МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«Ярославский государственный университет им. П.Г. Демидова»

Дипломная работа

Программная реализация современных отечественных стандартов функции хеширования и цифровой подписи

02.03.01. Математика и компьютерные науки

Исполнитель: Сычев Р.С.
гр. МКН-41 БО
Руководитель: к.ф.-м.н. Яблокова С.И.

Ярославль 2020

Содержание

Введение

В современном мире защита информации является неотъемлемой частью жизни общества. Криптография бывает необходима в самых разных сферах будь то переписка или доступ к документации. Особую важность имеет надежность и достоверность обмена информацией между государственными структурами, ведь любая невыявленная подделка может привести к колоссальным убыткам и нарушению общественного порядка. С другой стороны, проволочки в документообороте могут мешать оперативному реагированию, что недопустимо при наличии лучших альтернатив. Таким образом, среди прочего, возникает необходимость на высшем государственном уровне определить актуальные процедуры, позволяющие оперативно определить целостность сообщения и достоверность отправителя.

Ключевую роль в воросах удостоверения информации играет электронная цифровая подпись, которая позволяет установить не только подлинность отправителя но и целостность сообщения. Как правило, в процедурах генерации и проверки цифровой подписи используется хэш-код подписываемого сообщения.

Целью данной работы является программная реализация функции хэширования и цифровой подписи согласно ГОСТ Р 34.11-2012 и ГОСТ Р 34.10-2012 соответственно. Все специфичные классы предполагается реализовать на языке программирования C# в составе среды Visual Studio 2019. Выбор обусловлен популярностью данной среды разработки. В итоге, планируется реализовать хэш-функцию , процедуру генерации цифровой подписи и процедуру проверки цифровой подписи.

Объект исследования данной работы— криптографическая защита информации.

Предмет исследования— функция хэширования, процессы формирования и проверки электронной цифровой подписи.

В рамках реализации намеченой цели можно выделить следующие подзадачи:

— анализ ГОСТ Р 34.11-2012;
— реализация функции хэширования согласно ГОСТ Р 34.11-2012;
— анализ ГОСТ Р 34.10-2012;
— реализация цифровой подписи согласно ГОСТ Р 34.10-2012;
— анализ результатов.

Данная тема может быть интересна тем, кто заинтересован в реализации функции хэширования и цифровой подписи в различных программных средах. Данная работа также раскрывает некоторые прикладные вопросы процесса реализации. Актуальность выбранной темы обусловлена перспективой внедрения новых стандартов в государственные информационные системы.

1 Термины, определения и обозначения

В настоящей дипломной работе применяются следующие термины и обозначения с соответствующими определениями.

Дополнение — приписывание дополнительных бит к строке бит.

Инициализационный вектор — вектор, определенный как начальная точка работы функции хэширования.

Ключ подписи — элемент секретных данных, специфичный для субъекта и используемый только данным субъектом в процессе формирования цифровой подписи.

Ключ проверки подписи — элемент данных, математически связанный с ключом подписи и используемый проверяющей стороной в процессе проверки цифровой подписи.

Параметр схемы ЭЦП — элемент данных, общий для всех субъектов схемы цифровой подписи, известный или доступный всем этим субъектам.

Подписанное сообщение — набор элементов данных, состоящий из сообщения и дополнения, являющегося частью сообщения.

Последовательность псевдослучайных чисел — последовательность чисел, полученная в результате выполнения некоторого арифметического (вычислительного) процесса, используемая в конкретном случае вместо последовательности случайных чисел.

Последовательность случайных чисел — последовательность чисел, каждое из которых не может быть предсказано (вычислено) только на основе знания предшествующих ему чисел данной последовательности.

Процесс проверки подписи — процесс, в качестве исходных данных которого используются подписанное сообщение, ключ проверки подписи и параметры схемы ЭЦП, результатом которого является заключение о правильности или ошибочности цифровой подписи.

Процесс формирования подписи — процесс, в качестве исходных данных которого используются сообщение, ключ подписи и параметры схемы ЭЦП, а в результате формируется цифровая подпись.

Свидетельство — элемент данных, представляющий соответствующее доказательство достоверности (недостоверности) подписи проверяющей стороне.

Случайное число — число, выбранное из определенного набора чисел таким образом, что каждое число из данного набора может быть выбрано с одинаковой вероятностью.

Сообщение — строка бит произвольной конечной длины.

Функция сжатия — итеративно используемая функция, преобразующая строку бит длиной L_1 и полученную на предыдущем шаге строку бит длиной L_2 в строку бит длиной L_2.

Хэш-код — строка бит, являющаяся выходным результатом хэш-функции.

Хэш-функция — функция, отображающая строки бит в строки бит фиксированной длины и удовлетворяющая следующим свойствам:

а) по данному значению функции сложно вычислить исходные данные, отображаемые в это значение;

б) для заданных исходных данных сложно вычислить другие исходные данные, отображаемые в то же значение функции;

в) сложно вычислить какую-либо пару исходных данных, отображаемых в одно и то же значение.

Электронная цифровая подпись (ЭЦП) — строка бит, полученная в результате процесса формирования подписи.

V^* — множество всех двоичных векторов-строк конечной размерности (далее - векторы), включая пустую строку.

$|A|$— размерность (число компонент) вектора $A \in V^*$ (если A - пустая строка, то $|A| = 0$).

V_n — множество всех n-мерных двоичных векторов, где n - целое неотрицательное число; нумерация подвекторов и компонент вектора осуществляется справа налево, начиная с нуля.

$\oplus$ — операция покомпонентного сложения по модулю 2 двух двоичных векторов одинаковой размерности.

$A\|B$ — конкатенация векторов A,$B \in V^*$, т.е. вектор из $V_{|A|+|B|}$, в котором левый подвектор из $V_{|A|}$ совпадает с вектором , а правый подвектор из $V_{|B|}$ совпадает с вектором .

A^n — конкатенация n экземпляров вектора A.

$\mathbb{Z}_{2^n}$ — кольцо вычетов по модулю 2^n.

$\boxplus$ — операция сложения в кольце $\mathbb{Z}_{2^n}$.

$Vec_n : \mathbb{Z}_{2^n} \to V_n$ — биективное отображение, сопоставляющее элементу кольца $\mathbb{Z}_{2^n}$ его двоичное представление, т.е. для любого элемента $z \in \mathbb{Z}_{2^n}$ представленного вычетом $z_0 + 2z_1 + \ldots 2^{n-1}z_{n-1}$, где $z_j \in \{0,1\}, j = 0, \ldots, n-1$, выполнено равенство $Vec_n(z) = z_{n-1}\| \ldots \|z_1\|z_0$.

$Int_n \colon V_n \to \mathbb{Z}_{2^n}$ — отображение, обратное отображению Vec_n, т.е. $Int_n = Vec_n^{-1}$.

$MSB_n \colon V^* \to V_n$ — отображение, ставящее в соответствие вектору $z_{k-1}\| \ldots \|z_1\|z_0, k \geqslant n$, вектор $z_{k-1}\| \ldots \|z_{k-n+1}\|z_{k-n}$.

$a := b$ — операция присваивания переменной a значения b.

$\Phi \circ \Psi$ — произведение отображений, при котором отображение Ψ действует первым.

M — двоичный вектор, подлежащий хэшированию, $M \in V^*$, $|M| < 2^{512}$.

$H\colon V^* \to V_n$ — функция хэширования, отображающая вектор (сообщение) M в вектор (хэш-код) $H(M)$.

IV — инициализационный вектор функции хэширования, $IV \in V_{512}$.

Z — множество всех целых цисел.

p — простое число, $p > 3$.

F_p — конечное простое поле, представленное как множество из p наименьших неотрицательных вычетов $\{0, 1, \ldots, p-1\}$.

$b\ (mod\ p)$ — минимальное неотрицательное число, сравнимое с b по модулю p.

2 Первая глава. Функция хэширования

2.1 Краткий обзор

В целом, концепция развития стандартов на функцию хэширования заключается в использовании минимального числа ресурсов для достижения устойчивости хэш-функции ко всем известным атакам.

Так, хэш-функция должна удовлетворять следующим свойствам:

а) по данному значению функции сложно вычислить исходные данные, отображаемые в это значение;

б) для заданных исходных данных сложно вычислить другие исходные данные, отображаемые в то же значение функции;

в) сложно вычислить какую-либо пару исходных данных, отображаемых в одно и то же значение.

В качестве входных данных хэш-функции берется двоичный вектор произвольной конечной длины (в большинстве реализаций за единицу длины принимается 1 байт).

Впервые в России государственный стандарт на функцию хэширования был введен в 1994 году в ГОСТ Р 34.11-94 [1]. Данный стандарт был основным стандартом функции хэширования РФ с 1994 по 2011 годы пока не был заменен новым отечественным стандартом ГОСТ Р 34.11-2012 [2]. Причиной введения нового стандарта во многом являлись выявленные исследователями несовершенства используемого алгоритма [3]. Новый вариант функции, именуемой «Стрибог», выгодно отличается от используемой до нее функции лучшей на данный момент теоретической оценкой криптостойкости. Также, по сравнению со старой версией, возросла длина выходного значения с 256 до 512 бит. Еще одно отличие заключается в том, что ГОСТ Р 34.11-2012 строго регламентирует все пареметры алгоритма хэширования в то время как ГОСТ Р 34.11-94 содержит только примеры допустимых значений параметров, не рекомендуемых к использованию на практике.

Стандарт ГОСТ Р 34.11-2012 определяет семейство хэш-функций состоящее из двух функций с длинами хэш-кода равными 256 и 512 битам соответственно.

Общая схема основана на хорошо известных конструкциях и преобразованиях. Так, в алгоритме используются конструкции Меркла-Домгарда [4] и Миагути-Пренеля [5]. Также ему приписывают соответствие схеме HAIFA [6].

2.2 Параметры алгоритма

Параметры алгоритма хэширования включают:

а) значение инициализационного вектора IV;

б) значения подстановки $\pi' : \mathbb{Z}_{2^8} \to \mathbb{Z}_{2^8}$ в виде массива;

в) значения перестановки $\tau \in S_{64}$ в виде массива;

г) значения матрицы $A \in GF(2)$, умножением на которую справа задается линейное преобразование множества V_{64};

д) значения итерационных констант.

Значение инициализационного вектора IV для функции хэширования с длиной хэш-кода 512 бит равно 0^{512}. Значение инициализационного вектора IV для функции хэширования с длиной хэш-кода 256 бит равно $(00000001)^{64}$.

Значения подстановки $\pi' = (\pi'(0), \ldots, \pi'(255))$ записаны ниже:

$\pi' = (252, 238, 221, 17, 207, 110, 49, 22, 251, 196, 250, 218, 35, 197, 4, 77, 233, 119, 240, 219, 147, 46, 153, 186, 23, 54, 241, 187, 20, 205, 95, 193, 249, 24, 101, 90, 226, 92, 239, 33, 129, 28, 60, 66, 139, 1, 142, 79, 5, 132, 2, 174, 227, 106, 143, 160, 6, 11, 237, 152, 127, 212, 211, 31, 235, 52, 44, 81, 234, 200, 72, 171, 242, 42, 104, 162, 253, 58, 206, 204, 181, 112, 14, 86, 8, 12, 118, 18, 191, 114, 19, 71, 156, 183, 93, 135, 21, 161, 150, 41, 16, 123, 154, 199, 243, 145, 120, 111, 157, 158, 178, 177, 50, 117, 25, 61, 255, 53, 138, 126, 109, 84, 198, 128, 195, 189, 13, 87, 223, 245, 36, 169, 62, 168, 67, 201, 215, 121, 214, 246, 124, 34, 185, 3, 224, 15, 236, 222, 122, 148, 176, 188, 220, 232, 40, 80, 78, 51, 10, 74, 167, 151, 96, 115, 30, 0, 98, 68, 26, 184, 56, 130, 100, 159, 38, 65, 173, 69, 70, 146, 39, 94, 85, 47, 140, 163, 165, 125, 105, 213, 149, 59, 7, 88, 179, 64, 134, 172, 29, 247, 48, 55, 107, 228, 136, 217, 231, 137, 225, 27, 131, 73, 76, 63, 248, 254, 141, 83, 170, 144, 202, 216, 133, 97, 32, 113, 103, 164, 45, 43, 9, 91, 203, 155, 37, 208, 190, 229, 108, 82, 89, 166, 116, 210, 230, 244, 180, 192, 209, 102, 175, 194, 57, 75, 99, 182).$

(2.1)

Значения перестановки $\tau = (\tau(0), \ldots, \tau(255))$ записаны ниже:

$\tau = (0, 8, 16, 24, 32, 40, 48, 56, 1, 9, 17, 25, 33, 41, 49, 57, 2, 10, 18, 26, 34, 42, 50, 58, 3, 11, 19, 27, 35, 43, 51, 59, 4, 12, 20, 28, 36, 44, 52, 60, 5, 13, 21, 29, 37, 45, 53, 61, 6, 14, 22, 30, 38, 46, 54, 62, 7, 15, 23, 31, 39, 47, 55, 63).$

(2.2)

Строки матрицы $A \in GF(2)$ записаны ниже последовательно в шестнадцатеричном виде. Строка матрицы с номером j, $j = 0,1,\ldots,63$, записанная в виде

$a_{j,15} \dots a_{j,0}$, где $a_{j,i} \in \mathbb{Z}_{16}$, $j = 0, \dots, 15$ есть $Vec_4(a_{j,15}) \| \dots \| Vec_4(a_{j,0})$.

8e20faa72ba0b470	47107ddd9b505a38	ad08b0e0c3282d1c	d8045870ef14980e
6c022c38f90a4c07	3601161cf205268d	1b8e0b0e798c13c8	83478b07b2468764
a011d380818e8f40	5086e740ce47c920	2843fd2067adea10	14aff010bdd87508
0ad97808d06cb404	05e23c0468365a02	8c711e02341b2d01	46b60f011a83988e
90dab52a387ae76f	486dd4151c3dfdb9	24b86a840e90f0d2	125c354207487869
092e94218d243cba	8a174a9ec8121e5d	4585254f64090fa0	accc9ca9328a8950
9d4df05d5f661451	c0a878a0a1330aa6	60543c50de970553	302a1e286fc58ca7
18150f14b9ec46dd	0c84890ad27623e0	0642ca05693b9f70	0321658cba93c138
86275df09ce8aaa8	439da0784e745554	afc0503c273aa42a	d960281e9d1d5215
e230140fc0802984	71180a8960409a42	b60c05ca30204d21	5b068c651810a89e
456c34887a3805b9	ac361a443d1c8cd2	561b0d22900e4669	2b838811480723ba
9bcf4486248d9f5d	c3e9224312c8c1a0	effa11af0964ee50	f97d86d98a327728
e4fa2054a80b329c	727d102a548b194e	39b008152acb8227	9258048415eb419d
492c024284fbaec0	aa16012142f35760	550b8e9e21f7a530	a48b474f9ef5dc18
70a6a56e2440598e	3853dc371220a247	1ca76e95091051ad	0edd37c48a08a6d8
07e095624504536c	8d70c431ac02a736	c83862965601dd1b	641c314b2b8ee083

(2.3)

Здесь в одной строке записаны четыре строки матрицы A, при этом в строке с номером i, $i = 0, \dots, 15$ записаны строки матрицы A с номерами $4i + j$, $j = 0, \dots, 3$.

Итерационные константы записаны в шестнадцатеричном виде. Значение константы, записанное в виде $a_{127} \dots a_0$, где $a_i \in \mathbb{Z}_{16}$, $i = 0, \dots, 127$, есть $Vec_4(a_{127}) \| \dots \| Vec_4(a_0)$:

$C_1 =$ b1085bda1ecadae9ebcb2f81c0657c1f2f6a76432e45d016714eb88d7585c4fc
4b7ce09192676901a2422a08a460d31505767436cc744d23dd806559f2a64507;

$C_2 =$ 6fa3b58aa99d2f1a4fe39d460f70b5d7f3feea720a232b9861d55e0f16b50131
9ab5176b12d699585cb561c2db0aa7ca55dda21bd7cbcd56e679047021b19bb7;

$C_3 =$ f574dcac2bce2fc70a39fc286a3d843506f15e5f529c1f8bf2ea7514b1297b7b
d3e20fe490359eb1c1c93a376062db09c2b6f443867adb31991e96f50aba0ab2;

$C_4 =$ ef1fdfb3e81566d2f948e1a05d71e4dd488e857e335c3c7d9d721cad685e353f
a9d72c82ed03d675d8b71333935203be3453eaa193e837f1220cbebc84e3d12e;

$C_5 =$ 4bea6bacad4747999a3f410c6ca923637f151c1f1686104a359e35d7800fffbd
bfcd1747253af5a3dfff00b723271a167a56a27ea9ea63f5601758fd7c6cfe57;

$C_6 =$ ae4faeae1d3ad3d96fa4c33b7a3039c02d66c4f95142a46c187f9ab49af08ec6
cffaa6b71c9ab7b40af21f66c2bec6b6bf71c57236904f35fa68407a46647d6e;

$C_7 =$ f4c70e16eeaac5ec51ac86febf240954399ec6c7e6bf87c9d3473e33197a93c9
0992abc52d822c3706476983284a05043517454ca23c4af38886564d3a14d493;

$C_8 =$ 9b1f5b424d93c9a703e7aa020c6e41414eb7f8719c36de1e89b4443b4ddbc49a

f4892bcb929b069069d18d2bd1a5c42f36acc2355951a8d9a47f0dd4bf02e71e;

$C_9 =$ 378f5a541631229b944c9ad8ec165fde3a7d3a1b258942243cd955b7e00d0984
800a440bdbb2ceb17b2b8a9aa6079c540e38dc92cb1f2a607261445183235adb;

$C_{10} =$ abbedea680056f52382ae548b2e4f3f38941e71cff8a78db1fffe18a1b336103
9fe76702af69334b7a1e6c303b7652f43698fad1153bb6c374b4c7fb98459ced;

$C_{11} =$ 7bcd9ed0efc889fb3002c6cd635afe94d8fa6bbbebab07612001802114846679
8a1d71efea48b9caefbacd1d7d476e98dea2594ac06fd85d6bcaa4cd81f32d1b;

$C_{12} =$ 378ee767f11631bad21380b00449b17acda43c32bcdf1d77f82012d430219f9b
5d80ef9d1891cc86e71da4aa88e12852faf417d5d9b21b9948bc924af11bd720.

