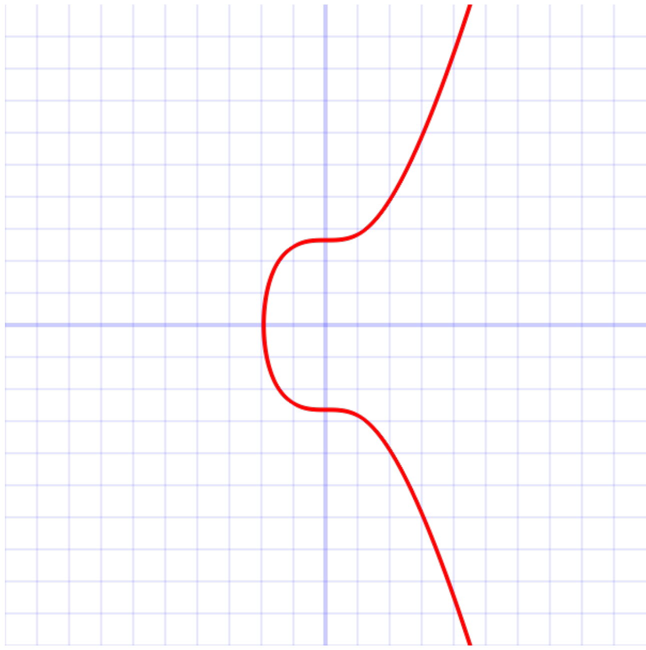

Рис. 5: Эллиптическая кривая secp256k1 для подписи ECDSA

3.1 Введение в блокчейн Bitshares

В 2013 году под авторством Даниила Ларимера была опубликована статья с упоминанием Bitshares. Идея протокола Bitshares состоит в создании платформы, с помощью которой можно было бы торговать разными активами и валютами в децентрализованной среде.

3.2 Назначение платформы Bitshares

Протокол реализует децентрализованную биржу, где этими цифровыми активами можно торговать. При проектировании учетной системы и механизма достижения консенсуса разработчики сделали большой упор на пропускную способность. Как результат, Bitshares позиционирует себя как децентрализованная альтернатива учетной системе $Visa$. В то время как $Visa$ заявляет, что может обрабатывать пару десятков тысяч транзакций в секунду, Bitshares говорит о способности обрабатывать сто тысяч транзакций в секунду, причем децентрализованным образом, с открытой базой данных и возможностью аудита.

3.3 Достижение консенсуса на основе DPoS

Правила работы протокола DPoS (Delegated Proof of Stake) предполагают, что все пользователи могут принимать участие в достижении консенсуса, выбирая валидаторов посредством голосования. В процессе голосования вес голоса пользователя определяется его балансом в базовой валюте. Формирование блоков выполняется подмножеством избранных валидаторов. В рамках протокола Bitshares валидатор называется $witness$.

3.4 Модель транзакций

Детальнее остановимся на модели транзакций в Bitshares. Так как основная работа заключалась в замене подписи транзакций в данной платформе на подписи на основе функций хэширования без состояния. (см. Рис. 6).

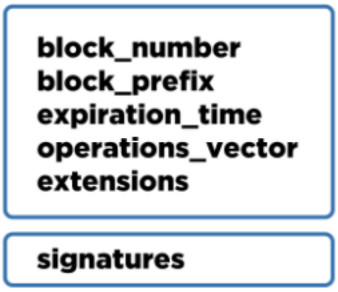

Рис. 6: Модель транзакции в Bitshares

На схеме видно, что тело транзакции состоит из пяти основных полей. Первые два поля транзакции необходимы для того, чтобы привязать ее к определенному блоку. Это нужно, чтобы определить цепочку блоков, в которую эта транзакция может быть добавлена, поскольку по правилам протокола транзакция не может быть подтверждена в той цепочке, к которой не привязана. Поле $expiration\_time$ задает время, до которого транзакция может быть добавлена в блок. Если она не была подтверждена до наступления этого времени, то она считается невалидной и уже не может быть включена в блокчейн.

Поле $operations\_vector$ является особенным. Эта особенность состоит в том, что в него можно поместить много разных операций. Операция — это еще один ключевой объект в протоколе Bitshares. Назовем несколько самых популярных типов операций: $transfer$ (перевод), $account\_update$ (обновление аккаунта), $asset\_issue$ (выпуск токена) Каждая операция имеет свой формат и необходимые параметры. Например, операция $transfer$ требует указания аккаунта отправителя, типа актива, суммы перевода и аккаунта получателя. Сами операции независимы друг от друга, но могут быть выполнены только вместе, если транзакция будет принята. То есть мы можем сделать несколько переводов средств между аккаунтами и выпустить все эти переводы одной транзакцией.