(2.4)

2.3 Преобразования

В алгоритме хэширования применяются следующие преобразования:

$$\pi = Vec_8 \pi' Int_8 \colon V_8 \to V_8; \tag{2.5}$$

$$X[k] \colon V_{512} \to V_{512},\ X[k](a) = k \oplus a,\ k, a \in V_{512}, \tag{2.6}$$

где $a = a_{63} \| \ldots \| a_0 \in V_{512}$, $a_i \in V_8$, $i = 0, \ldots, 63$;

$$S \colon V_{512} \to V_{512},\ S(a) = S(a_{63} \| \ldots \| a_0) = \pi(a_{63}) \| \ldots \| \pi(a_0), \tag{2.7}$$

где $a = a_{63} \| \ldots \| a_0 \in V_{512}$, $a_i \in V_8$, $i = 0, \ldots, 63$;

$$P \colon V_{512} \to V_{512},\ P(a) = P(a_{63} \| \ldots \| a_0) = a_{\tau(63)} \| \ldots \| a_{\tau(0)}, \tag{2.8}$$

где $a = a_{63} \| \ldots \| a_0 \in V_{512}$, $a_i \in V_8$, $i = 0, \ldots, 63$;

$$L \colon V_{512} \to V_{512},\ L(a) = L(a_7 \| \ldots \| a_0) = l(a_7) \| \ldots \| l(a_0), \tag{2.9}$$

где $a = a_7 \| \ldots \| a_0 \in V_{512}$, $a_i \in V_{64}$, $i = 0, \ldots, 7$;

При этом, результат умножения вектора $b = b_{63} \ldots b_0 \in V_{64}$ на матрицу A есть вектор $c \in V_{64}$:

$$c = b_{63}(Vec_4(a_{0,15}) \| \ldots \| Vec_4(a_{0,0})) \oplus \cdots \oplus b_0(Vec_4(a_{63,15}) \| \ldots \| Vec_4(a_{63,0})), \tag{2.10}$$

где $b_i(Vec_4(a_{63-i,15}) \| \ldots \| Vec_4(a_{63-i,0})) =$

$$= \begin{cases} 0^{64}, & \text{если } b_i = 0, \\ (Vec_4(a_{63-i,15}) \| \ldots \| Vec_4(a_{63-i,0})), & \text{если } b_i = 1. \end{cases}$$

для всех $i = 0, \ldots, 63$.

2.4 Функция сжатия

Значение хэш-кода сообщения $M \in V_*$ вычисляется с использованием итерационной процедуры. На каждой итерации вычисления хэш-кода используется функция сжатия:

$$g_N : V_{512} \times V_{512} \to V_{512},\ N \in V_{512}, \tag{2.11}$$

значение которой вычисляется по формуле

$$g_N(h,m) = E(LPS(h \oplus N),m) \oplus h \oplus m, \tag{2.12}$$

где $E(K,m) = X[K_{13}]LPSX[K_{12}] \dots LPSX[K_2]LPSX[K_1](m)$.

Значения $K_i \in V_{512}$, $i = 1, \dots, 13$ вычисляются по формулам:

$$K_1 = K; \tag{2.13}$$

$$K_i = LPS(K_{i-1} \oplus C_{i-1}),\ i = 2, \dots, 13. \tag{2.14}$$

2.5 Алгоритм хэширования

Исходными данными для хэш-функции является сообщение $M \in V^*$ и $IV \in V_{512}$-инициализационный вектор.

Алгоритм вычисления состоит из трех этапов

Этап 1

Присвоить начальные значения текущих величин

1.1 $h := IV$;

1.2 $N := 0^{512} \in V_{512}$;

1.3 $\Sigma := 0^{512} \in V_{512}$;

1.4 Перейти к этапу 2.

Этап 2

2.1 Проверить условие $|M| < 512$.

При положительном исходе перейти к этапу 3.

В противном случае выполнить последовательность вычислений по 2.2—2.7.

2.2 Вычислить подвектор $m \in V_{512}$ сообщения M: $M = M'\|m$. Далее выполнить последовательность вычислений:

2.3 $h := g_N(h,m)$.

2.4 $N := Vec_{512}(Int_{512}(N) \boxplus 512)$.

2.5 $\Sigma := Vec_{512}(Int_{512}(\Sigma) \boxplus Int_{512}(m))$.

2.6 $M := M'$.

2.7 Перейти к шагу 2.1.

Этап 3

3.1 $m := 0^{511-|M|}\|1\|M$.

3.2 $h := g_N(h,m)$.

3.3 $N := Vec_{512}(Int_{512}(N) \boxplus |M|)$.

3.4 $\Sigma := Vec_{512}(Int_{512}(\Sigma) \boxplus Int_{512}(m))$.

3.5 $h := g_{0^{512}}(h, N)$.

3.6 $h := \begin{cases} g_{0^{512}}(h, \Sigma), & \text{для функции хэширования с длиной хэш-кода 512 бит;} \\ MSB_{256}(g_{0^{512}}(h, \Sigma)), & \text{для функции хэширования с длиной хэш-кода 256 бит.} \end{cases}$

3.7 Конец работы алгоритма.

Значение величины h, полученное на шаге 3.6, является значением функции хэширования $H(M)$.

Графическое изображение схемы алгоритма представлено на рисунке 2.1.

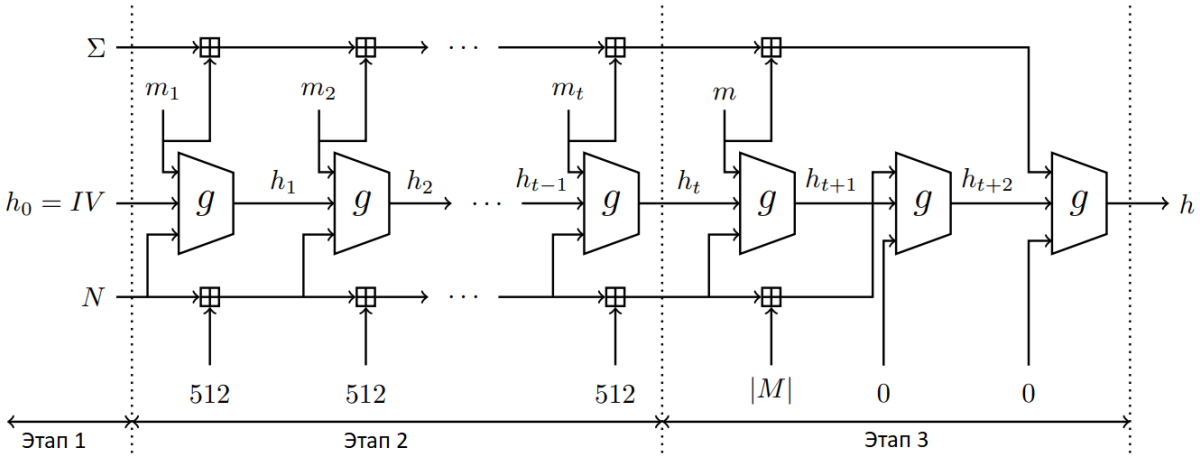

Рисунок 2.1 — Схема вычисления хэш-функции. В случае длины хэша равной 256 бит возвращается $MSB_{256}(h)$.

3 Вторая глава. Программная реализация хэш-функции

3.1 Общая схема алгоритма

В данном разделе рассматриваются типичные алгоритмические приемы используемые в различных реализациях хэш-функции. Затрагиваются вопросы механизма хранения констант и перечня используемых конструкций. Весь исходный код написан на языке программирования C#. По возможности минимизировано использование специфичных языковых конструкций, благодаря чему код может быть легко перенесен на любой C-подобный язык программирования.

Для начала определим используемые в алгоритме типы данных. Все описания функций и переменных, используемых при расчете хэша, целесообразно выделить в отдельный класс. Присвоим ему идентификатор «GR3411_2012_Hash», который и будем использовать. Использование выделенного класса упрощает работу с параметрами и позволяет удобно организовать дополнительные разовые расчеты, речь о которых пойдет в следующем пункте. Все дальнейшие описанные в данной главе методы и свойства предполагаются членами данного класса.

Простейшая сигнатура метода расчета значения хэша предполагает использование массива байт в качестве входного значения и массива байт в качестве возвращаемого значения. В случае, если необходимо вычислить хэш-значение от файла, более оптимален подход последовательного чтения файла по блокам в процессе вычисления хэша, а не чтение файла целиком в массив. Для этих целей в разных средах разработки служат специфичные объекты, имеющие похожий функционал. В рассматриваемом случае, для данных целей будет служить класс «Stream», который и будем передавать в качестве аргумента метода. Таким образом, определим два метода для расчета хэша.

ГОСТ Р 34.11-2012 определяет 2 варианта хэш-функции с длиной выходного значения 256 и 512 бит. Алгоритм расчета значения в обоих вариантах схож, и для краткости изложения выделим параметр длины возвращаемого значения в качестве свойства нашего класса. В итоге, приходим к следующему шаблону (Листинг 3.1).

Листинг 3.1 — Схема класса GR3411_2012_Hash

```
1 public class GR3411_2012_Hash
2 {
3     <константы, переменные и вспомогательные методы>
4     public GR3411_2012_Hash(int outputLength) {<конструктор с
          параметром длины>}
5     public byte[] GetHash(byte [] m) {<вычисление жэш-функции >}
6     public byte[] GetHash(Stream st) {<вычисление жэш-функции >}
7 }
```

Далее определим код исполняющий основную схему алгоритма хэширования. Пока представим, что у нас уже определена функция сжатия и она имеет следующую сигнатуру (Листинг 3.2).

Листинг 3.2 — Сигнатура функции сжатия

```
1 //Функция сжатия
2 private byte[] G_n(byte[] N, byte[] h, byte[] m)
```

Реализуем основную схему алгоритма. Также будем группировать операции по трем этапам. Для начала ограничимся случаем вычисления хэша от массива байт. Промежуточные блоки будем хранить в виде массива байт размера 64. Сложение по модулю 512 будем осуществлять с помощью специального метода (Листинг 3.3).

Листинг 3.3 — Метод, осуществляющий сложение по модулю 2^{512}

```
1  //Сложение по модулю 2^512
2  private void AddModulo512(byte[] a, byte[] b, byte[] c)
3  {
4      int i, t = 0;
5      for (i = 0; i < 64; i++)
6      {
7          t = a[i] + b[i] + (t >> 8);
8          c[i] = (byte)(t & 0xFF);
9      }
10 }
```

Итерационным переменным на первом этапе присвоим нулевые значения, за исключением вектора IV в случае длины выходного значения равной 256 битам. Далее будем последовательно копировать из основного массива во вспомогательный блоки согласно шагу алгоритма 2.1 и производить вычисления 2.2-2.7, если это возможно. Для контроля длины оставшейся части введем специальную переменную, значение которой будем уменьшать по мере обработки массива. Далее следует третий этап алгоритма и возврат значения. Код метода представлен в Листинге 3.4.

Листинг 3.4 — Метод, вычисляющий хэш-сумму от массива байт

```
1  //Хешфункция от массива байт
2  public byte[] GetHash(byte[] message)
3  {
4      //Этап 1
5      byte[] paddedMes = new byte[64];
6      byte[] h = new byte[64];
7      byte[] N_0 = new byte[64];
8      byte[] N = new byte[64];
```

```
9       byte[] Sigma = new byte[64];
10      if (outLen == 256)
11      {
12          for (int i = 0; i < 64; i++)
13          {
14              //Вместо использования вектора IV сразу присвоим
                    переменной h его начальное значение
15              h[i] = 0x01;
16          }
17      }
18      byte[] N_512 = new byte[64];
19      Array.Copy(BitConverter.GetBytes(512), 0, N_512, 0, 4);
20      int inc = 0;
21      byte[] tempMes = new byte[64];
22      //Этап 2
23      //Длинна сообщения в битах
24      int len = message.Length * 8;
25      while (len >= 512)
26      {
27          //Копирование текущего блока для дальнейшей обработки
28          Array.Copy(message, inc * 64, tempMes, 0, 64);
29          h = G_n(N, h, tempMes);
30          AddModulo512(N, N_512, N);
31          AddModulo512(Sigma, tempMes, Sigma);
32          len -= 512;
33          inc++;
34      }
35      //Этап 3
36      byte[] message1 = new byte[message.Length - inc * 64];
37      Array.Copy(message, inc * 64, message1, 0, message.Length - inc *
            64);
38      if (message1.Length < 64)
39      {
40          for (int i = message1.Length + 1; i < 64; i++)
41          {
42              paddedMes[i] = 0;
43          }
44          //Добавляем единицу в конец
45          paddedMes[message1.Length] = 0x01;
46          Array.Copy(message1, 0, paddedMes, 0, message1.Length);
47      }
```

```
48      h = G_n(N, h, paddedMes);
49      byte[] MesLen = new byte[64];
50      //Конвертация длины усеченного  сообщения в байты, и запись в
            массив
51      MesLen[0] = (byte)(message1.Length * 8);
52      MesLen[1] = (byte)((message1.Length * 8) >> 8);
53
54      AddModulo512(N, MesLen.ToArray(), N);
55      AddModulo512(Sigma, paddedMes, Sigma);
56      h = G_n(N_0, h, N);
57      h = G_n(N_0, h, Sigma);
58      //Возврат значения
59      if (outLen == 512)
60      {
61          return h;
62      }
63      else
64      {
65          byte[] h256 = new byte[32];
66          Array.Copy(h, 32, h256, 0, 32);
67          return h256;
68      }
69  }
```

Что касается поточного расчета хэш-суммы, то общую схему можно оставить прежней за исключением, непосредственно, чтения из потока и начальной обработки полученных данных. Код поточной реализации представлен в Листинге 3.5.

Листинг 3.5 — Метод, вычисляющий хэш-сумму потоковых данных

```
1  //Поточная реализация хешфункции
2  public byte[] GetHash(Stream st)
3  {
4      //Этап 1
5      byte[] paddedMes = new byte[64];
6      byte[] h = new byte[64];
7      byte[] N_0 = new byte[64];
8      byte[] N = new byte[64];
9      byte[] Sigma = new byte[64];
10     if (outLen == 256)
11     {
12         for (int i = 0; i < 64; i++)
13         {
```

```
            h[i] = 0x01;
        }
    }
    byte[] N_512 = new byte[64];
    Array.Copy(BitConverter.GetBytes(512), 0, N_512, 0, 4);
    byte[] tempMes = new byte[64];
    //Чтение блока из потока
    int blockLen = st.Read(tempMes, 0, 64);
    int progress = 0;
    //Этап 2
    while (blockLen == 64)
    {
        h = G_n(N, h, tempMes);
        AddModulo512(N, N_512, N);
        AddModulo512(Sigma, tempMes, Sigma);
        //Чтение блока из потока
        blockLen = st.Read(tempMes, 0, 64);
        progress++;
        if (progress % 2048 == 0 && Hash1MBitStep != null)
            Hash1MBitStep(progress);
    }
    //Этап 3
    byte[] message1 = new byte[blockLen];
    Array.Copy(tempMes, 0, message1, 0, blockLen);
    if (message1.Length < 64)
    {
        for (int i = message1.Length + 1; i < 64; i++)
        {
            paddedMes[i] = 0;
        }
        paddedMes[message1.Length] = 0x01;
        Array.Copy(message1, 0, paddedMes, 0, message1.Length);
    }
    h = G_n(N, h, paddedMes);
    byte[] MesLen = new byte[64];
    MesLen[0] = (byte)(message1.Length * 8);
    MesLen[1] = (byte)((message1.Length * 8) >> 8);

    AddModulo512(N, MesLen.ToArray(), N);
    AddModulo512(Sigma, paddedMes, Sigma);
    h = G_n(N_0, h, N);
```

```
55      h = G_n(N_0, h, Sigma);
56
57      if (outLen == 512)
58      {
59          return h;
60      }
61      else
62      {
63          byte[] h256 = new byte[32];
64          Array.Copy(h, 32, h256, 0, 32);
65          return h256;
66      }
67  }
```

3.2 Определение констант

Параметры алгоритма включают подстановку бит (2.1), перестановку байт (2.2), матрицу A (2.3) и итерационные константы (2.4). Удобно определять необходимые константы в программном коде, оттуда они будут считанны компилятором. Иной вариант предполагает размещение параметров в отдельном файле, откуда их будет необходимо считать перед вычислением значения. Для простоты изложения выберем первый вариант. Начнем с подстановки байт. Запишем ее в виде массива байт, в котором индекс элемента выполняет роль входного значения, а элемент является выходным значением (Листинг 3.6).

Листинг 3.6 — Массив подстановки байт

```
1  // Подстановка байт
2  private readonly byte[] Sbox ={
3      0xFC, 0xEE, 0xDD, 0x11, 0xCF, 0x6E, 0x31, 0x16, 0xFB, 0xC4, 0xFA,
4      0xDA, 0x23, 0xC5, 0x04, 0x4D, 0xE9, 0x77, 0xF0, 0xDB, 0x93, 0x2E,
5      0x99, 0xBA, 0x17, 0x36, 0xF1, 0xBB, 0x14, 0xCD, 0x5F, 0xC1, 0xF9,
6      0x18, 0x65, 0x5A, 0xE2, 0x5C, 0xEF, 0x21, 0x81, 0x1C, 0x3C, 0x42,
7      0x8B, 0x01, 0x8E, 0x4F, 0x05, 0x84, 0x02, 0xAE, 0xE3, 0x6A, 0x8F,
8      0xA0, 0x06, 0x0B, 0xED, 0x98, 0x7F, 0xD4, 0xD3, 0x1F, 0xEB, 0x34,
9      0x2C, 0x51, 0xEA, 0xC8, 0x48, 0xAB, 0xF2, 0x2A, 0x68, 0xA2, 0xFD,
10     0x3A, 0xCE, 0xCC, 0xB5, 0x70, 0x0E, 0x56, 0x08, 0x0C, 0x76, 0x12,
11     0xBF, 0x72, 0x13, 0x47, 0x9C, 0xB7, 0x5D, 0x87, 0x15, 0xA1, 0x96,
12     0x29, 0x10, 0x7B, 0x9A, 0xC7, 0xF3, 0x91, 0x78, 0x6F, 0x9D, 0x9E,
13     0xB2, 0xB1, 0x32, 0x75, 0x19, 0x3D, 0xFF, 0x35, 0x8A, 0x7E, 0x6D,
14     0x54, 0xC6, 0x80, 0xC3, 0xBD, 0x0D, 0x57, 0xDF, 0xF5, 0x24, 0xA9,
15     0x3E, 0xA8, 0x43, 0xC9, 0xD7, 0x79, 0xD6, 0xF6, 0x7C, 0x22, 0xB9,
```

```
    0x03, 0xE0, 0x0F, 0xEC, 0xDE, 0x7A, 0x94, 0xB0, 0xBC, 0xDC, 0xE8,
    0x28, 0x50, 0x4E, 0x33, 0x0A, 0x4A, 0xA7, 0x97, 0x60, 0x73, 0x1E,
    0x00, 0x62, 0x44, 0x1A, 0xB8, 0x38, 0x82, 0x64, 0x9F, 0x26, 0x41,
    0xAD, 0x45, 0x46, 0x92, 0x27, 0x5E, 0x55, 0x2F, 0x8C, 0xA3, 0xA5,
    0x7D, 0x69, 0xD5, 0x95, 0x3B, 0x07, 0x58, 0xB3, 0x40, 0x86, 0xAC,
    0x1D, 0xF7, 0x30, 0x37, 0x6B, 0xE4, 0x88, 0xD9, 0xE7, 0x89, 0xE1,
    0x1B, 0x83, 0x49, 0x4C, 0x3F, 0xF8, 0xFE, 0x8D, 0x53, 0xAA, 0x90,
    0xCA, 0xD8, 0x85, 0x61, 0x20, 0x71, 0x67, 0xA4, 0x2D, 0x2B, 0x09,
    0x5B, 0xCB, 0x9B, 0x25, 0xD0, 0xBE, 0xE5, 0x6C, 0x52, 0x59, 0xA6,
    0x74, 0xD2, 0xE6, 0xF4, 0xB4, 0xC0, 0xD1, 0x66, 0xAF, 0xC2, 0x39,
    0x4B, 0x63, 0xB6
};
```

Перестановку байт также можно бы было записать в виде массива байт. Однако, перестановка τ (2.2) имеет специфический вид аналогичный транспонированию матрицы 8×8. Это позволяет использовать смещение индекса вместо обращения к массиву. Поэтому массив перестановки байт редко встречается в реализациях. Пока отложим этот момент до следующего пукта, и опустим объявление массива перестановки.

Матрица A имеет размеры 64×64. Операция умножения на матрицу A в L-преобразовании сопряжена с побитовым сложением различных строк матрицы, согласно формуле (2.10). Этот процесс эффективно реализуется путем сложения 64-битных целых чисел, которые удобно представлять в памяти компьютера. В используемом языке данный тип имеет идентификатор «ulong». Таким образом, запишем матрицу A как массив из целых 64-битных чисел, представляющих ее строки (Листинг 3.7).

Листинг 3.7 — Массив строк матрицы A

```
// Матрица A для L функции
private readonly ulong[] A = {
    0x8E20FAA72BA0B470, 0x47107DDD9B505A38, 0xAD08B0E0C3282D1C,
    0xD8045870EF14980E, 0x6C022C38F90A4C07, 0x3601161CF205268D,
    0x1B8E0B0E798C13C8, 0x83478B07B2468764, 0xA011D380818E8F40,
    0x5086E740CE47C920, 0x2843FD2067ADEA10, 0x14AFF010BDD87508,
   0x0AD97808D06CB404, 0x05E23C0468365A02, 0x8C711E02341B2D01,
    0x46B60F011A83988E, 0x90DAB52A387AE76F, 0x486DD4151C3DFDB9,
    0x24B86A840E90F0D2, 0x125C354207487869, 0x092E94218D243CBA,
    0x8A174A9EC8121E5D, 0x4585254F64090FA0, 0xACCC9CA9328A8950,
   0x9D4DF05D5F661451, 0xC0A878A0A1330AA6, 0x60543C50DE970553,
    0x302A1E286FC58CA7, 0x18150F14B9EC46DD, 0x0C84890AD27623E0,
    0x0642CA05693B9F70, 0x0321658CBA93C138, 0x86275DF09CE8AAA8,
```

```
    0x439DA0784E745554, 0xAFC0503C273AA42A, 0xD960281E9D1D5215,
    0xE230140FC0802984, 0x71180A8960409A42, 0xB60C05CA30204D21,
    0x5B068C651810A89E, 0x456C34887A3805B9, 0xAC361A443D1C8CD2,
    0x561B0D22900E4669, 0x2B838811480723BA, 0x9BCF4486248D9F5D,
    0xC3E9224312C8C1A0, 0xEFFA11AF0964EE50, 0xF97D86D98A327728,
    0xE4FA2054A80B329C, 0x727D102A548B194E, 0x39B008152ACB8227,
    0x9258048415EB419D, 0x492C024284FBAEC0, 0xAA16012142F35760,
    0x550B8E9E21F7A530, 0xA48B474F9EF5DC18, 0x70A6A56E2440598E,
    0x3853DC371220A247, 0x1CA76E95091051AD, 0x0EDD37C48A08A6D8,
    0x07E095624504536C, 0x8D70C431AC02A736, 0xC83862965601DD1B,
    0x641C314B2B8EE083
};
```

Для хранения итерационных констант в памяти, можно выделить 2 основных подхода, различающиеся типом используемого массива. Все зависит от того, какой тип применяется для хранения промежуточных значений при расчете функции сжатия. Алгоритм работает с блоками длиной 512 бит, что соответствует массиву байт длиной 64 или массиву 64-битных целых чисел длиной 8. Второй подход, как правило, приводит к некоторому ускорению расчетов за счет применения операций к более емкому типу. В данной работе будем использовать второй вариант. Все 12 констант определим в двумерном массиве 64-битных целых чисел (Листинг 3.8).

Листинг 3.8 — Значения итерационных констант

```
//Итерационные константы
private readonly ulong[][] C = {
    new ulong[8]{ 0xDD806559F2A64507, 0x5767436CC744D23,
        0xA2422A08A460D315, 0x4B7CE09192676901, 0x714EB88D7585C4FC,
        0x2F6A76432E45D016, 0xEBCB2F81C0657C1F, 0xB1085BDA1ECADAE9 },
    new ulong[8]{ 0xE679047021B19BB7, 0x55DDA21BD7CBCD56,
        0x5CB561C2DB0AA7CA, 0x9AB5176B12D69958, 0x61D55E0F16B50131,
        0xF3FEEA720A232B98, 0x4FE39D460F70B5D7, 0x6FA3B58AA99D2F1A },
    new ulong[8]{ 0x991E96F50ABA0AB2, 0xC2B6F443867ADB31,
        0xC1C93A376062DB09, 0xD3E20FE490359EB1, 0xF2EA7514B1297B7B,
        0x6F15E5F529C1F8B, 0xA39FC286A3D8435, 0xF574DCAC2BCE2FC7 },
    new ulong[8]{ 0x220CBEBC84E3D12E, 0x3453EAA193E837F1,
        0xD8B71333935203BE, 0xA9D72C82ED03D675, 0x9D721CAD685E353F,
        0x488E857E335C3C7D, 0xF948E1A05D71E4DD, 0xEF1FDFB3E81566D2 },
    new ulong[8]{ 0x601758FD7C6CFE57, 0x7A56A27EA9EA63F5,
        0xDFFF00B723271A16, 0xBFCD1747253AF5A3, 0x359E35D7800FFFBD,
        0x7F151C1F1686104A, 0x9A3F410C6CA92363, 0x4BEA6BACAD474799 },
    new ulong[8]{ 0xFA68407A46647D6E, 0xBF71C57236904F35,
        0xAF21F66C2BEC6B6, 0xCFFAA6B71C9AB7B4, 0x187F9AB49AF08EC6,
```

```
        0x2D66C4F95142A46C, 0x6FA4C33B7A3039C0, 0xAE4FAEAE1D3AD3D9 },
    new ulong[8]{ 0x8886564D3A14D493, 0x3517454CA23C4AF3,
        0x6476983284A0504, 0x992ABC52D822C37, 0xD3473E33197A93C9,
        0x399EC6C7E6BF87C9, 0x51AC86FEBF240954, 0xF4C70E16EEAAC5EC },
    new ulong[8]{ 0xA47F0DD4BF02E71E, 0x36ACC2355951A8D9,
        0x69D18D2BD1A5C42F, 0xF4892BCB929B0690, 0x89B4443B4DDBC49A,
        0x4EB7F8719C36DE1E, 0x3E7AA020C6E4141, 0x9B1F5B424D93C9A7 },
    new ulong[8]{ 0x7261445183235ADB, 0xE38DC92CB1F2A60,
        0x7B2B8A9AA6079C54, 0x800A440BDBB2CEB1, 0x3CD955B7E00D0984,
        0x3A7D3A1B25894224, 0x944C9AD8EC165FDE, 0x378F5A541631229B },
    new ulong[8]{ 0x74B4C7FB98459CED, 0x3698FAD1153BB6C3,
        0x7A1E6C303B7652F4, 0x9FE76702AF69334B, 0x1FFFE18A1B336103,
        0x8941E71CFF8A78DB, 0x382AE548B2E4F3F3, 0xABBEDEA680056F52 },
    new ulong[8]{ 0x6BCAA4CD81F32D1B, 0xDEA2594AC06FD85D,
        0xEFBACD1D7D476E98, 0x8A1D71EFEA48B9CA, 0x2001802114846679,
        0xD8FA6BBBEBAB0761, 0x3002C6CD635AFE94, 0x7BCD9ED0EFC889FB },
    new ulong[8]{ 0x48BC924AF11BD720, 0xFAF417D5D9B21B99,
        0xE71DA4AA88E12852, 0x5D80EF9D1891CC86, 0xF82012D430219F9B,
        0xCDA43C32BCDF1D77, 0xD21380B00449B17A, 0x378EE767F11631BA }
};
```

3.3 Функция сжатия

Как уже ранее было отмечено, постараемся реализовать функцию сжатия так, чтобы основная часть операций производилась над 64-битными целыми числами. Еще стоит отметить, что выделение новой памяти в программной куче, как правило, является трудоемкой операцией и по возможности его необходимо избежать в функции сжатия. Для этого результаты всех расчетов необходимо записывать в предварительно выделенные области памяти. Для этого, все необходимые переменные можно было бы передать в функцию сжатия в качестве аргументов. Но это придаст сигнатуре громоздкость, которой можно избежать, определив все необходимые переменные во внешней области. В нашем случае их можно определить непосредственно внутри класса. Приведем их список в Листинге 3.9.

Листинг 3.9 — Вспомогательные переменные

```
//Вспомогательные переменные функции сжатия
private byte[] newh = new byte[64];
private ulong[] N64 = new ulong[8];
private ulong[] h64 = new ulong[8];
private ulong[] m64 = new ulong[8];
private ulong[] lResultG = new ulong[8];
```

```
private ulong[] lResultK1 = new ulong[8];
private ulong[] lResultK2 = new ulong[8];
private ulong[] lResultE1 = new ulong[8];
private ulong[] lResultE2 = new ulong[8];
private ulong[] t = new ulong[8];
private ulong[] newh64 = new ulong[8];
private ulong[] state = new ulong[8];
//Матрица предпросчета
private ulong[,] LSPrecalc = new ulong[256, 8];
```

Из сигнатуры функции сжатия, которой мы пользовались ранее, видно, что входные и выходные значения представляют собой массивы байт. Для того, чтобы производить основные операции над 64-битными целыми числами, необходимо иметь возможность переводить массив байт в соответствующий ему массив 64-битных целых чисел и наоборот. Для этих целей выделим 2 специальных метода (Листинг 3.10).

Листинг 3.10 — Методы для преобразования типов

```
//Преобразование массива byte в массив ulong
public void Byte64ToUlong8(byte[] a, ulong[] b)
{
    for (int i = 0; i < 8; i++)
    {
        b[i] = (ulong)a[i * 8] | ((ulong)a[i * 8 + 1] << 8) |
          ((ulong)a[i * 8 + 2] << 16) | ((ulong)a[i * 8 + 3] << 24) |
          ((ulong)a[i * 8 + 4] << 32) | ((ulong)a[i * 8 + 5] << 40) |
          ((ulong)a[i * 8 + 6] << 48) | ((ulong)a[i * 8 + 7] << 56);
    }
}
//Преобразование массива ulong в массив byte
public void Ulong8ToByte64(ulong[] a, byte[] b)
{
    for (int i = 0; i < 64; i++)
    {
        b[i] = (byte)(a[i / 8] >> (i % 8) * 8);
    }
}
```

В ходе вычисления функции сжатия используется побитовое сложение двоичных векторов. Определим данную операцию для массива из 8-ми 64-битных целых чисел. Будем передавать в метод адрес слагаемых и результата. Вместо перебора

индексов в цикле эффективней, с точки зрения времени выполнения, указать все индексы вручную. Код метода представлен в Листинге 3.11.

Листинг 3.11 — Метод для побитового сложения

```
//Побитовое сложение по модулю 2
private void X(ulong[] a, ulong[] b, ulong[] c)
{
    c[0] = a[0] ^ b[0];
    c[1] = a[1] ^ b[1];
    c[2] = a[2] ^ b[2];
    c[3] = a[3] ^ b[3];
    c[4] = a[4] ^ b[4];
    c[5] = a[5] ^ b[5];
    c[6] = a[6] ^ b[6];
    c[7] = a[7] ^ b[7];
}
```

Преобразование $LPSX$ оформим одним методом, принимающим слагаемые X и записывающим результат в третью переменную. Но сначала необходимо определиться с его оптимизацией, заключающейся в использовании массива предпросчета для расчета LS преобразования. Поясним данный момент. Вначале запишем реализацию без использования предпросчета (Листинг 3.12).

Листинг 3.12 — Неоптимизированный вариант $LPSX$-преобразования

```
private void LPSX(ulong[] stateA, ulong[] stateB, ulong[] result)
{
    ulong[] t1 = new ulong[8];
    X(stateA, stateB, t1);
    int i1, tTemp;
    ulong t;
    for (int i = 0; i < 8; i++)//Перебор восьмерок байт
    {
        //Сохранение величины сдвига
        i1 = i * 8;
        t = 0;
        for (int j = 0; j < 8; j++)
        {
            //Получение байта со смещением индекса для реализации
               P—преобразования
            tTemp = Sbox[(byte)(t1[j] >> i1)];
            for (int k = 0; k < 8; k++)
            {
```

```
                    //Разбиение байта на биты
                    if (((tTemp >> (7 - k)) & 0x01) != 0)
                    {
                        //Побитовое сложение строк матрицы
                        t ^= A[(7 - j) * 8 + k];
                    }
                }
            }
            result[i] = t;
        }
}
```

В данном варианте уже используется смещение индекса для реализации P - преобразования. Подробнее об этом приеме написано, например, в [7].

Разберем Листинг 3.12 подробнее. Вначале вычисляется X-преобразование, и его результат записывается во вспомогательный массив. Далее, в цикле происходит перебор индексов предполагаемого вывода. Затем, внутри этого цикла происходит выборка байт со смещением индекса и учетом S-преобразования и побитовое сложение строк матрицы по битам каждого байта, результат которого и записывается в результирующий массив по текущему индексу.

Суть оптимизации заключается в том, что для каждого байта по битам которого складываются строки матрицы A, зная его смещение вдоль 64-битного блока, можно заранее посчитать сумму строк. В дополнение к этому, к каждому из возможных байт можно применить S-преобразование, для того, чтобы сократить дальнейшие расчеты. Так, для хранения всех таких сумм понадобится 8×256 64-битных целых чисел. Все такие значения вычислим однократно и запишем в двумерный массив «LSPrecalc» с помощью специального метода (Листинг 3.13).

Листинг 3.13 — Метод предпросчета

```
private void Precalc()
{
    for (int temp = 0; temp < 256; temp++)//перебор байт
    {
        for (int j = 0; j < 8; j++)
        {
            byte temp1 = Sbox[(byte)temp];
            ulong t1 = 0;
            for (int k = 0; k < 8; k++)
            {
                if (((temp1 >> (7 - k)) & 0x01) != 0)
                {
```

```
                    t1 ^= A[(7 - j) * 8 + k];
                }
            }
            LSPrecalc[temp, j] = t1;
        }
    }
}
```

Метод, описанный в Листинге 3.13, можно вызвать как в конструкторе, так и непосредственно перед вычислением хэша. Ввиду небольшого размера таблицы предпросчета иногда ее целиком размещают в коде программы, опуская определения S и A. Например, такой подход используется в реализации [8]. Вариант расчета LPSX с использованием предпросчета показан в Листинге 3.14.

Листинг 3.14 — Оптимизированный вариант $LPSX$-преобразования

```
private void LPSX(ulong[] stateA, ulong[] stateB, ulong[] result)
{
    ulong
            t1 = stateA[0] ^ stateB[0],
            t2 = stateA[1] ^ stateB[1],
            t3 = stateA[2] ^ stateB[2],
            t4 = stateA[3] ^ stateB[3],
            t5 = stateA[4] ^ stateB[4],
            t6 = stateA[5] ^ stateB[5],
            t7 = stateA[6] ^ stateB[6],
            t8 = stateA[7] ^ stateB[7];
    int i1;
    for (int i = 0; i < 8; i++)
    {
        i1 = i * 8;
        result[i] = LSPrecalc[(byte)(t1 >> i1), 0] ^
        LSPrecalc[(byte)(t2 >> i1), 1] ^
        LSPrecalc[(byte)(t3 >> i1), 2] ^
        LSPrecalc[(byte)(t4 >> i1), 3] ^
        LSPrecalc[(byte)(t5 >> i1), 4] ^
        LSPrecalc[(byte)(t6 >> i1), 5] ^
        LSPrecalc[(byte)(t7 >> i1), 6] ^
        LSPrecalc[(byte)(t8 >> i1), 7];
    }
}
```

Далее остается реализовать преобразование $E(2.12)$, подстановку значений $K_i(2.13)$ и непосредственно расчет функции сжатия $g_N(2.11)$ с поправкой на преобразования типов. Перечисленные методы показаны в Листинге 3.15.

Листинг 3.15 — Расчет функции g_N

```
//Расчет Ki
private ulong[] KeySchedule(ulong[] K, int i)
{
    //Проверка на запись в себя
    if (K.Equals(lResultK1))
    {
        LPSX(K, C[i], lResultK2);
        return lResultK2;
    }
    else
    {
        LPSX(K, C[i], lResultK1);
        return lResultK1;
    }
}

private ulong[] E(ulong[] K, ulong[] m)
{
    Array.Copy(m, state, 8);
    for (int i = 0; i < 12; i++)
    {
        //Проверка на запись в себя
        if (state.Equals(lResultE1))
        {
            LPSX(state, K, lResultE2);
            state = lResultE2;
        }
        else
        {
            LPSX(state, K, lResultE1);
            state = lResultE1;
        }
        K = KeySchedule(K, i);
    }
    X(state, K, state);
    return state;
}
```

```
38
39 private byte[] G_n(byte[] N, byte[] h, byte[] m)
40 {
41     Byte64ToUlong8(N, N64);
42     Byte64ToUlong8(h, h64);
43     Byte64ToUlong8(m, m64);
44     LPSX(h64, N64, lResultG);
45     t = E(lResultG, m64);
46     X(t, h64, t);
47     X(t, m64, newh64);
48     Ulong8ToByte64(newh64, newh);
49     return newh;
50 }
```

В итоге определены все необходимые методы и переменные для расчета хэш-функции по ГОСТ Р 34.11-2012. Полная версия класса с реализацией представлена в приложении А. Примеры расчета различных значений хэша представлены в приложении Б.