Поле $extensions$ сделано для обратной совместимости, чтобы текущая версия программного обеспечения могла обрабатывать транзакции новой версии, где могут быть добавлены дополнительные поля. Конечно же, старое ПО не будет знать, как правильно верифицировать дополнительные поля новых транзакций, но хотя бы сможет корректно обрабатывать транзакции согласно старым правилам.

Это формат неподписанной транзакции. Для того чтобы транзакцию правильно подписать, нужно проанализировать все операции из *operations_vector* и составить список аккаунтов, которые должны подтвердить данную транзакцию. Тогда станет ясно, какими ключами нужно подписывать транзакцию. Все необходимые подписи помещаются в отдельное поле — *signatures*. Если не будет хватать хотя бы одной подписи, то вся транзакция будет считаться неправильной.

Отметим, что за счет оптимизации размера идентификаторов финальный размер транзакции, которая содержит одну операцию будет равен приблизительно 100 байт. Это действительно очень компактная транзакция, если сравнить ее с транзакцией в других протоколах.

Что касается комиссионных сборов, то в протоколе Bitshares реализован особый подход, называется он fee. Каждая операция требует определенной оплаты, которая снимается с баланса аккаунта инициатора в момент подтверждения транзакции. Комиссия за осуществление операций может быть постоянной, а может меняться. В качество грубого сравнения можно отметить, что комиссии за обычные переводы и торговлю значительно ниже, чем комиссии за выпуск новых активов и регистрацию нового аккаунта.

3.5 Взаимодействие с Bitshares

API Bitshares доступны с помощью удаленных вызовов процедур(RPC) и вызовов и уведомлений $WebSocket$. Все вызовы API форматируются в формате $JSON$ и возвращают только $JSON$. Ссылки на API Bitshares-$Core$ находятся в документации *Doxygen*, которая генерируется для каждой версии Bitshares на языке $Perl$. Кроме того, вы можете найти информацию о классах, компонентах и элементах API в подробной и структурной документации Bitshares.

API — интерфейсы разделяются на две категории, а именно:

- $Blockchain\ API$ — используется для запроса блокчейн-данных(счета, активы, торговая история и т.д.). Кроме того, данные хранятся в самом блокчейне (блоки, транзакции и т.д.), объекты более высокого (например, счета, балансы и т.д.) можно получить через полную базу данных узла.

- $Wallet\ API$ – отдельный модуль взаимодействия с блокчейном, для удобство разработчиков и тестирование новых операций.

Кошелек (*cli-wallet*) имеет ваши личные ключи и возможности подписи. Он требует работающего полного узла ($witness$) (не обязательно локально) и подключается к нему. Потому что кошелек не предлагает возможности $P2P$ или *blockchain* напрямую.

3.6 Одноранговый сетевой протокол Bitshares

Узлы Bitshares взаимодействуют друг с другом через одноранговый сетевой протокол ($P2P$).

Каждый узел принимает соединения через TCP-сокет(не обязательно открытый). Сразу же после установления соединения узлы обмениваются криптографическими ключами, которые впоследствии используются для шифрования трафика внутри этого соединения.

Протокол состоит из сообщений, которыми обмениваются через зашифрованное соединение. Протокол поддерживает различные типы сообщений для запроса информации или передачи элементов блокчейна.

3.6.1 Коммуникационные уровни

- Уровень шифрования

  Весь сетевой трафик после первоначального обмена ключами шифруется с помощью $AES-256$.

  Для обмена ключами каждый узел создает случайный личный ключ на кривой $secp256k1$, вычисляет соответствующий открытый ключ и передает его в открытом виде по соединению.

  После получения удаленного открытого ключа он умножается на собственный личный ключ. Результирующая точка кривой хэшируется с помощью $SHA-512$, чтобы получить общий хэш 512 бит.

  Из этого общего секрета создается 256-битный ключ путем хэширования его с помощью $SHA-256$. Аналогично, 128-битный создается путем хэширования секрета с помощью $city\_hash\_128$. 256-битный ключ и 128-битный затем используются для настройки потоков шифрования и расшифрования $AES-256-CBC$ для отправки и приема данных.