4 Третья глава. Цифровая подпись

4.1 Краткий обзор

Механизм цифровой подписи определяется посредством реализации двух основных процессов:

— формирование подписи;
— проверка подписи.

Цифровая подпись предназначена для аутентификации лица, подписывающего электронное сообщение. Кроме того, использование ЭЦП предоставляет возможность обеспечить следующие свойства при передаче в системе подписанного сообщения:

— осуществление контроля целостности передаваемого подписанного сообщения,
— доказательное подтверждение авторства лица,
— защита сообщения от возможной подделки.

Впервые стандарт на цифровую подпись был введен в 1995 году в составе ГОСТ Р 34.10-1994 [9]. Процедуры проверки и генерации цифровой подписи в нем основывались на операциях в поле $\mathbb{Z}_p$. Стойкость его алгоритмов основывалась на сложности решения задачи логарифмирования в $\mathbb{Z}_p$ и стойкости функции хэширования, определенной в ГОСТ Р 34.11-1994. Помимо прочего в стандарте указаны процедуры генерации параметров цифровой подписи.

В следующем стандарте ГОСТ Р 34.10-2001 [10], введеном в 2001 году, вопрос обеспечения стойкости был пересмотрен по сравнению с предыдущим стандартом. Стойкость нового алгоритма основывалась на сложности решения задачи дискретного логарифмирования в группе точек эллиптической кривой. В то же время основная схема алгоритмов осталась прежней, как и используемая хэш-функция. При этом, вопрос генерации параметров в стандарте не поясняется.

Стандарт ГОСТ Р 34.10-2012 [11], введенный в 2012 году унаследовал идеи ГОСТ Р 34.10-2001. Основным отличием ГОСТ Р 34.10-2012 от ГОСТ Р 34.10-2001 является использование в первом новой хэш-функции из ГОСТ Р 34.11-2012. По аналогии с новой хэш-функцией, новый стандарт также предусматривает два варианта процедур генерации и проверки цифровой подписи для длины хэша 256 и 512 бит соответственно. Длина хэша в данном случае сказывается на диапазоне используемых значений параметров.

Стоит отметить, что использование групп точек эллиптической кривой в криптографии является современным и хорошо зарекомендовавшим себя подходом.

Так, например, на тех же механизмах основаны новые версии американского стандарта DSS [12].

4.2 Группа точек эллиптической кривой

В алгоритмах цифровой подписи, описанных в ГОСТ Р 34.11-2012, используется группа точек эллиптической кривой. Введем необходимые определения.

Эллиптической кривой E, определенной над конечным простым полем F_p (где $p > 3$ - простое число), называется множество

$$E = \left\{(x, y) \colon x, y \in F_p \wedge y^2 \equiv x^3 + ax + b \ (mod\ p)\right\}, \tag{4.1}$$

где a, $b \in F_p$ и $4a^3 + 27b^2$ несравнимо с 0 по модулю p.

Инвариантом эллиптической кривой называется величина $J(E)$, удовлетворяющая соотношению

$$J(E) \equiv 1728\frac{4a^3}{4a^3 + 27b^2} \ (mod\ p).$$

Коэффициенты a, b эллиптической кривой E по известному инварианту $J(E)$ определяются формулой

$$\begin{cases} a \equiv 3k \ (mod\ p), \\ b \equiv 2k \ (mod\ p), \end{cases}$$

где $k \equiv \dfrac{J(E)}{1728 - J(E)}(mod\ p)$, $J(E) \neq 0$ или 0,1728.

Пары $(x, y) \in E$ называют точками эллиптической кривой E. Точки будем обозначать как $Q(x, y)$. Точки $Q_1(x_1, y_1)$ и $Q_2(x_2, y_2)$ равны, если равны их соответствующие координаты.

Операцию сложения на E обозначим знаком «+». Для двух точек $Q_1(x_1, y_1)$ и $Q_2(x_2, y_2)$ эллиптической кривой, при вычисления $Q_3(x_3, y_3) = Q_1(x_1, y_1) + Q_2(x_2, y_2)$, рассматривают несколько случаев.

Если $x_1 \neq x_2$, то

$$\begin{cases} x_3 \equiv \lambda^2 - x_1 - x_2 \ (mod\ p), \\ y_3 \equiv \lambda(x_1 - x_3) - y_1 \ (mod\ p), \end{cases} \tag{4.2}$$

где $\lambda \equiv \dfrac{y_2 - y_1}{x_2 - x_1}(mod\ p)$.

Если $x_1 = x_2$ и $y_1 = y_2 \neq 0$, то

$$\begin{cases} x_3 \equiv \lambda^2 - 2x_1 \ (mod\ p), \\ y_3 \equiv \lambda(x_1 - x_3) - y_1 \ (mod\ p), \end{cases} \tag{4.3}$$

где $\lambda \equiv \dfrac{3x_1^2 + a}{2y_1}(mod\, p)$.

Если $x_1 = x_2$ и $y_1 \equiv -y_2(mod\ p)$, то сумма точек Q_1 и Q_2 называется нулевой точкой O без определения ее координат. В этом случае точка Q_2 называется отрицанием Q_1. Для точки O выполнены равенства

$$Q + O = O + Q = Q, \tag{4.4}$$

где $Q \in E$.

Относительно введенной операции сложения множество всех точек E, вместе с нулевой точкой, образуют конечную абелеву группу порядка m.

Точка Q называется «точкой кратности k», если для некоторой точки P выполнено равенство

$$Q = \underbrace{P + \cdots + P}_{k} = kP. \tag{4.5}$$

4.3 Параметры цифровой подписи

Параметры схемы цифровой подписи включают:

— простое число p - модуль эллиптической кривой;

— эллиптическая кривая E, задаваемая своим инвариантом $J(E)$ или коэффициентами $a, b \in F_p$;

— целое число m - порядок группы точек эллиптической кривой E;

— простое число q - порядок циклической подгруппы группы точек эллиптической кривой E, для которого выполнены следующие условия:

$$\begin{cases} m = nq,\, n \in Z, n \geqslant 1 \\ 2^{254} < q < 2^{256} \text{ или } 2^{508} < q < 2^{512} \end{cases};$$

— точка $P \neq O$ эллиптической кривой E, с координатами (x_p, y_p), удовлетворяющая равенству $qP = O$;

— хэш-функция $h(\cdot)\colon V^* \to V_l$, отображающая сообщения, представленные в виде двоичных векторов из V^*, в двоичные векторы длины l бит. Хэш-функция h определена в ГОСТ Р 34.11. При этом, в зависимости от длины хэш-функции, равной 256 или 512 бит, должно выполняться ограничение $2^{254} < q < 2^{256}$ или $2^{508} < q < 2^{512}$ соответственно.

Каждый пользователь схемы цифровой подписи должен обладать личными ключами:

— ключом подписи — целым числом d, удовлетворяющим неравенству $0 < d < q$;

— ключом проверки подписи — точкой эллиптической кривой Q с координатами (x_q, y_q), удовлетворяющей равенству $dP = Q$.

К приведенным параметрам цифровой подписи предъявляют следующие требования:

— должно быть выполнено условие $p^t \neq 1 \ (mod \ p)$, для всех целых $t = 1, 2, \ \ldots \ B$, где $B = 31$, если $2^{254} < q < 2^{256}$, и $B = 131$, если $2^{508} < q < 2^{512}$;

— должно быть выполнено неравенство $m \neq p$;

— инвариант кривой должен удовлетворять условию $J(E) \neq 0{,}1728$.

4.4 Двоичные векторы

В стандарте ГОСТ Р 34.10-2012 особым образом определено соответствие чисел и двоичных векторов длины l.

Рассмотрим двоичный ввектор длины l бит, в котором младшие биты расположены справа, а старшие слева:

$$\overline{h} = (\alpha_{l-1}, \ \ldots, \alpha_0), \overline{h} \in V_l, \tag{4.6}$$

где α_i, $i = 0, \ \ldots, l-1$ равно 1, либо 0.

Число $\alpha \in Z$ соответствует двоичному вектору $\overline{h}$, если выполнено равенство

$$\alpha = \sum_{i=0}^{l-1} \alpha_i 2^i. \tag{4.7}$$

4.5 Формирование цифровой подписи

Для реализации процесса формирования цифровой подписи, должны быть известны параметры схемы цифровой подписи (пункт 4.3). Исходными данными процесса являются ключ d и сообщение M, а выходным результатом — ζ.

Для получения цифровой подписи под сообщением $M \in V^*$ необходимо выполнить следующие шаги по алгоритму I:

Шаг 1 — вычислить хэш-код сообщения M: $\overline{h} = h(M)$.

Шаг 2 — вычислить целое число α, двоичным представлением которого является вектор $\overline{h}$, и определить

$$e \equiv \alpha \ (mod \ q).$$

Если $e = 0$, то определить $e = 1$.

Шаг 3 — сгенерировать случайное целое число k, удовлетворяющее неравенству

$$0 < k < q.$$

Шаг 4 — вычислить точку эллиптической кривой $C = kP$ и определить

$$r \equiv x_c \ (mod \ q),$$

где x_c — x-координата точки C.

Если $r = 0$, то вернуться к шагу 3.

Шаг 5 — вычислить значение

$$s \equiv (rd + ke)(mod\ q).$$

Если $s = 0$, то вернуться к шагу 3.

Шаг 6 — вычислить двоичные векторы $\overline{r}$ и $\overline{s}$, соответствующие r и s, и определить цифровую подпись $\zeta = \overline{r} \| \overline{s}$.

Схема процесса формирования ЭЦП приведена на рисунке 4.1.

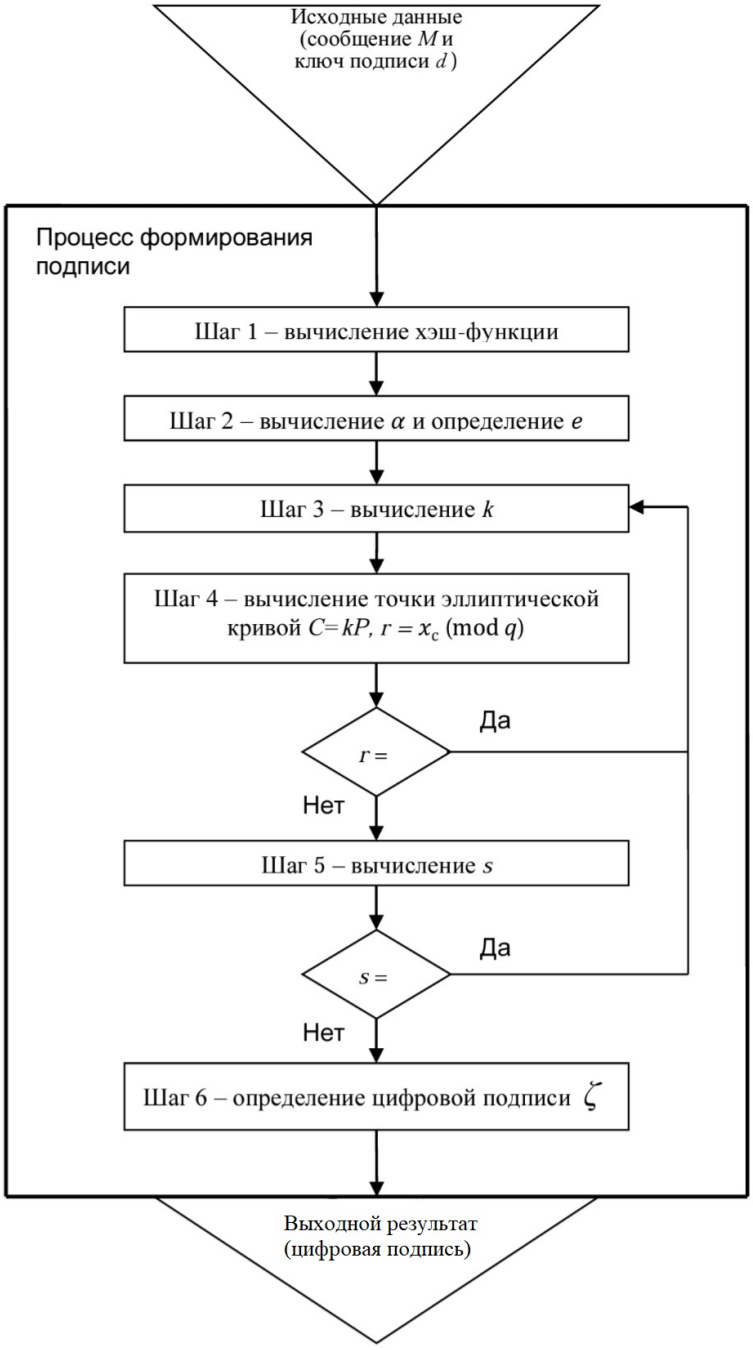

Рисунок 4.1 — Схема процесса формирования ЭЦП

4.6 Проверка подписи

Для реализации процесса проверки цифровой подписи, должны быть известны параметры схемы цифровой подписи (пункт 4.3). Исходными данными процесса являются подписанное сообщение M, цифровая подпись ζ и ключ проверки подписи Q, а выходным — свидетельство достоверности подписи.

Для проверки цифровой подписи ζ под полученным сообщением M необходимо выполнить следующие шаги по алгоритму II:

Шаг 1 — по полученной подписи ζ вычислить целые числа r и s. Если выполнены неравенства $0 < r < q$, $0 < s < q$, то перейти к шагу 2. В противном случае подпись неверна.

Шаг 2 — вычислить хэш-код полученного сообщения M

$$\overline{h} = h(M).$$

Шаг 3 — вычислить целое число α, представлением которого является вектор $\overline{h}$ и определить

$$e \equiv \alpha \ (mod\ q).$$

Если $e = 0$, то определить $e = 1$.

Шаг 4 — вычислить значение $v \equiv e^{-1} \ (mod\ q)$.

Шаг 5 — вычислить значения

$$z_1 \equiv sv \ (mod\ q),\ z_2 \equiv -rv \ (mod\ q).$$

Шаг 6 — вычислить точку эллиптической кривой $C = z_1P + z_2Q$ и определить

$$R \equiv x_c (mod\ q),$$

где x_c — x-координата точки C.

Шаг 7 — если выполнено равенство $R = r$, то подпись принимается, в противном случае - подпись неверна.

Схема процесса проверки ЭЦП приведена на рисунке 4.2.

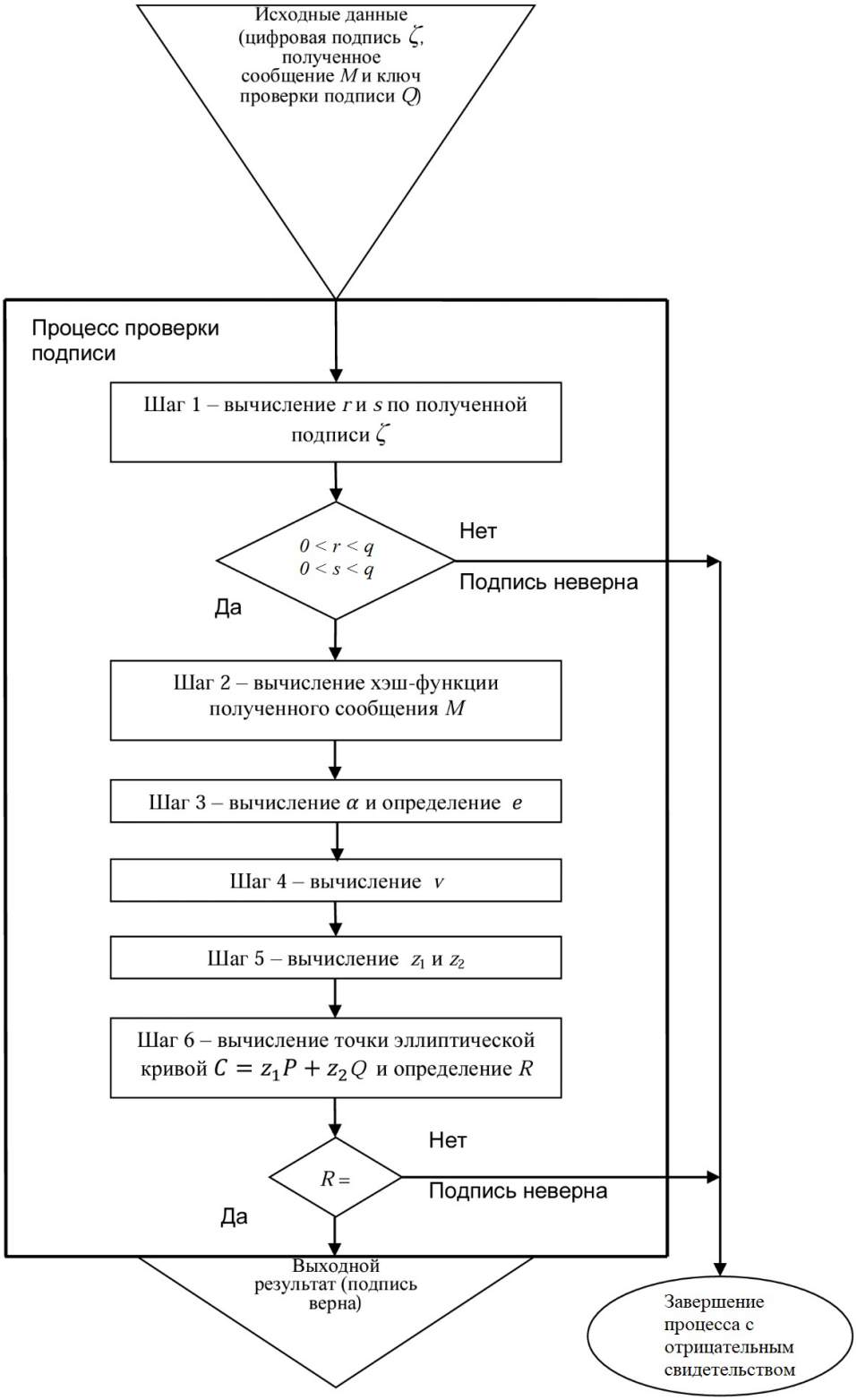

Рисунок 4.2 — Схема процесса проверки ЭЦП

5 Четвертая глава. Программная реализация цифровой подписи

5.1 Арифметические операции

В описанных ранее процессах генерации и проверки подписи часто возникает необходимость производить арифметические операции с большими целыми числами. Сложность оптимальной реализации данных процессов связана с тем, что на большинстве ЭВМ хорошо отлажены арифметические операции на ограниченном наборе типов, которые, как правило, не позволяют напрямую работать с целыми числами, занимающими более 64 бит памяти.