- Уровень обмена сообщениями

  Сообщения состоят из заголовка 8 байт (4 байта *little-endian* целочисленного размера, 4 байта *little-endian* целочисленного типа) плюс фактическое содержимое сообщения. Содержимое представляет собой двоичное сериализованное представление структуры данных, обозначенной полем тип.

  Для передачи сообщения дополняются кратным 16 байтам. (16 байт — это размер блока, обрабатываемого базовыми потоками AES. Таким образом, сообщения всегда могут быть зашифрованные или расшифрованными без необходимости ждать дальнейших данных.)

3.6.2 Жизненный цикл подключения

$P2P$ — соединения, как правило, долговечны. Узел будет пытаться подключиться к определенному минимальному числу одноранговых узлов и может принимать дополнительные соединения до определенного максимального числа. Узлы разъединяются только тогда, когда они в каком-то смысле плохо себя ведут, то есть вредят сети отправляя некорректные данные.

3.7 Интеграция языков программирования

В данной работе, реализация подписей на основе функций хэширования использовался $Python$, в том время, когда платформа Bitshares написана на $C++$. Поэтому появилась необходимость интегрировать $Python$ в проект Bitshares. Для интеграции $C++$ кода в $Python$ используется библиотека $Boost.Python$. Однако в данной работе потребовалось сделать обратное: вызвать код $Python$ со стороны $C++$. Это требует встроить интерпретатор $Python$ в $C++$ программу.

В настоящее время $Boost.Python$ не поддерживает напрямую все, что нужно при встраивании. Поэтому нужно использовать $APIPython/C$ для заполнения пробелов. Тем не менее, $Boost.Python$ уже значительно упрощает встраивание и в будущей версии может вообще не потребоваться касаться $APIPython/C$.

3.7.1 Сборка встроенных программ

Чтобы иметь возможность встраивать $Python$ в свои программы, мы должны ссылаться как на $Boost.Python$, так и на собственную библиотеку времени выполнения $Python$.

Библиотека $Boost.Python$ поставляется в двух вариантах. Оба находятся в */libs/python/build/bin.stage* подкаталоге $Boost$. В $Windows$ варианты называются *boost_python.lib*(для выпусков сборки) и *boost_python_debug.lib*(для отладки). Если вы не можете найти библиотеки, возможно, вы еще не создали $Boost.Python$.

Библиотека $Python$ находится в */libs* подкаталоге вашего каталога $Python$. В $Windows$ это называется *pythonXY.lib*, где XY — ваш основной номер версии $Python$.

Кроме того, */include* подкаталог $Python$ должен быть добавлен в ваш путь включения.

В $Jamfile$(краткое описание вышеперечисленного) сводится к:

```
Projectroot c:\projects\embedded_program ;

SEARCH on python.jam = $(BOOST_BUILD_PATH) ;
include python.jam ;

exe embedded_program
: #sources
    embedded_program.cpp
: # requirements
    <find-library>boost_python <library-path>c:\boost\libs\python
```

```
    $(PYTHON_PROPERTIES)
        <library-path>$(PYTHON_LIB_PATH)
        <find-library>$(PYTHON_EMBEDDED_LIBRARY) ;
```

3.7.2 Подготовка к работе

Для встраивания интерпретатора *Python* в одну из программ на $C++$ необходимо выполнить следующие 3 шага:

1. Подключить $\#include < boost/python.hpp >$.

2. Вызовите $Py\_Initialize()$ для запуска интерпретатора и создать $\_\_main\_\_$ модуль.

3. Вызовите другие процедуры *API Python C*, чтобы использовать интерпретатор.

3.7.3 Использование интерпетатора

Объекты в *Python* подсчитываются по ссылкам. Естественно, *PyObjectAPI Python C* также подсчитываются по ссылкам. Однако есть разница. Хотя подсчет ссылок в Python полностью автоматический, *API*-интерфейс *Python C* требует, чтобы вы делали это вручную . Это грязно и особенно трудно понять в присутствии исключений $C++$. К счастью, *Boost.Python* предоставляет шаблоны дескрипторов и классов объектов для автоматизации процесса.

3.7.4 Запуск кода Python

Boost.python предоставляет три связанные функции для запуска кода *Python* из $C++$.

```
object eval(str expression, object globals = object(), object locals = object())
object exec(str code, object globals = object(), object locals = object())
object exec_file(str filename, object globals = object(), object locals = object())
```

функция *eval* вычисляет выражение и возвращает полученное значение. *exec* выполняет данный код(обычно набор операторов), возвращающий результат, а $exec\_file$ выполняет код, содержащийся в данном файле.

Параметры *globals* и *locals* — это словари *Python*, содержащие глобальные и локальные значения контекста, в котором выполняется код. Для большинства намерений и целей вы можете использовать словарь пространства имен модуля $\_\_main\_\_$ для обоих параметров.

Boost.python предоставляет функцию для импорта модуля:

```
object import(str name)
```

import импортирует модуль *python*(потенциально загружая его сначала в запущенный процесс) и возвращает его.

Давайте импортируем модуль $\_\_main\_\_$ и запустим некоторый код *Python* в его пространстве имен:

```
object main_module = import("__main__");
object main_namespace = main_module.attr("__dict__");

object ignored = exec("hello = file('hello.txt', 'w')\n"
                      "hello.write('Hello world!')\n"
                      "hello.close()",
                      main_namespace);
```

Это должно создать файл под названием "*hello.txt*" в текущем каталоге, содержащем фразу, которая хорошо известна в кругах программирования.

3.8 Результаты

Создание на Macbook Pro(3.1 GHz i5, 8GB оперативной памяти), пар ключей одноразовой подписи и дерева сертификации Меркля разных размеров дало следующие результаты(WOTS): $2^4 = 0.465s, 2^5 = 1.135s, 2^6 = 3.650s, 2^8 = 14.540s$. Создание гипердерева, состоящего из начальной генерации двух 2^4 деревьев, занимает около 1 секунды по сравнению с $14s$, требующимися для генерации стандартного 2^8 дерева MSS для одного и того же объема подписей.

Общая идея гипердерева состоит в том, что корень дочернего дерева Меркля подписывается ключом одноразовой подписи из хэша листа родительского дерева Меркля, известного как дерево сертификации. Проблема с базовой MSS заключается в том, что количество доступных подписей ограничено, и все пары ключей одноразовых подписей должны быть предварительно сгенерированы до вычисления дерева Меркля. Генерация ключей и время подписания растут экспоненциально относительно высоты дерева, h, что означает, что деревья, превышающие 256 ключей одноразовой подписи, становятся затратными по параметрам времени и вычислительной мощности, необходимых для генерации. Стратегия отсрочки вычислений при генерации ключей и деревьев, а также расширение количества доступных пар ключей одноразовой подписи заключается в использовании дерева, которое само состоит из деревьев Меркля, называемого гипердеревом. Размер подписей растет линейно для каждого дополнительного дерева, которое подписывается, в то время как объём подписей гипердерева увеличиваентся экспоненциально.

Увеличение глубины(или высоты) гипердерева продолжает эту тенденцию. Гипердерево, состоящее из четырех соединенных 2^4 деревьев сертификации и дерева подписи размером 2^4, может содержать $2^{20} = 1048576$ подписей с увеличенным размером подписи, но при этом время создания составляет всего $2.420s$.

Нет необходимости, чтобы гипердерево было симметричным, и поэтому, если оно состояло первоначально из двух деревьев, оно может быть расширено впоследствии путем присоединения дополнительных слоев деревьев. Таким образом, подлписи блока транзакций будут изначально небольшого размера, который будет постепенно возрастать по мере увеличения глубины гипердерева. Использование гипердерева Меркля для создания и подписи адреса блока транзакций вряд ли потребуется для количества транзакций превышающего 2^{12}. Таким образом, возможность создать с вычислительной легкостью 2^{20} защищенных подписей для глубины гипердерева $h = 5$ является более чем достаточной.

Использование схемы подписи Меркля MSS безопасно основывается на неиспользовании повторно ключей одноразовой подписи. Таким образом, это зависит только от состояния подписей или записей о подписанных транзакциях. Как правило, в реальном мире это потенциально может быть проблемой, но неизменяемый публичный блок цепочки транзакций является идеальным хранилищем для криптографической схемы подписи с учетом состояния. В 2015 году стало известно о новой схеме криптографической подписи на основе функций хэширования под названием SPHINCS (с алгоритмом подписи можно ознакомится выше), которая предлагает практически не зависящие от состояния подписи с 2^{128}-битной защитой.