Для реализации арифметики длинных целых чисел (более 64 бит) обычно используют специальные процедуры, работающие с конструкциями замещающими целые числа. Так, большое целое число можно представить в виде массива значений целочисленного типа, каждый элемент которого отождествляется с разрядом этого числа, как если бы оно было записанно в системе счисления с некоторым основанием. Оптимальная реализация процессов умножения или деления таких структур сама по себе является сложной задачей.

Распространенность данной проблемы способствовала созданию готовых библиотек для реализации длинной арифметики в разных языках программирования. В языке C# длинная арифметика представлена классом «BigInteger». Данный класс позволяет работать с большими целыми числами, используя стандартные операторы («+», «-», «*», «/» и «%»).

Стоит отметить одну особенность данного типа, связанную с процессом формирования и проверки ЭЦП. Дело в том, что класс «BigInteger» содержит методы преобразования чисел в массив байт и обратно. Данные методы можно использовать для реализации процессов эквивалентных тем, что описаны в формулах (4.6), (4.7). Основным отличием является то, что в стандарте описан порядок бит от старшего к младшему, в то время как класс «BigInteger» предполагает порядок от младшего к старшему. В данной работе будем использовать порядок бит от младшего к старшему, для избежания лишних инверсий массивов при преобразованиях. Однако, при интеграции с внешними системами, необходимо учесть возможные инверсии.

5.2 Действия в группе точек эллиптической кривой

По аналогии с реализацией функции хэширования, выделим все свойства и методы касающиеся реализации процессов цифровой подписи в отдельный класс — «GR3410_2012_Main». В качестве свойств класса определим параметры схемы цифровой подписи (пункт 4.3) и, дополнительно, добавим переменную, отражающую длину хэша — «L» (Листинг 5.1).

Листинг 5.1 — Параметры схемы цифровой подписи

```
//Параметры эллиптической кривой
private BigInteger p, a, b, m, q;
//Точка P
private BigInteger xp, yp;
//Длина хэша
private int l;
```

Далее определим методы, необходимые для расчета сравнений по модулю. Определим метод, вычисляющий вычет по модулю, метод, вычисляющий обратный элемент по модулю и метод, вычисляющий деление по модулю. Во втором методе, для расчета обратного элемента, используем расширенный алгоритм Евклида. Код методов представлен в Листинге 5.2.

Листинг 5.2 — Методы для расчета сравнений по модулю

```
//Расширенный алгоритм Евклида
private BigInteger[] EGCD(BigInteger a, BigInteger b)
{
    if (b == 0)
        return new BigInteger[] { a, 1, 0 };
    BigInteger[] t = EGCD(b, a % b);
    return new BigInteger[] { t[0], t[2], t[1] - t[2] * (a / b) };
}
//Вычисление обратного к a элемента по модулю p. Возвращает 0 в
   случае необратимости
private BigInteger Inverse(BigInteger a, BigInteger p)
{
    BigInteger[] t = EGCD(a, p);
    if (t[0] != 1)
        return 0;
    return ModP(t[1], p);
}
//Возвращает a по модулю p
private BigInteger ModP(BigInteger a, BigInteger p)
{
    if (a >= 0)
        return a % p;
    else
        return a % p + p;
}
//a/b mod p
private BigInteger DivModP(BigInteger a, BigInteger b, BigInteger p)
```

```
{
    BigInteger c = Inverse(b, p);
    if (c == 0)
        throw new Exception("Деление на необратимый элемент");
    return ModP(a * c, p);
}
```

Каждую точку эллиптической кривой $Q(x,y)$ будем хранить в 2-х целочисленных переменных, отождествляемых с x-координатой и y-координатой точки. Метод, осуществляющий сложение в группе точек эллиптической кривой, определим как процедуру, принимающую координаты слагаемых точек и записывающую результат сложения в координаты третьей точки. В качестве нулевой точки O возьмем точку с координатами $(-1, -1)$. Код метода, осуществляющего сложение в группе точек эллиптической кривой, представлен в Листинге 5.3.

Листинг 5.3 — Метод, осуществляющий сложение в группе точек

```
//Реализует сложение точек (xa, ya) и (xb, yb) эллиптической кривой и
    записывает результат в (xc; yc)
//За бесконечно удаленную точку принимается точка (—1, —1)
private void EllipticSum(BigInteger xa, BigInteger ya, BigInteger xb,
    BigInteger yb, ref BigInteger xc, ref BigInteger yc)
{
    if (xa == —1 && xb == —1)
    {
        xc = —1;
        yc = —1;
    }
    else if (xa == —1)
    {
        xc = xb;
        yc = yb;
    }
    else if (xb == —1)
    {
        xc = xa;
        yc = ya;
    }
    else if (xa != xb)
    {
        BigInteger lambda = DivModP(ModP(yb — ya, p), ModP(xb — xa, p
            ), p);
        BigInteger xct = ModP(lambda * lambda — xa — xb, p);
```

```
24          yc = ModP(lambda * (xa - xct) - ya, p);
25          xc = xct;
26      }
27      else if (xa == xb && ModP(ya - yb, p) == 0)
28      {
29          BigInteger lambda = DivModP(ModP(3 * xa * xa + a, p), ModP(2
                * ya, p), p);
30          BigInteger xct = ModP(lambda * lambda - xa - xa, p);
31          yc = ModP(lambda * (xa - xct) - ya, p);
32          xc = xct;
33      }
34      else
35      {
36          xc = -1;
37          yc = -1;
38      }
39  }
```

Для получения кратных точек (4.5) необходимо реализовать скалярное умножение точки эллиптической кривой на положительное целое число. Умножение должно эффективно работать с большими числами. В связи с этим, необходимо реализовать механизм быстрого умножения на скаляр по следующей формуле

$$kP = k_0P + k_1P^2 + \cdots + k_nP^{2^n}, k = \sum_{i=0}^{i=n} k_i 2^i, k_i \in \{0, 1\}\ i = 0, \ldots, n, \qquad (5.1)$$

где $P \in E(4.1)$, $k \in \mathbb{N}$.

Код метода, осуществляющего умножение на скаляр в группе точек эллиптической кривой, представлен в Листинге 5.4.

Листинг 5.4 — Метод, осуществляющий умножение на скаляр

```
1  //Реализует механизм быстрого скалярного роизведения точки (x, y) на
       k и записывает результат в (x1, y1)
2  private void ScalarProd(BigInteger k, BigInteger x, BigInteger y, ref
       BigInteger x1, ref BigInteger y1)
3  {
4      byte[] kBytes = k.ToByteArray();
5      BigInteger p2x = x, p2y = y;
6      x1 = -1; y1 = -1;
7      for (int i = 0; i < kBytes.Count<byte>(); i++)
8      {
9          for (int j = 0; j < 8; j++)
10         {
```

```
11              if ((kBytes[i] >> j & 0x01) > 0)
12              {
13                  EllipticSum(x1, y1, p2x, p2y, ref x1, ref y1);
14              }
15              EllipticSum(p2x, p2y, p2x, p2y, ref p2x, ref p2y);
16          }
17      }
18  }
```

5.3 Процесс формирования подписи

В процессе формирования подписи необходимо генерировать случайные (псевдослучайные) числа (Шаг 3 алгоритма I). Для генерации псевдослучайных значений в рассматриваемом языке служит класс «Random», экземпляры которого нужно инициализировать некоторым значением (например, текущим системным временем). Определить подобный объект в классе цифровой подписи можно следующим образом (Листинг 5.5).

Листинг 5.5 — Объявление генератора случайных чисел

```
1  //Генератор случайных чисел
2  private Random rand;
3  //Конструктор
4  public GR3410_2012_Main(...)
5  {
6      ...
7      //Инициализация генератора случайных чисел текущим системным
           временем
8      rand = new Random();
9  }
```

Для генерации больших случайных чисел будем использовать специальный метод, принимающий в качестве аргумента целое положительное число p и возвращающий случайное число в диапазоне от 0 до p (Листинг 5.6).

Листинг 5.6 — Метод для генерации случайных чисел

```
//Возвращает случайное число в диапазоне от 0 до p — 1
private BigInteger RandomBelow(BigInteger p)
{
    byte[] bytes = p.ToByteArray();
    rand.NextBytes(bytes);
    bytes[bytes.Length — 1] &= (byte)0x7F;
    return new BigInteger(bytes) % p;
}
```

Завершающий этап процесса генерации подписи предполагает преобразование чисел в двоичные векторы и их последующую конкатенацию (Шаг 6 алгоритма I). Данный процесс выделим в отдельный метод, учитывая специфику реализации (Листинг 5.7).

Листинг 5.7 — Преобразование чисел в двоичные векторы и их конкатенация

```
//Осуществляет перевод в двоичную форму и конкатенацию чисел a и b
private byte[] BinVectorConc(BigInteger a, BigInteger b)
{
    int zetaL = l * 2 / 8;
    byte[] zeta = new byte[zetaL];
    byte[] bytesA = a.ToByteArray();
    byte[] bytesB = b.ToByteArray();
    Array.Copy(bytesA, 0, zeta, 0, Math.Min(bytesA.Length, l / 8));
    Array.Copy(bytesB, 0, zeta, zetaL / 2, Math.Min(bytesB.Length, l
        / 8));
    return zeta;
}
```

Процесс формирования цифровой подписи будем осуществлять в помощью метода, принимающего в качестве аргументов хэш подписываемого сообщения и ключ подписи и возвращающего цифровую подпись. Входные и выходные значения запишем в формате массива байт. Для преобразования массива байт в числа определим специальный метод (Листинг 5.8).

Листинг 5.8 — Преобразование массива байт в соответствующее ему целое число

```
//Преобразует массива байт в соответствующее ему целое число
private BigInteger GetPositive(byte[] bytes)
{
    byte[] bytes1;
    if ((bytes[bytes.Length — 1] & 0x80) > 0)
    {
```

```
7           bytes1 = new byte[bytes.Length + 1];
8           Array.Copy(bytes, bytes1, bytes.Length);
9           bytes1[bytes1.Length - 1] = 0x00;
10      }
11      else
12          bytes1 = bytes;
13      return new BigInteger(bytes1);
14  }
```

Метод, осуществляющий процесс формирования подписи, представлен в Листинге 5.9. Порядок вычислений в описанном в Листинге 5.9 методе аналогичен алгоритму I.

Листинг 5.9 — Метод, осуществляющий процесс формирования подписи

```
1  public byte[] GenerateSign(byte[] hash, byte[] signKey)
2  {
3      BigInteger h = GetPositive(hash);
4      BigInteger d = GetPositive(signKey);
5      BigInteger e = ModP(h, q);
6      if (e == 0)
7          e = 1;
8      BigInteger xc = -1, yc = -1, k, r, s;
9      do
10     {
11         do
12         {
13             k = RandomBelow(q - 1) + 1;
14             ScalarProd(k, xp, yp, ref xc, ref yc);
15             r = ModP(xc, q);
16         }
17         while (r == 0);
18         s = ModP(r * d + k * e, q);
19     }
20     while (s == 0);
21     return BinVectorConc(r, s);
22 }
```

5.4 Процесс проверки подписи

В начале процесса проверки цифровой подписи необходимо получить по имеющийся подписи ζ числа r и s (Шаг 1 алгоритма II). Для этого, определим специальный метод, для получения чисел r и s по известной подписи ζ (Листинг 5.10).

Листинг 5.10 — Получение значений r и s из цифровой подписи

```
//Получение значений r и s из цифровой подписи
private void GetRSFromVector(byte[] zeta, out BigInteger r, out
    BigInteger s)
{
    byte[] bytesR = new byte[l / 8];
    Array.Copy(zeta, 0, bytesR, 0, l / 8);
    r = GetPositive(bytesR);

    byte[] bytesS = new byte[l / 8];
    Array.Copy(zeta, l / 8, bytesS, 0, l / 8);
    s = GetPositive(bytesS);
}
```

Аналогично процессу формирования подписи, процесс проверки цифровой подписи оформим в виде отдельного метода, принимающего хэш подписанного сообщения, цифровую подпись и координаты ключа проверки подписи x_q, y_q. Выходным значением метода, отвечающего за проверку подписи, положим логическое значение, являющееся свидетельством достоверности подписи. Код метода проверки подписи представлен в Листинге 5.11. Порядок вычислений в описанном в Листинге 5.11 методе аналогичен алгоритму II.

Листинг 5.11 — Метод, осуществляющий проверку подписи

```
public bool verifySign(byte[] hash, byte[] sign, byte[] xqb, byte[]
    yqb)
{
    BigInteger r, s;
    GetRSFromVector(sign, out r, out s);

    if (r >= q || s >= q)
        return false;

    BigInteger xq = GetPositive(xqb);
    BigInteger yq = GetPositive(yqb);

    BigInteger h = GetPositive(hash);
    BigInteger e = ModP(h, q);
    if (e == 0)
        e = 1;
    BigInteger v = Inverse(e, q);

```

```
18      BigInteger z1 = ModP(s * v, q);
19      BigInteger z2 = ModP(-r * v, q);
20
21      BigInteger xc = -1, yc = -1, xt = -1, yt = -1;
22      ScalarProd(z1, xp, yp, ref xc, ref yc);
23      ScalarProd(z2, xq, yq, ref xt, ref yt);
24      EllipticSum(xc, yc, xt, yt, ref xc, ref yc);
25      BigInteger R = ModP(xc, q);
26
27      return R == r;
28  }
```

5.5 Хранение параметров и значений

В описанных ранее методах класса все глобальные методы (доступные при работе с классом вне его кода) определены относительно входных и выходных переменных, заданных массивами байт. Данный подход позволяет легко хранить данные параметры в памяти компьютера. Однако стоит заметить, что однотипность всех этих переменных может способствовать возникновению ошибок, связанных с неверным определением назначения той или иной переменной. Со структурной точки зрения более верным подходом являлось бы определение более точных типов для различных параметров алгоритма. Такой подход, в конечном итоге, упростит работу с переменными.

Язык программирования C# определяет механизмы сериализации классов в файлы формата XML, позволяющие хранить в файле не только переменные класса но и его структуру. Такой подход позволяет точно идентифицировать XML-файл как сериализованную версию экземпляра класса, что позволяет избежать подмены типов.

Таким образом, определим ряд классов, отождествленных с различными параметрами процессов формирования и проверки подписи. Выделим следующие группы параметров:

— параметры схемы цифровой подписи;
— ключ подписи;
— ключ проверки подписи;
— цифровая подпись.

Каждой из выделенных групп поставим в соответствие сериализуемый класс. Список классов представлен в Листинге 5.12.

Листинг 5.12 — Список классов, представляющих параметры алгоритмов

```
//Класс параметров схемы цифровой подписи
[Serializable]
public class GR3410_2012_Parameters
{
    public byte[] P { get; set; }
    public byte[] A { get; set; }
    public byte[] B { get; set; }
    public byte[] M { get; set; }
    public byte[] Q { get; set; }
    public byte[] Xp { get; set; }
    public byte[] Yp { get; set; }
    public int L { get; set; }

    public GR3410_2012_Parameters() { }

    public GR3410_2012_Parameters(byte[] p, byte[] a, byte[] b, byte
        [] m, byte[] q, byte[] xp, byte[] yp, int l)
    {
        P = (byte[])p.Clone();
        A = (byte[])a.Clone();
        B = (byte[])b.Clone();
        M = (byte[])m.Clone();
        Q = (byte[])q.Clone();
        Xp = (byte[])xp.Clone();
        Yp = (byte[])yp.Clone();
        L = l;
    }
}
//Класс ключа подписи
[Serializable]
public class GR3410_2012_SignKey
{
    public byte[] D { get; set; }

    public GR3410_2012_SignKey() { }

    public GR3410_2012_SignKey(byte[] d)
    {
        D = (byte[])d.Clone();
    }
```

```
}
//Класс ключа проверки подписи
[Serializable]
public class GR3410_2012_VerifyKey
{
    public byte[] Xq { get; set; }
    public byte[] Yq { get; set; }

    public GR3410_2012_VerifyKey() { }

    public GR3410_2012_VerifyKey(byte[] xq, byte[] yq)
    {
        Xq = (byte[])xq.Clone();
        Yq = (byte[])yq.Clone();
    }
}
//Класс подписи
[Serializable]
public class GR3410_2012_Sign
{
    public byte[] Sign { get; set; }

    public GR3410_2012_Sign() { }

    public GR3410_2012_Sign(byte[] sign)
    {
        Sign = (byte[])sign.Clone();
    }
}
```

Полученные классы для параметров можно использовать в классе «GR3410_2012_Main» при инициализации и при расчете процессов формирования и проверки подписи. Таким образом, определим 2 варианта конструктора класса: с использованием класса параметров схемы и без (Листинг 5.13).

Листинг 5.13 — 2 варианта конструктора класса «GR3410_2012_Main»

```
//Конструктор с параметрами
public GR3410_2012_Main(byte[] p, byte[] a, byte[] b, byte[] m, byte
    [] q, byte[] xp, byte[] yp, int l)
{
    this.p = GetPositive(p);
    this.a = GetPositive(a);
```

```
6       this.b = GetPositive(b);
7       this.m = GetPositive(m);
8       this.q = GetPositive(q);
9       this.xp = GetPositive(xp);
10      this.yp = GetPositive(yp);
11      this.l = l;
12      rand = new Random();
13  }
14  //Другой вариант конструктора с параметрами
15  public GR3410_2012_Main(GR3410_2012_Parameters parameters)
16  {
17      this.p = GetPositive(parameters.P);
18      this.a = GetPositive(parameters.A);
19      this.b = GetPositive(parameters.B);
20      this.m = GetPositive(parameters.M);
21      this.q = GetPositive(parameters.Q);
22      this.xp = GetPositive(parameters.Xp);
23      this.yp = GetPositive(parameters.Yp);
24      this.l = parameters.L;
25      rand = new Random();
26  }
```

Вариант расчета процедур формирования и проверки подписи с использованием выделенных классов параметров представлен в Листинге 5.14.

Листинг 5.14 — Расчет процедур формирования и проверки подписи с использованием выделенных классов параметров

```
1  //Второй вариант генерации подписи
2  public GR3410_2012_Sign GenerateSign(byte[] hash, GR3410_2012_SignKey
       signKey)
3  {
4      return new GR3410_2012_Sign(GenerateSign(hash, signKey.D));
5  }
6  //Второй вариант проверки подписи
7  public bool verifySign(byte[] hash, GR3410_2012_Sign sign,
       GR3410_2012_VerifyKey verifyKey)
8  {
9      return verifySign(hash, sign.Sign, verifyKey.Xq, verifyKey.Yq);
10 }
```

Таким образом, определены все необходимые компоненты для реализации процессов проверки и генерации цифровой подписи согласно ГОСТ Р 34.10-2012.

Полный код класса «GR3410_2012_Main» представлен в приложении В. В качестве практического применения разработанного класса в рамках данной работы был написан ряд вспомогательных программ, осуществляющих редактирование параметров, генерацию подписи и проверку подписи. Пример работы с данными программами представлен в приложении Г.

Заключение

Реализация современных криптографических алгоритмов, безусловно, требует высокой квалификации разработчика. Это связанно с тем, что подобные алгоритмы должны работать как можно более эффективно для обеспечения быстродействия обслуживаемых ими систем. Бывает весьма трудоемко оптимальное распределение переменных по регистрам процессора и уровням кэша. Также, особого рассмотрения требует использование встроенных типов. Подобные проблемы часто бывают узкоспециализированны в зависимости от архитектуры ЭВМ и выливаются в широкий спектр прикладных вопросов.

В данной работе были рассмотрены наиболее общие и надежные высокоуровневые языковые конструкции, призванные сформировать у читателя представление о процессах практического вычисления функции хэширования согласно ГОСТ Р 34.11-2012 и реализации процессов генерации и проверки ЭЦП согласно ГОСТ Р 34.10-2012.

В дополнение к изложенному, в рамках данной работы был разработан ряд приложений для операционной системы Windows с графическим интерфейсом, которые позволяют непосредственно использовать описанные механизмы.

Приложения реализуют:

— вычисление хэш-функции от файла;
— редактирование параметров и ключей схемы цифровой подписи:
— генерацию цифровой подписи;
— проверку цифровой подписи.

Список использованных источников

1. ГОСТ Р 34.11—94. Информационная технология. Криптографическая защита информации. Функция хэширования. — М.: Издательство стандартов, 1994, 16 с.

2. ГОСТ Р 34.11—2012. Информационная технология. Криптографическая защита информации. Функция хэширования. — М.: Стандартинформ, 2013, 25 с.

3. Cryptanalysis of the GOST Hash Function / Christian Rechberger, Marcin Kontak, Janusz Szmidt, Florian Mendel, Norbert Pramstaller [Электронный ресурс]. — URL: `https://online.tu-graz.ac.at/tug_online/voe_main2.getVollText?pDocumentNr=80200&pCurrPk=36649`.

4. *Merkle, R.C.* Secrecy, authentication, and public key systems / R.C. Merkle. — Stanford, CA, USA: Stanford University, 1979.

5. *Miyaguchi S. Ohta K., Iwata M.* 128-bit hash function (NHash) // Proceedings of SECURICOM '90. — 03.1990 / Iwata M. Miyaguchi S., Ohta K. — С. 123—137. — (SECURICOM '90).

6. *Biham E., Dunkelman O.* A Framework for Iterative Hash Functions – HAIFA / Dunkelman O. Biham E. — International Association for Cryptologic Research, 2007.

7. Лебедев П.А. Сравнение старого и нового стандартов РФ на криптографическую хэш-функцию на ЦП и графических процессорах NVIDIA // Матем. вопр. криптогр. — М., 2013. — Т. 4, вып. 2. — С. 73—80.

8. GOST R 34.11-2012: RFC-6986 cryptographic hash function [Электронный ресурс]. — URL: `https://github.com/adegtyarev/streebog`.

9. ГОСТ Р 34.10—94. Информационная технология. Криптографическая защита информации. Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма. — М.: Издательство стандартов, 1994, 14 с.

10. ГОСТ Р 34.10—2001. Информационная технология. Криптографическая защита информации Процессы формирования и проверки электронной цифровой подписи. — М.: Издательство стандартов, 2001, 16 с.

11. ГОСТ Р 34.10—2012. Информационная технология. Криптографическая защита информации Процессы формирования и проверки электронной цифровой подписи. — М.: Стандартинформ, 2013, 20 с.

12. FIPS PUB 186-4 Federal Information Processing Standards Publication. Digital Signature Standard (DSS) [Электронный ресурс]. — URL: `http://dx.doi.org/10.6028/NIST.FIPS.186-4`.

13. *Thomas H. Cormen Charles E. Leiserson, Ronald L. Rivest Clifford Stein.* Introduction to Algorithms 3rd ed / Ronald L. Rivest Clifford Stein Thomas H. Cormen,

Charles E. Leiserson. — L: The MIT Press, 2009.

14. *Эндрю Троелсен, Филипп Джепикс.* Язык программирования C# 6.0 и платформа .NET 4.6 7-е изд. / Филипп Джепикс Эндрю Троелсен. — М. : ООО «И.Д. Вильямс», 2008.

Приложение А Код класса хэш-функции

```
using System;
using System.IO;
using System.Linq;
//Класс реализующий функцию хеширования
namespace GR3411_2012
{
    public class GR3411_2012_Hash
    {
        // Матрица A для L функции
        private readonly ulong[] A = {
            0x8E20FAA72BA0B470, 0x47107DDD9B505A38,
            0xAD08B0E0C3282D1C, 0xD8045870EF14980E,
            0x6C022C38F90A4C07, 0x3601161CF205268D,
            0x1B8E0B0E798C13C8, 0x83478B07B2468764,
            0xA011D380818E8F40, 0x5086E740CE47C920,
            0x2843FD2067ADEA10, 0x14AFF010BDD87508,
            0x0AD97808D06CB404, 0x05E23C0468365A02,
            0x8C711E02341B2D01, 0x46B60F011A83988E,
            0x90DAB52A387AE76F, 0x486DD4151C3DFDB9,
            0x24B86A840E90F0D2, 0x125C354207487869,
            0x092E94218D243CBA, 0x8A174A9EC8121E5D,
            0x4585254F64090FA0, 0xACCC9CA9328A8950,
            0x9D4DF05D5F661451, 0xC0A878A0A1330AA6,
            0x60543C50DE970553, 0x302A1E286FC58CA7,
            0x18150F14B9EC46DD, 0x0C84890AD27623E0,
            0x0642CA05693B9F70, 0x0321658CBA93C138,
            0x86275DF09CE8AAA8, 0x439DA0784E745554,
            0xAFC0503C273AA42A, 0xD960281E9D1D5215,
            0xE230140FC0802984, 0x71180A8960409A42,
            0xB60C05CA30204D21, 0x5B068C651810A89E,
            0x456C34887A3805B9, 0xAC361A443D1C8CD2,
            0x561B0D22900E4669, 0x2B838811480723BA,
            0x9BCF4486248D9F5D, 0xC3E9224312C8C1A0,
            0xEFFA11AF0964EE50, 0xF97D86D98A327728,
            0xE4FA2054A80B329C, 0x727D102A548B194E,
            0x39B008152ACB8227, 0x9258048415EB419D,
            0x492C024284FBAEC0, 0xAA16012142F35760,
            0x550B8E9E21F7A530, 0xA48B474F9EF5DC18,
            0x70A6A56E2440598E, 0x3853DC371220A247,
            0x1CA76E95091051AD, 0x0EDD37C48A08A6D8,
```

```
            0x07E095624504536C, 0x8D70C431AC02A736,
            0xC83862965601DD1B, 0x641C314B2B8EE083
        };
        // Подстановка байт
        private readonly byte[] Sbox ={
            0xFC, 0xEE, 0xDD, 0x11, 0xCF, 0x6E, 0x31, 0x16, 0xFB,
            0xC4, 0xFA, 0xDA, 0x23, 0xC5, 0x04, 0x4D, 0xE9, 0x77,
            0xF0, 0xDB, 0x93, 0x2E, 0x99, 0xBA, 0x17, 0x36, 0xF1,
            0xBB, 0x14, 0xCD, 0x5F, 0xC1, 0xF9, 0x18, 0x65, 0x5A,
            0xE2, 0x5C, 0xEF, 0x21, 0x81, 0x1C, 0x3C, 0x42, 0x8B,
            0x01, 0x8E, 0x4F, 0x05, 0x84, 0x02, 0xAE, 0xE3, 0x6A,
            0x8F, 0xA0, 0x06, 0x0B, 0xED, 0x98, 0x7F, 0xD4, 0xD3,
            0x1F, 0xEB, 0x34, 0x2C, 0x51, 0xEA, 0xC8, 0x48, 0xAB,
            0xF2, 0x2A, 0x68, 0xA2, 0xFD, 0x3A, 0xCE, 0xCC, 0xB5,
            0x70, 0x0E, 0x56, 0x08, 0x0C, 0x76, 0x12, 0xBF, 0x72,
            0x13, 0x47, 0x9C, 0xB7, 0x5D, 0x87, 0x15, 0xA1, 0x96,
            0x29, 0x10, 0x7B, 0x9A, 0xC7, 0xF3, 0x91, 0x78, 0x6F,
            0x9D, 0x9E, 0xB2, 0xB1, 0x32, 0x75, 0x19, 0x3D, 0xFF,
            0x35, 0x8A, 0x7E, 0x6D, 0x54, 0xC6, 0x80, 0xC3, 0xBD,
            0x0D, 0x57, 0xDF, 0xF5, 0x24, 0xA9, 0x3E, 0xA8, 0x43,
            0xC9, 0xD7, 0x79, 0xD6, 0xF6, 0x7C, 0x22, 0xB9, 0x03,
            0xE0, 0x0F, 0xEC, 0xDE, 0x7A, 0x94, 0xB0, 0xBC, 0xDC,
            0xE8, 0x28, 0x50, 0x4E, 0x33, 0x0A, 0x4A, 0xA7, 0x97,
            0x60, 0x73, 0x1E, 0x00, 0x62, 0x44, 0x1A, 0xB8, 0x38,
            0x82, 0x64, 0x9F, 0x26, 0x41, 0xAD, 0x45, 0x46, 0x92,
            0x27, 0x5E, 0x55, 0x2F, 0x8C, 0xA3, 0xA5, 0x7D, 0x69,
            0xD5, 0x95, 0x3B, 0x07, 0x58, 0xB3, 0x40, 0x86, 0xAC,
            0x1D, 0xF7, 0x30, 0x37, 0x6B, 0xE4, 0x88, 0xD9, 0xE7,
            0x89, 0xE1, 0x1B, 0x83, 0x49, 0x4C, 0x3F, 0xF8, 0xFE,
            0x8D, 0x53, 0xAA, 0x90, 0xCA, 0xD8, 0x85, 0x61, 0x20,
            0x71, 0x67, 0xA4, 0x2D, 0x2B, 0x09, 0x5B, 0xCB, 0x9B,
            0x25, 0xD0, 0xBE, 0xE5, 0x6C, 0x52, 0x59, 0xA6, 0x74,
            0xD2, 0xE6, 0xF4, 0xB4, 0xC0, 0xD1, 0x66, 0xAF, 0xC2,
            0x39, 0x4B, 0x63, 0xB6
        };
        //Итерационные константы
        private readonly ulong[][] C = {
            new ulong[8]{
                0xDD806559F2A64507, 0x5767436CC744D23,
                0xA2422A08A460D315, 0x4B7CE09192676901,
                0x714EB88D7585C4FC, 0x2F6A76432E45D016,
```

```
                0xEBCB2F81C0657C1F, 0xB1085BDA1ECADAE9
            },
            new ulong[8]{
                0xE679047021B19BB7, 0x55DDA21BD7CBCD56,
                0x5CB561C2DB0AA7CA, 0x9AB5176B12D69958,
                0x61D55E0F16B50131, 0xF3FEEA720A232B98,
                0x4FE39D460F70B5D7, 0x6FA3B58AA99D2F1A
            },
            new ulong[8]{
                0x991E96F50ABA0AB2, 0xC2B6F443867ADB31,
                0xC1C93A376062DB09, 0xD3E20FE490359EB1,
                0xF2EA7514B1297B7B, 0x6F15E5F529C1F8B,
                0xA39FC286A3D8435, 0xF574DCAC2BCE2FC7
            },
            new ulong[8]{
                0x220CBEBC84E3D12E, 0x3453EAA193E837F1,
                0xD8B71333935203BE, 0xA9D72C82ED03D675,
                0x9D721CAD685E353F, 0x488E857E335C3C7D,
                0xF948E1A05D71E4DD, 0xEF1FDFB3E81566D2
            },
            new ulong[8]{
                0x601758FD7C6CFE57, 0x7A56A27EA9EA63F5,
                0xDFFF00B723271A16, 0xBFCD1747253AF5A3,
                0x359E35D7800FFFBD, 0x7F151C1F1686104A,
                0x9A3F410C6CA92363, 0x4BEA6BACAD474799
            },
            new ulong[8]{
                0xFA68407A46647D6E, 0xBF71C57236904F35,
                0xAF21F66C2BEC6B6, 0xCFFAA6B71C9AB7B4,
                0x187F9AB49AF08EC6, 0x2D66C4F95142A46C,
                0x6FA4C33B7A3039C0, 0xAE4FAEAE1D3AD3D9
            },
            new ulong[8]{
                0x8886564D3A14D493, 0x3517454CA23C4AF3,
                0x6476983284A0504, 0x992ABC52D822C37,
                0xD3473E33197A93C9, 0x399EC6C7E6BF87C9,
                0x51AC86FEBF240954, 0xF4C70E16EEAAC5EC
            },
            new ulong[8]{
                0xA47F0DD4BF02E71E, 0x36ACC2355951A8D9,
                0x69D18D2BD1A5C42F, 0xF4892BCB929B0690,
```

```
123                 0x89B4443B4DDBC49A, 0x4EB7F8719C36DE1E,
124                 0x3E7AA020C6E4141, 0x9B1F5B424D93C9A7
125             },
126             new ulong[8]{
127                 0x7261445183235ADB, 0xE38DC92CB1F2A60,
128                 0x7B2B8A9AA6079C54, 0x800A440BDBB2CEB1,
129                 0x3CD955B7E00D0984, 0x3A7D3A1B25894224,
130                 0x944C9AD8EC165FDE, 0x378F5A541631229B
131             },
132             new ulong[8]{
133                 0x74B4C7FB98459CED, 0x3698FAD1153BB6C3,
134                 0x7A1E6C303B7652F4, 0x9FE76702AF69334B,
135                 0x1FFFE18A1B336103, 0x8941E71CFF8A78DB,
136                 0x382AE548B2E4F3F3, 0xABBEDEA680056F52
137             },
138             new ulong[8]{
139                 0x6BCAA4CD81F32D1B, 0xDEA2594AC06FD85D,
140                 0xEFBACD1D7D476E98, 0x8A1D71EFEA48B9CA,
141                 0x2001802114846679, 0xD8FA6BBBEBAB0761,
142                 0x3002C6CD635AFE94, 0x7BCD9ED0EFC889FB
143             },
144             new ulong[8]{
145                 0x48BC924AF11BD720, 0xFAF417D5D9B21B99,
146                 0xE71DA4AA88E12852, 0x5D80EF9D1891CC86,
147                 0xF82012D430219F9B, 0xCDA43C32BCDF1D77,
148                 0xD21380B00449B17A, 0x378EE767F11631BA
149             }
150         };
151         //Делегат обработчика события прогресса вычисления (для
                обратной связи)
152         public delegate void sendProgressDel(long progress);
153         //Событие отражающее увеличение прогресса вычисления на 1мБит
                (для обратной связи)
154         public event sendProgressDel Hash1MBitStep;
155         //Прогресс вычисления (для обратной связи)
156         public long progress;
157         //Вспомогательные переменные функции сжатия
158         private byte[] newh = new byte[64];
159         private ulong[] N64 = new ulong[8];
160         private ulong[] h64 = new ulong[8];
161         private ulong[] m64 = new ulong[8];
```

```
162         private ulong[] lResultG = new ulong[8];
163         private ulong[] lResultK1 = new ulong[8];
164         private ulong[] lResultK2 = new ulong[8];
165         private ulong[] lResultE1 = new ulong[8];
166         private ulong[] lResultE2 = new ulong[8];
167         private ulong[] t = new ulong[8];
168         private ulong[] newh64 = new ulong[8];
169         private ulong[] state = new ulong[8];
170 
171         //Матрица предпросчета
172         private ulong[,] LSPrecalc = new ulong[256, 8];
173         //Длина хеша
174         public int OutLen { get; private set; }
175         //Конструктор с пареметром длины
176         public GR3411_2012_Hash(int outputLenght)
177         {
178             if (outputLenght != 256 && outputLenght != 512)
179                 throw new Exception("Указанный размер вывода не
                        поддерживается");
180             OutLen = outputLenght;
181             Precalc();
182         }
183         //Предпросчет
184         private void Precalc()
185         {
186             for (int temp = 0; temp < 256; temp++)//перебор байт
187             {
188                 for (int j = 0; j < 8; j++)
189                 {
190                     byte temp1 = Sbox[(byte)temp];
191                     ulong t1 = 0;
192                     for (int k = 0; k < 8; k++)
193                     {
194                         if (((temp1 >> (7 - k)) & 0x01) != 0)
195                         {
196                             t1 ^= A[(7 - j) * 8 + k];
197                         }
198                     }
199                     LSPrecalc[temp, j] = t1;
200                 }
201             }
```

```
202         }
203         //Сложение по модулю 2^512
204         private void AddModulo512(byte[] a, byte[] b, byte[] c)
205         {
206             int i, t = 0;
207             for (i = 0; i < 64; i++)
208             {
209                 t = a[i] + b[i] + (t >> 8);
210                 c[i] = (byte)(t & 0xFF);
211             }
212         }
213         //Побитовое сложение по модулю 2 (X)
214         private void X(ulong[] a, ulong[] b, ulong[] c)
215         {
216             c[0] = a[0] ^ b[0];
217             c[1] = a[1] ^ b[1];
218             c[2] = a[2] ^ b[2];
219             c[3] = a[3] ^ b[3];
220             c[4] = a[4] ^ b[4];
221             c[5] = a[5] ^ b[5];
222             c[6] = a[6] ^ b[6];
223             c[7] = a[7] ^ b[7];
224         }
225
226         private void LPSX(ulong[] stateA, ulong[] stateB, ulong[]
                result)
227         {
228             ulong
229             t1 = stateA[0] ^ stateB[0],
230             t2 = stateA[1] ^ stateB[1],
231             t3 = stateA[2] ^ stateB[2],
232             t4 = stateA[3] ^ stateB[3],
233             t5 = stateA[4] ^ stateB[4],
234             t6 = stateA[5] ^ stateB[5],
235             t7 = stateA[6] ^ stateB[6],
236             t8 = stateA[7] ^ stateB[7];
237             int i1;
238             for (int i = 0; i < 8; i++)//перебор восьмерок байт
239             {
240                 i1 = i * 8;
241                 //Преобразование P осуществляется смещением индекса
```

```
242                 result[i] = LSPrecalc[(byte)(t1 >> i1), 0] ^
243                 LSPrecalc[(byte)(t2 >> i1), 1] ^
244                 LSPrecalc[(byte)(t3 >> i1), 2] ^
245                 LSPrecalc[(byte)(t4 >> i1), 3] ^
246                 LSPrecalc[(byte)(t5 >> i1), 4] ^
247                 LSPrecalc[(byte)(t6 >> i1), 5] ^
248                 LSPrecalc[(byte)(t7 >> i1), 6] ^
249                 LSPrecalc[(byte)(t8 >> i1), 7];
250             }
251         }
252 
253         private ulong[] KeySchedule(ulong[] K, int i)
254         {
255             //Проверка на запись в себя
256             if (K.Equals(lResultK1))
257             {
258                 LPSX(K, C[i], lResultK2);
259                 return lResultK2;
260             }
261             else
262             {
263                 LPSX(K, C[i], lResultK1);
264                 return lResultK1;
265             }
266         }
267 
268         private ulong[] E(ulong[] K, ulong[] m)
269         {
270             Array.Copy(m, state, 8);
271             for (int i = 0; i < 12; i++)
272             {
273                 //Проверка на запись в себя
274                 if (state.Equals(lResultE1))
275                 {
276                     LPSX(state, K, lResultE2);
277                     state = lResultE2;
278                 }
279                 else
280                 {
281                     LPSX(state, K, lResultE1);
282                     state = lResultE1;
```

```
283                 }
284                 K = KeySchedule(K, i);
285             }
286             X(state, K, state);
287             return state;
288         }
289         //Функция сжатия
290         private byte[] G_n(byte[] N, byte[] h, byte[] m)
291         {
292             Byte64ToUlong8(N, N64);
293             Byte64ToUlong8(h, h64);
294             Byte64ToUlong8(m, m64);
295             LPSX(h64, N64, lResultG);
296             t = E(lResultG, m64);
297             X(t, h64, t);
298             X(t, m64, newh64);
299             Ulong8ToByte64(newh64, newh);
300             return newh;
301         }
302         //Преобразование массива byte в массив ulong
303         public void Byte64ToUlong8(byte[] a, ulong[] b)
304         {
305             for (int i = 0; i < 8; i++)
306             {
307                 b[i] = (ulong)a[i * 8] | ((ulong)a[i * 8 + 1] << 8) |
308                     ((ulong)a[i * 8 + 2] << 16) | ((ulong)a[i * 8 +
                        3] << 24) |
309                     ((ulong)a[i * 8 + 4] << 32) | ((ulong)a[i * 8 +
                        5] << 40) |
310                     ((ulong)a[i * 8 + 6] << 48) | ((ulong)a[i * 8 +
                        7] << 56);
311             }
312         }
313         //Преобразование массива ulong в массив byte
314         public void Ulong8ToByte64(ulong[] a, byte[] b)
315         {
316             for (int i = 0; i < 64; i++)
317             {
318                 b[i] = (byte)(a[i / 8] >> (i % 8) * 8);
319             }
320         }
```

```
321         //Хешфункция от массива байт
322         public byte[] GetHash(byte[] message)
323         {
324             //Этап 1
325             byte[] paddedMes = new byte[64];
326             byte[] h = new byte[64];
327             byte[] N_0 = new byte[64];
328             byte[] N = new byte[64];
329             byte[] Sigma = new byte[64];
330             if (OutLen == 256)
331             {
332                 for (int i = 0; i < 64; i++)
333                 {
334                     h[i] = 0x01;
335                 }
336             }
337             byte[] N_512 = new byte[64];
338             Array.Copy(BitConverter.GetBytes(512), 0, N_512, 0, 4);
339             int inc = 0;
340             byte[] tempMes = new byte[64];
341             int progress = 0;
342             //Этап 2
343             int len = message.Length * 8;
344             while (len >= 512)
345             {
346                 Array.Copy(message, inc * 64, tempMes, 0, 64);
347                 h = G_n(N, h, tempMes);
348                 AddModulo512(N, N_512, N);
349                 AddModulo512(Sigma, tempMes, Sigma);
350                 len -= 512;
351                 inc++;
352                 progress++;
353                 if (progress % 2048 == 0 && Hash1MBitStep != null)
354                     Hash1MBitStep(progress);
355             }
356             //Этап 3
357             byte[] message1 = new byte[message.Length - inc * 64];
358             Array.Copy(message, inc * 64, message1, 0, message.Length
                    - inc * 64);
359             if (message1.Length < 64)
360             {
```

```
361                 for (int i = message1.Length + 1; i < 64; i++)
362                 {
363                     paddedMes[i] = 0;
364                 }
365                 paddedMes[message1.Length] = 0x01;
366                 Array.Copy(message1, 0, paddedMes, 0, message1.Length
                        );
367             }
368             h = G_n(N, h, paddedMes);
369             byte[] MesLen = new byte[64];
370             //Конвертация длины усеченного  сообщения в байты, и
                    запись в массив
371             MesLen[0] = (byte)(message1.Length * 8);
372             MesLen[1] = (byte)((message1.Length * 8) >> 8);
373 
374             AddModulo512(N, MesLen.ToArray(), N);
375             AddModulo512(Sigma, paddedMes, Sigma);
376             h = G_n(N_0, h, N);
377             h = G_n(N_0, h, Sigma);
378 
379             if (OutLen == 512)
380             {
381                 return h;
382             }
383             else
384             {
385                 byte[] h256 = new byte[32];
386                 Array.Copy(h, 32, h256, 0, 32);
387                 return h256;
388             }
389         }
390 
391         //Поточная реализация хешфункции
392         public byte[] GetHash(Stream st)
393         {
394             //Этап 1
395             byte[] paddedMes = new byte[64];
396             byte[] h = new byte[64];
397             byte[] N_0 = new byte[64];
398             byte[] N = new byte[64];
399             byte[] Sigma = new byte[64];
```

```
400             if (OutLen == 256)
401             {
402                 for (int i = 0; i < 64; i++)
403                 {
404                     h[i] = 0x01;
405                 }
406             }
407             byte[] N_512 = new byte[64];
408             Array.Copy(BitConverter.GetBytes(512), 0, N_512, 0, 4);
409             byte[] tempMes = new byte[64];
410             int blockLen = st.Read(tempMes, 0, 64);
411             int progress = 0;
412             //Этап 2
413             while (blockLen == 64)
414             {
415                 h = G_n(N, h, tempMes);
416                 AddModulo512(N, N_512, N);
417                 AddModulo512(Sigma, tempMes, Sigma);
418                 blockLen = st.Read(tempMes, 0, 64);
419                 progress++;
420                 if (progress % 2048 == 0 && Hash1MBitStep != null)
421                     Hash1MBitStep(progress);
422             }
423             //Этап 3
424             byte[] message1 = new byte[blockLen];
425             Array.Copy(tempMes, 0, message1, 0, blockLen);
426             if (message1.Length < 64)
427             {
428                 for (int i = message1.Length + 1; i < 64; i++)
429                 {
430                     paddedMes[i] = 0;
431                 }
432                 paddedMes[message1.Length] = 0x01;
433                 Array.Copy(message1, 0, paddedMes, 0, message1.Length
                        );
434             }
435             h = G_n(N, h, paddedMes);
436             byte[] MesLen = new byte[64];
437             //Конвертация длины усеченного  сообщения в байты, и
                запись в массив
438             MesLen[0] = (byte)(message1.Length * 8);
```

```
439                 MesLen[1] = (byte)((message1.Length * 8) >> 8);
440
441                 AddModulo512(N, MesLen.ToArray(), N);
442                 AddModulo512(Sigma, paddedMes, Sigma);
443                 h = G_n(N_0, h, N);
444                 h = G_n(N_0, h, Sigma);
445
446                 if (OutLen == 512)
447                 {
448                     return h;
449                 }
450                 else
451                 {
452                     byte[] h256 = new byte[32];
453                     Array.Copy(h, 32, h256, 0, 32);
454                     return h256;
455                 }
456             }
457         }
458 }
```