Чтобы получить контрольные показатели, мы оцениваем реализацию на машине, используя набор инструкций Intel x86-64. В частности, используем одноядерный процессор $Intel\ Core\ i5$ с частотой 3,1 ГГц. Мы следуем стандартной практике отключения $TurboBoost$ и $hyper-threading$, для чистоты эксперимента. Система имеет 32 КБ кэша инструкций $L1$, 32 КБ кэша данных $L1$, 256 КБ кэша $L2$ и 8192 КБ кэша $L3$. Кроме того, он имеет 8 ГБ оперативной памяти. При выполнении тестов производительности система работала на ядре $Linux4.9.0-4-amd64$. Для компилиляции кода, использовался GCC версии 8.3.0, с флагом оптимизации компилятора.

Таблица 1: Сравнение подписей

Algorithm	**Key generation**	**Sign**	**Verify**
SPHINCS-256	12.6 ms	236 ms	2.73 ms
SPHINCS$^+$	11.7 ms	196 ms	2.3 ms
Gravity-SPHINCS	10.3 ms	204 ms	2.4ms
ECDSA(P-256)	0.924 ms	0.553 ms	0.478 ms

4 Заключение

В дипломной работе получены следующие результаты:

1. Подготовлен анализ публикаций по основам HBC (Hash-Based Cryptography).

2. Изучены современные алгоритмы электронно цифровых подписей на основе функций хэширования.

3. Реализованы ЭЦП на основе функций хэширования на языке Python.

4. Интегрированны в блокчейн архитектуру Bitshares.

5. Составлена таблица сравнения скорости алгоритмов подписей без состояния с подписью на эллиптических кривых ECDSA для подписания транзакции (См. 1).

Исходя из моей работы и тестирования их на практике, подписи на основе функций хэширования не идеальны, так как требуют большего времени на генерацию ключей, подпись и проверку, чем нынешние решения на эллиптических кривых. А так же существует проблема и с хранением самих подписей, они требуют больше затрат по памяти, но не смотря на это они обеспечивают безопасность данных, что важнее в наше время. Сейчас ведутся активные исследования в данной области и совсем скоро проблемы с их оптимизацией решатся. У меня получилось реализовать электронно цифровые подписи на основе функций хэширования без состояния, такие как SPHINCS-256, SPHINCS$^+$, Gravity-SPHINCS для подписания транзакций и интегрировать в существующий протокол блокчейна под названием Bitshares, а так же сравнить с ECDSA. Доказав, что распределенную модель хранения данных, блокчейн, и одноранговую сеть для обмена сообщений можно защитить используя актуальные криптографические работы и статьи ученых в области HBC (Hash-Based Cryptography).

Список литературы

[1] Security of One-Time Signatures under Two-Message Attacks. Andreas Hülsing, https://eprint.iacr.org/2016/1042.pdf.

[2] On the Security of the Winternitz One-Time Signature Scheme. Johannes Buchmann, Erik Dahmen, Sarah Ereth, https://eprint.iacr.org/2010/446.pdf.

[3] Short One-Time Signatures. Gregory M. Zaverucha and Douglas R. Stinson. https://eprint.iacr.org/2011/191.pdf.

[4] WOTS$^+$ – Shorter Signatures for Hash-Based Signature Schemes. Andreas Hulsing, https://eprint.iacr.org/2017/965.pdf.

[5] Proof-of-forgery for hash-based signatures. E.O. Kiktenko, M.A. Kudinov, A.A. Bulychev, and A.K. Fedorov, https://arxiv.org/pdf/1905.12993.pdf.

[6] Improving Stateless Hash-Based Signatures. Jean-Philippe Aumasson and Guillaume Endignoux, https://eprint.iacr.org/2017/933.pdf.

[7] The SPHINCS$^+$ Signature Framework. Daniel J. Bernstein, https://eprint.iacr.org/2019/1086.pdf.

Приложение

Пример реализации SPHINCS-256 на языке Python

```
class SPHINCS(object):

    #    def __init__(self, n=256, m=512, h=60, d=12, w=16, tau=16, k=32):
    def __init__(self, n=256, m=512, h=60, d=12, w=16, tau=16, k=32):

        self.n = n
        self.m = m
        self.h = h
        self.d = d
        self.w = w
        self.tau = tau
        self.t = 1 << tau
        self.k = k

        self.Hdigest = lambda r, m: BLAKE(512).digest(r + m)
        self.Fa = lambda a, k: BLAKE(256).digest(k + a)
        self.Frand = lambda m, k: BLAKE(512).digest(k + m)

        C = bytes("expand 32-byte to 64-byte state!", 'latin-1')
        perm = ChaCha().permuted
        self.Glambda = lambda seed, n: ChaCha(key=seed).keystream(n)
        self.F = lambda m: perm(m + C)[:32]
        self.H = lambda m1, m2: perm(xor(perm(m1 + C), m2 + bytes(32)))[:32]

        self.wots = WOTSplus(n=n, w=w, F=self.F, Gl=self.Glambda)
        self.horst = HORST(n=n, m=m, k=k, tau=tau,
                           F=self.F, H=self.H, Gt=self.Glambda)

    def address(self, level, subtree, leaf):
        t = level | (subtree << 4) | (leaf << 59)
        return int.to_bytes(t, length=8, byteorder='little')

    def wots_leaf(self, address, SK1, masks):
        seed = self.Fa(address, SK1)
        pk_A = self.wots.keygen(seed, masks)

        def H(x, y, i): return self.H(xor(x, masks[2*i]), xor(y, masks[2*i+1]))
        return root(l_tree(H, pk_A))

    def wots_path(self, a, SK1, Q, subh):
        ta = dict(a)
        leafs = []
        for subleaf in range(1 << subh):
            ta['leaf'] = subleaf
            leafs.append(self.wots_leaf(self.address(**ta), SK1, Q))
        Qtree = Q[2 * ceil(log(self.wots.l, 2)):]

        def H(x, y, i): return self.H(xor(x, Qtree[2*i]), xor(y, Qtree[2*i+1]))
        tree = list(hash_tree(H, leafs))
        return auth_path(tree, a['leaf']), root(tree)
```

```
def keygen(self):
    SK1 = os.urandom(self.n // 8)
    SK2 = os.urandom(self.n // 8)
    p = max(self.w-1, 2 * (self.h + ceil(log(self.wots.l, 2))), 2*self.tau)
    Q = [os.urandom(self.n // 8) for _ in range(p)]
    PK1 = self.keygen_pub(SK1, Q)
    return (SK1, SK2, Q), (PK1, Q)

def keygen_pub(self, SK1, Q):
    addresses = [self.address(self.d - 1, 0, i)
                 for i in range(1 << (self.h//self.d))]
    leafs = [self.wots_leaf(A, SK1, Q) for A in addresses]
    Qtree = Q[2 * ceil(log(self.wots.l, 2)):]

    def H(x, y, i): return self.H(xor(x, Qtree[2*i]), xor(y, Qtree[2*i+1]))
    PK1 = root(hash_tree(H, leafs))
    return PK1

def sign(self, M, SK):
    SK1, SK2, Q = SK

    R = self.Frand(M, SK2)
    R1, R2 = R[:self.n // 8], R[self.n // 8:]
    D = self.Hdigest(R1, M)
    i = int.from_bytes(R2, byteorder='big')
    i >>= self.n - self.h
    subh = self.h // self.d
    a = {'level': self.d,
         'subtree': i >> subh,
         'leaf': i & ((1 << subh) - 1)}
    a_horst = self.address(**a)
    seed_horst = self.Fa(a_horst, SK1)
    sig_horst, pk_horst = self.horst.sign(D, seed_horst, Q)
    pk = pk_horst
    sig = [i, R1, sig_horst]
    for level in range(self.d):
        a['level'] = level
        a_wots = self.address(**a)
        seed_wots = self.Fa(a_wots, SK1)
        wots_sig = self.wots.sign(pk, seed_wots, Q)
        sig.append(wots_sig)
        path, pk = self.wots_path(a, SK1, Q, subh)
        sig.append(path)
        a['leaf'] = a['subtree'] & ((1 << subh) - 1)
        a['subtree'] >>= subh
    return tuple(sig)

def verify(self, M, sig, PK):
    i, R1, sig_horst, *sig = sig
    PK1, Q = PK
    Qtree = Q[2 * ceil(log(self.wots.l, 2)):]
    D = self.Hdigest(R1, M)
    pk = pk_horst = self.horst.verify(D, sig_horst, Q)
```

```
if pk_horst is False:
    return False
subh = self.h // self.d

def H(x, y, i): return self.H(xor(x, Q[2*i]), xor(y, Q[2*i+1]))

def Ht(x, y, i): return self.H(
    xor(x, Qtree[2*i]), xor(y, Qtree[2*i+1]))
for _ in range(self.d):
    wots_sig, wots_path, *sig = sig
    pk_wots = self.wots.verify(pk, wots_sig, Q)
    leaf = root(l_tree(H, pk_wots))
    pk = construct_root(Ht, wots_path, leaf, i & 0x1f)
    i >>= subh
return PK1 == pk
```