Приложение Б Примеры расчета хэша

В рамках данной работы, для вычисления хэша файлов, была создана программа с графическим интерфейсом. Скриншот окна программы представлен на Рисунке Б.1.

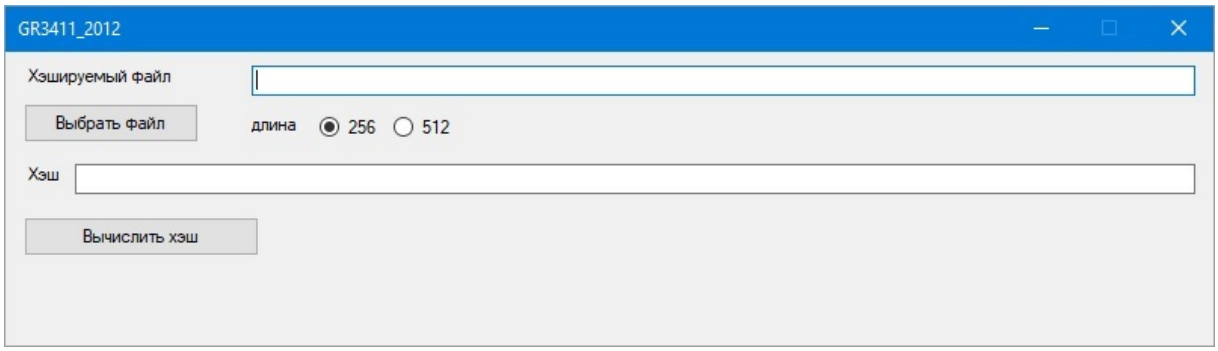

Рисунок Б.1 — Окно программы для вычисления хэша файлов

В окне, представленном на Рисунке Б.1, можно задать имя хэшируемого файла и длину хэша (256 или 512). Для запуска процесса вычисления хэша необходимо нажать кнопку «Вычислить хэш».

Далее представлены результата расчета хэша от некоторых файлов, заданных набором байт. Байты каждого файла и результата расчета хэша представлены $A \in V_{4n}$ и записаны в формате $a_0,\ a_1\ \ldots\ ,a_{n-1}$, где $a_i \in \mathbb{Z}_{16}$, $i = 0,\ 2,\ \ldots,\ n-1$. При этом, A есть $Vec_4(a_0)\| \ldots \|Vec_4(a_{n-1})$. Байты исходного файла обозначим как F. Значение хэша размера 256 бит обозначим как H_{256}. Значение хэша размера 512 бит обозначим как H_{512}.

Пример 1:

$F =$ 303132333435363738393031323334353637383930313233343536373839303132333435363738393031323334353637383930313233343536373839303132333435363738393031323334

$H_{256} =$ 9D151EEFD8590B89DAA6BA6CB74AF9275DD051026BB149A452FD84E5E57B5500

$H_{512} =$ 1B54D01A4AF5B9D5CC3D86D68D285462B19ABC2475222F35C085122BE4BA1FFA00AD30F8767B3A82384C6574F024C311E2A481332B08EF7F41797891C1646F48

Пример 2:

$F =$ D1E520E2E5F2F0E82C20D1F2F0E8E1EEE6E820E2EDF3F6E82C20E2E5FEF2FA20F120ECEEF0FF20F1F2F0E5EBE0ECE820EDE020F5F0E0E1F0FBFF20EFEBFAEAFB20C8E3EEF0E5E2FB

$H_{256} =$ 9DD2FE4E90409E5DA87F53976D7405B0C0CAC628FC669A741D50063C557E8F50

$H_{512} =$ 1E88E62226BFCA6F9994F1F2D51569E0DAF8475A3B0FE61A5300EEE46D961376035FE83549ADA2B8620FCD7C496CE5B33F0CB9DDDC2B6460143B03DABAC9FB28

Пример 3:

$F =$ 4FEF841A4CD8ED40C398991B0B18CFF540B766FC56A590BA633C47F97ADA4A83943140C527AD134A7C3E45DCC9D2BC81D052CC35408B0BAA5EC9B71090F4C5E84D1687D9D809CE92A41C80175FB3F19080C528270E1442082695675DF603B988785C8EB24BE2A9C7843991230B

F34BABD8268FCACB9DDA38D37EB342A2709CA063F79F02243E459E1C61A3D5831FFD334A

H_{256} = 02B5722E1375A8E64351E36DE113A131059FAC8E42F33EB48277D6C3B98840BE

H_{512} = DCB9FE9BB8D270771DFF3DAEEBB48D7E86D17600DCEBA5240BC87297D093C1FCC1BE
DCEFCD79C34EB6A300FE40D9BE366AE3A81A78AFFC070401B8C6D8E7FA86

Пример 4:

F = 84D074EEBCDB8255DECC2E6624139CF1C01354488699E0F1AE6AF904B5B48180F7E6EC
CB2C743BC37FCF60796FB53D9038925EE9DF0135EBEF824052E17DA0294AAE6E3D27D1C54C
D890C8B7396781F82D69CC235698A0F5B100DCA934E8E2022E362E

H_{256} = D4F3E5DEAB479E26B40FA8ACACFC4E2F509732C1B5AF92D437BD3352E328A56E

H_{512} = AE9E06B039BCE6F640ADA7CB7B18D344C39BBD84F1812B84CF0CC23A58D585123B6A
E7AE543B1649C9DD548EF45AADA07F5B1E1D08DBA83532DD5D1D7459D747

Пример 5:

F = 47FF6967CC12114E397EB18800472578B07CD3B20D788B26F51B17EFADC6880338C0B1
2DFF6556438651BC334F53A8EDC36A51B2627CB3CE734272B0FA21A172F5AE4F4095F8784B
25902BBBBC58D7A1732A2C4375D6A0BC0D

H_{256} = 49FA7153B608E6E38E6E4F5058E02E598F6912CB70C7B5F07D363643BCB5F93E

H_{512} = 44A6329382D6970D77ABD8F49CC11E3CA82E8F845F87684EA18981A9F27E87B729BC
28174641521E1F4AE200051BD7EA90879553950D1B46D9662ECE59FEB78C

Пример 6:

F = B9BCF3A54965B04A3DA1046857CBF1058CD250F7121A6BD13DCFF55DFC1F0D9232B23E
BB577F46C60A711B369C0BD361516F49CE6B15D19E512268796F524D1D937235A58C36AD0C
8B8AECA0FF353432FC940C8DD63E261DE0A06A3517CD56B724FEA2

H_{256} = 5D2759000A1F5B46A6CADFF3D04D5F590A475DAB00F5C6BB697D528697CDB67E

H_{512} = 8EE49C5B382AA3F48334D95258DEED51BC0BC682243CAAAF1562EA101C888A01FFEA
F365CCCB16A40CA2445C16C22B1D9A2DF8B4B32DA932F034197CFD0F09FE

Пример 7:

F = D62ED41AC13D43E028309F1BED367DDA46C047554B17C4ECCC78F4BE33357C00930B5C
CEBFF1CB98FB45F4B0004F47E1507C7163B74DDEBE69E84E494885ADF3B39E87E2DAAF1A55
F4AE59052214EB7D30136EB7C31849462C9D64D7A2824F7698

H_{256} = 5C11C68851E700EB385EA64225E8910972CBBD08E59B98BC0C2B75C7ACFCBCA0

H_{512} = 381F13407000C4A931BF3535831826E354D17BBDCFE070A46BC3D71C62393A5A5270
3B3D9A0F99CE42260F0F89224003D238C45F9AEBF4FE25F9DE4CA82F8CE6

Пример 8:

F = FC62C76507A0611B6F08F3D89B55FDD1183517F52F9B5441B53C4A43B2725EE36C2117
88B5CB0CCD56A7A9415254020ED9D1691A5FA3D1610D85AFCA551D69FCA90F6B793D751B4C
29499E87F113EE613A3C6AE1084484C94FB67B301C4177383E

H_{256} = 50F4C7486A94E5D881943829D03C4647186C384AB03ED12A200E114CE650AE6B

H_{512} = 0CF1E8DB676B4219389D6A033C39327402EB38A45EF9166B901DE6E8889C26E06AD7
EE8A996EE7777E9FDA65A5201098E3D4C8AD2370CB40DF13F5DC05905FB7

Приложение В Код класса цифровой подписи

```
using System;
using System.Linq;
using System.Numerics;


namespace GR3410_2012
{
    //Основной класс цифровой подписи
    public class GR3410_2012_Main
    {
        //Параметры эллиптической кривой
        private BigInteger p, a, b, m, q;
        //Точка P
        private BigInteger xp, yp;
        //Длина хэша
        private int l;
        //Генератор случайных чисел
        private Random rand;
        //Конструктор с параметрами
        public GR3410_2012_Main(byte[] p, byte[] a, byte[] b, byte[]
            m, byte[] q, byte[] xp, byte[] yp, int l)
        {
            this.p = GetPositive(p);
            this.a = GetPositive(a);
            this.b = GetPositive(b);
            this.m = GetPositive(m);
            this.q = GetPositive(q);
            this.xp = GetPositive(xp);
            this.yp = GetPositive(yp);
            this.l = l;
            rand = new Random();
        }
        //Другой вариант конструктора с параметрами
        public GR3410_2012_Main(GR3410_2012_Parameters parameters)
        {
            this.p = GetPositive(parameters.P);
            this.a = GetPositive(parameters.A);
            this.b = GetPositive(parameters.B);
            this.m = GetPositive(parameters.M);
            this.q = GetPositive(parameters.Q);
```

```
40              this.xp = GetPositive(parameters.Xp);
41              this.yp = GetPositive(parameters.Yp);
42              this.l = parameters.L;
43              rand = new Random();
44          }
45          //Расширенный алгоритм Евклида
46          private BigInteger[] EGCD(BigInteger a, BigInteger b)
47          {
48              if (b == 0)
49                  return new BigInteger[] { a, 1, 0 };
50              BigInteger[] t = EGCD(b, a % b);
51              return new BigInteger[] { t[0], t[2], t[1] - t[2] * (a /
                    b) };
52          }
53          //Вычисление обратного к a элемента по модулю p. Возвращает 0
                в случае необратимости
54          private BigInteger Inverse(BigInteger a, BigInteger p)
55          {
56              BigInteger[] t = EGCD(a, p);
57              if (t[0] != 1)
58                  return 0;
59              return ModP(t[1], p);
60          }
61          //Возвращает a по модулю p
62          private BigInteger ModP(BigInteger a, BigInteger p)
63          {
64              if (a >= 0)
65                  return a % p;
66              else
67                  return a % p + p;
68          }
69          //a/b mod p
70          private BigInteger DivModP(BigInteger a, BigInteger b,
                BigInteger p)
71          {
72              BigInteger c = Inverse(b, p);
73              if (c == 0)
74                  throw new Exception("Деление на необратимый элемент")
                        ;
75              return ModP(a * c, p);
76          }
```

```
77          //Реализует сложение точек (xa, ya) и (xb, yb) эллиптической
                кривой и записывает результат в (xc, yc)
78          //За бесконечно удаленную точку принимается точка (-1, -1)
79          private void EllipticSum(BigInteger xa, BigInteger ya,
                BigInteger xb, BigInteger yb, ref BigInteger xc, ref
                BigInteger yc)
80          {
81              if (xa == -1 && xb == -1)
82              {
83                  xc = -1;
84                  yc = -1;
85              }
86              else if (xa == -1)
87              {
88                  xc = xb;
89                  yc = yb;
90              }
91              else if (xb == -1)
92              {
93                  xc = xa;
94                  yc = ya;
95              }
96              else if (xa != xb)
97              {
98                  BigInteger lambda = DivModP(ModP(yb - ya, p), ModP(xb
                        - xa, p), p);
99                  BigInteger xct = ModP(lambda * lambda - xa - xb, p);
100                 yc = ModP(lambda * (xa - xct) - ya, p);
101                 xc = xct;
102             }
103             else if (xa == xb && ModP(ya - yb, p) == 0)
104             {
105                 BigInteger lambda = DivModP(ModP(3 * xa * xa + a, p),
                        ModP(2 * ya, p), p);
106                 BigInteger xct = ModP(lambda * lambda - xa - xa, p);
107                 yc = ModP(lambda * (xa - xct) - ya, p);
108                 xc = xct;
109             }
110             else
111             {
112                 xc = -1;
```

```
113                 yc = -1;
114             }
115         }
116         //Реализует механизм быстрого скалярного роизведения точки (x
                , y) на k и записывает результат в (x1, y1)
117         private void ScalarProd(BigInteger k, BigInteger x,
                BigInteger y, ref BigInteger x1, ref BigInteger y1)
118         {
119             byte[] kBytes = k.ToByteArray();
120             BigInteger p2x = x, p2y = y;
121             x1 = -1; y1 = -1;
122             for (int i = 0; i < kBytes.Count<byte>(); i++)
123             {
124                 for (int j = 0; j < 8; j++)
125                 {
126                     if ((kBytes[i] >> j & 0x01) > 0)
127                     {
128                         EllipticSum(x1, y1, p2x, p2y, ref x1, ref y1)
                            ;
129                     }
130                     EllipticSum(p2x, p2y, p2x, p2y, ref p2x, ref p2y)
                        ;
131                 }
132             }
133         }
134         //Возвращает случайное число в диапазоне от 0 до p - 1
135         private BigInteger RandomBelow(BigInteger p)
136         {
137             byte[] bytes = p.ToByteArray();
138             rand.NextBytes(bytes);
139             bytes[bytes.Length - 1] &= (byte)0x7F;
140             return new BigInteger(bytes) % p;
141         }
142         //Преобразует массива байт в соответствующее ему целое число
143         private BigInteger GetPositive(byte[] bytes)
144         {
145             byte[] bytes1;
146             if ((bytes[bytes.Length - 1] & 0x80) > 0)
147             {
148                 bytes1 = new byte[bytes.Length + 1];
149                 Array.Copy(bytes, bytes1, bytes.Length);
```

```
150                 bytes1[bytes1.Length - 1] = 0x00;
151             }
152             else
153                 bytes1 = bytes;
154             return new BigInteger(bytes1);
155         }
156         //Осуществляет перевод в двоичную форму и конкатенацию чисел
               a и b
157         private byte[] BinVectorConc(BigInteger a, BigInteger b)
158         {
159             int zetaL = l * 2 / 8;
160             byte[] zeta = new byte[zetaL];
161             byte[] bytesA = a.ToByteArray();
162             byte[] bytesB = b.ToByteArray();
163             Array.Copy(bytesA, 0, zeta, 0, Math.Min(bytesA.Length, l
                   / 8));
164             Array.Copy(bytesB, 0, zeta, zetaL / 2, Math.Min(bytesB.
                   Length, l / 8));
165             return zeta;
166         }
167         //Получение значений r и s из цифровой подписи
168         private void GetRSFromVector(byte[] zeta, out BigInteger r,
               out BigInteger s)
169         {
170             byte[] bytesR = new byte[l / 8];
171             Array.Copy(zeta, 0, bytesR, 0, l / 8);
172             r = GetPositive(bytesR);
173 
174             byte[] bytesS = new byte[l / 8];
175             Array.Copy(zeta, l / 8, bytesS, 0, l / 8);
176             s = GetPositive(bytesS);
177         }
178         //Генерирует открытый и закрытый ключи
179         public void GenerateKeys(out GR3410_2012_SignKey signKey, out
               GR3410_2012_VerifyKey verifyKey)
180         {
181             BigInteger xc = -1, yc = -1, k, r;
182             do
183             {
184                 k = RandomBelow(q - 1) + 1;
185                 ScalarProd(k, xp, yp, ref xc, ref yc);
```

```
186                 r = ModP(xc, q);
187             }
188             while (r == 0);
189             signKey = new GR3410_2012_SignKey(k.ToByteArray());
190             verifyKey = new GR3410_2012_VerifyKey(xc.ToByteArray(),
                    yc.ToByteArray());
191         }
192         //Второй вариант генерации подписи
193         public GR3410_2012_Sign GenerateSign(byte[] hash,
                GR3410_2012_SignKey signKey)
194         {
195             return new GR3410_2012_Sign(GenerateSign(hash, signKey.D)
                    );
196         }
197         //Первый вариант генерации подписи
198         public byte[] GenerateSign(byte[] hash, byte[] signKey)
199         {
200             BigInteger h = GetPositive(hash);
201             BigInteger d = GetPositive(signKey);
202             BigInteger e = ModP(h, q);
203             if (e == 0)
204                 e = 1;
205             BigInteger xc = -1, yc = -1, k, r, s;
206             do
207             {
208                 do
209                 {
210                     k = RandomBelow(q - 1) + 1;
211                     ScalarProd(k, xp, yp, ref xc, ref yc);
212                     r = ModP(xc, q);
213                 }
214                 while (r == 0);
215                 s = ModP(r * d + k * e, q);
216             }
217             while (s == 0);
218             return BinVectorConc(r, s);
219         }
220         //Второй вариант проверки подписи
221         public bool verifySign(byte[] hash, GR3410_2012_Sign sign,
                GR3410_2012_VerifyKey verifyKey)
222         {
```

```
223             return verifySign(hash, sign.Sign, verifyKey.Xq,
                    verifyKey.Yq);
224         }
225         //Первый вариант проверки подписи
226         public bool verifySign(byte[] hash, byte[] sign, byte[] xqb,
                byte[] yqb)
227         {
228             BigInteger r, s;
229             GetRSFromVector(sign, out r, out s);
230 
231             if (r >= q || s >= q)
232                 return false;
233 
234             BigInteger xq = GetPositive(xqb);
235             BigInteger yq = GetPositive(yqb);
236 
237             BigInteger h = GetPositive(hash);
238             BigInteger e = ModP(h, q);
239             if (e == 0)
240                 e = 1;
241             BigInteger v = Inverse(e, q);
242 
243             BigInteger z1 = ModP(s * v, q);
244             BigInteger z2 = ModP(-r * v, q);
245 
246             BigInteger xc = -1, yc = -1, xt = -1, yt = -1;
247             ScalarProd(z1, xp, yp, ref xc, ref yc);
248             ScalarProd(z2, xq, yq, ref xt, ref yt);
249             EllipticSum(xc, yc, xt, yt, ref xc, ref yc);
250             BigInteger R = ModP(xc, q);
251 
252             return R == r;
253         }
254     }
255 }
```

Приложение Г Пример работы с цифровой подписью

Для работы с цифровой подписью в рамках данной работы реализовано 3 приложения. Первое предназначено для задания просмотра и редактирование параметров. Изображение его главного окна представлено на Рисунке Г.1.

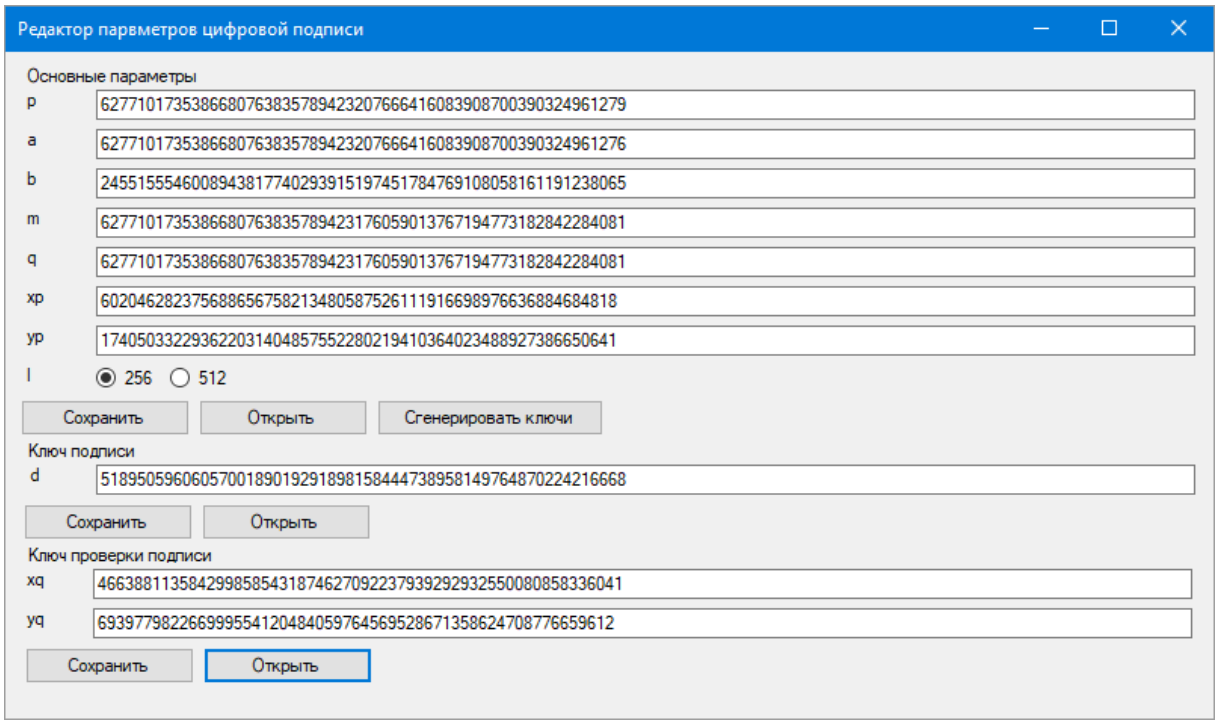

Рисунок Г.1 — Окно программы для редактирования параметров

В окне программы на Рисунке Г.1 уже введены данные в десятеричном формате. В дальнейшем будем использовать эти значения. Интерфейс позволяет редактировать и сохранять объекты «GR3410_2012_Parameters», «GR3410_2012_SignKey» и «GR3410_2012_VerifyKey» в виде XML файлов. Пример кода XML файла показан на Рисунке .

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <GR3410_2012_Parameters
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <P>////////////+////////////////////AA==</P>
      <A>/P////////+////////////////////AA==</A>
      <B>sblGwezeuP5JMCRyq+mnD+eAnOUZBSFk</B>
      <M>MSjStLHJaxQ2+N6Z/////////////////AA==</M>
      <Q>MSjStLHJaxQ2+N6Z/////////////////AA==</Q>
      <Xp>EhD/gv0K//QAiKFD6yC/fPaQMLAOqI0Y</Xp>
      <Yp>EUh5HqF3+XPVzSRr7REQY3jayP+VKxkH</Yp>
      <L>256</L>
  </GR3410_2012_Parameters>
```

Рисунок Г.2 — Пример кода XML файла с параметрами

Далее рассмотрим работу приложения для генерации подписи. Изображение главного окна представлено на Рисунке Г.3.

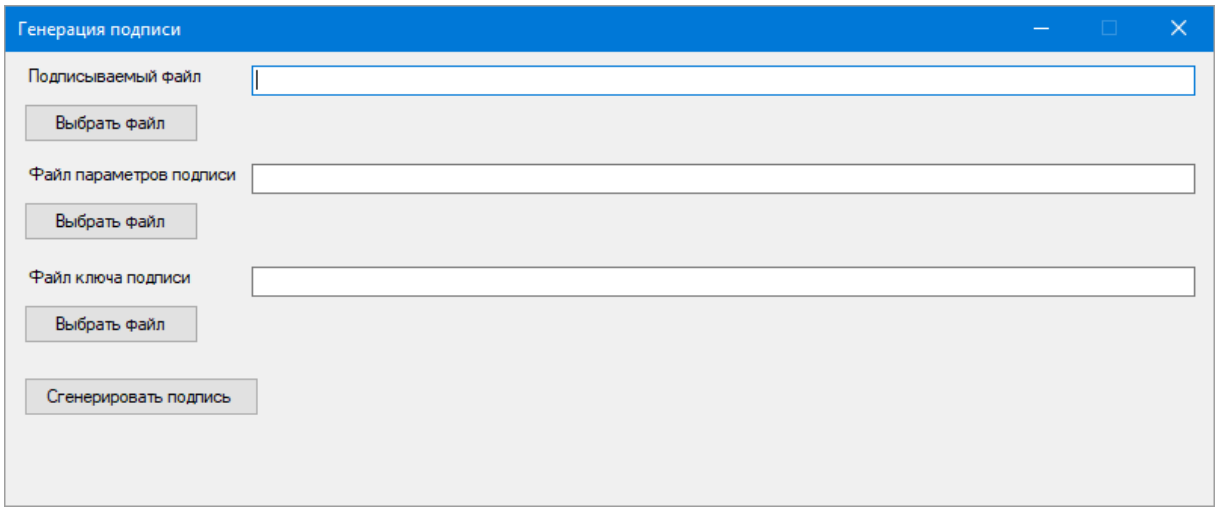

Рисунок Г.3 — Окно программы для генерации подписи

В окне, изображенном на Рисунке Г.3, можно задать подписываемый файл и файлы с параметрами схемы цифровой подписи и ключом подписи. После указания необходимых параметров, для генерации подписи необходимо нажать кнопку «Сгенерировать подпись». Далее начнется расчет хэша. По его завершении, можно сохранить файл, содержащий подпись.

Для примера возьмем файл содержащий следующий набор байт $F =$ `D1E520E2E5F2F0E82C20D1F2F0E8E1EEE6E820E2EDF3F6E82C20E2E5FEF2FA20F120EC EEF0FF20F1F2F0E5EBE0ECE820EDE020F5F0E0E1F0FBFF20EFEBFAEAFB20C8E3EEF0E5E2FB` и определенные ранее параметры схемы цифровой подписи и ключ шифрования. Сгенерируем подпись, в результате чего, она будет сохранена в файл (Рисунок Г.4). Назовем его «sign.xml».

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <GR3410_2012_Sign xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <Sign>ke4bNqf0xCMU7luEc4sp95OnEJdE3uV4AAAAAAAAAAAaB
          XIhfJj7p75+V83JcDhKWFaqh0kThqYAAAAAAAAAAA==</Sign>
  </GR3410_2012_Sign>
```

Рисунок Г.4 — Код файла «sign.xml»

Для проверки подписи также используется специальное приложение. Изображение главного окна приложения для проварки подписи показано на Рисунке Г.5.

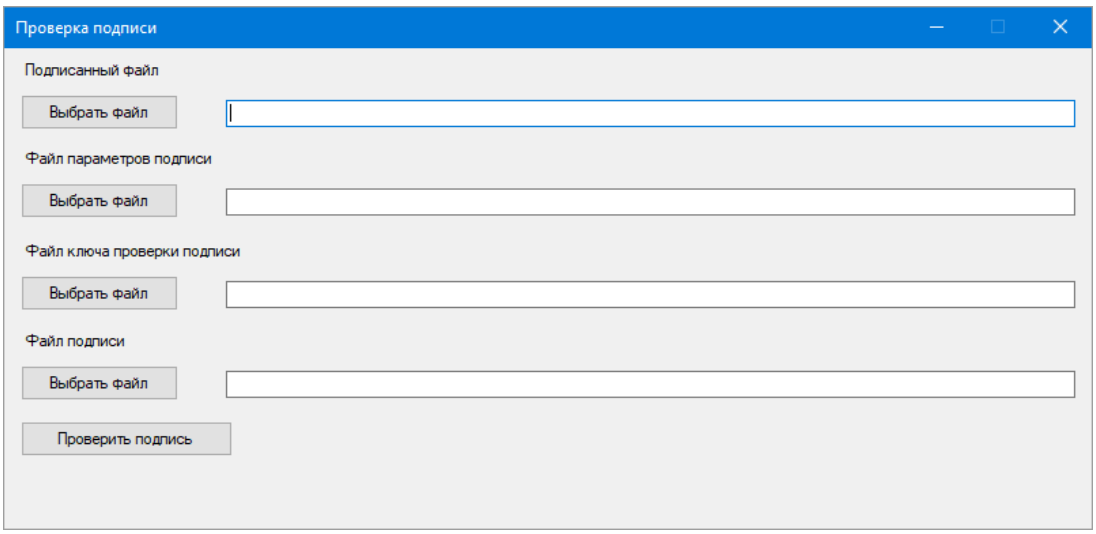

Рисунок Г.5 — Окно программы для проверки подписи

возьмем тот же файл, те же параметры схемы цифровой подписи и определенный ранее ключ проверки подписи. В качестве подписи зададим файл «sign.xml» и запустим процесс проверки подписи. В результате подпись пройдет проверку, о чем мы получим всплывающее сообщение (Рисунок Г.6).

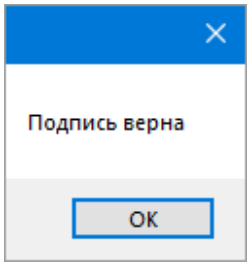

Рисунок Г.6 — Сообщение о том, что подпись верна

Попробуем изменить подписанный файл. Запишем вместо него следующее $F =$ `01D1E520E2E5F2F0E82C20D1F2F0E8E1EEE6E820E2EDF3F6E82C20E2E5FEF2FA20F120EC EEF0FF20F1F2F0E5EBE0ECE820EDE020F5F0E0E1F0FBFF20EFEBFAEAFB20C8E3EEF0E5E2FB.` При проверке сгенерированной ранее подписи, программа выдаст сообщение о том, что подпись неверна (Рисунок Г.7).

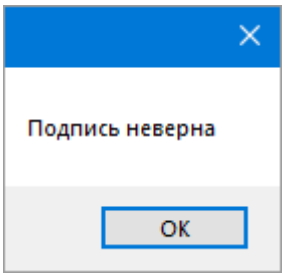

Рисунок Г.7 — Сообщение о том, что подпись неверна