МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики
Базовая кафедра №252 – информационной безопасности

РАБОТА ДОПУЩЕНА К ЗАЩИТЕ
Заведующий кафедрой _____ А.В. Корольков
«___» _____ 2021 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

по специальности 10.05.01 Компьютерная безопасность

на тему: **«Алгоритм построения 2^k-мультиколлизий для хэш- функции «Мора»**

Обучающийся		Чижов Никита Дмитриевич
Шифр		15К0207
Группа		ККСО-03-15
Руководитель работы	Старший преподаватель базовой кафедры №252	Кирюхин В.А.

Москва 2021 г.

Содержание

ВВЕДЕНИЕ

Актуальность. Криптографические методы не только позволяют обеспечивать конфиденциальность, но и контролировать целостность передаваемых данных. Контроль целостности, как правило, осуществляется методом вычисления контрольной суммы данных. В настоящее время разработано большое число алгоритмов, вычисляющих контрольные суммы передаваемых данных. Дело в том, что в большинстве случаев достаточно простой контрольной суммы, например, в тех случаях, когда объем информации не известен заранее или имеет значение скорость обработки. Но при использовании простой контрольной суммы можно подобрать несколько массивов данных, у которых будет получаться одинаковая контрольная сумма.

Криптографически стойкие контрольные суммы получаются в случае применения к исходному тексту хэш-функции. Возьмем ситуацию: необходимо отправить по сети документ и нужна абсолютная уверенность в том, что полученный файл совершенно идентичен оригиналу и содержащиеся в нем данные не были изменены «в пути», т.е. целостность документа была не нарушена. Для этого существуют алгоритмы идентификации цифровой информации. Такие алгоритмы работают только с цифровой информацией. При необходимости идентификации текстового документа или изображения, нужно с помощью кодирования вначале получить цифровой аналог данной информации. Различают несколько способов идентификации: контрольные суммы, контроль CRC, хэширование и цифровая подпись – таковы базовые средства аутентификации при цифровой передаче данных.

Контрольные суммы – наиболее простой способ проверки целостности данных, передаваемых в цифровом виде, при котором вычисляется контрольная сумма сообщения (определенное значение, которое и идентифицирует цифровую информацию).

Наиболее совершенный способ идентификации цифровой информации – алгоритмы цифровой подписи и хэширование.

Функция хэширования H (m) или хэш-функция (hash-function) –

это детерминированная функция, на вход которой подается строка битов произвольной длины, а выходом всегда является битовая строка фиксированной длины n.

Значение хэш-функции H (m) для входа m называют хэшем. Исходная строка m, для которой вычислено хэш-значение, называется прообразом хэш-функции.

Свойства, которые должны быть присущи криптографическим хэш-функциям:

1. **Сопротивление поиску первого прообраза**: при наличии хэша h должно быть трудно найти какое-либо сообщение m, такое что h = hash(m), это свойство связано с понятием односторонней функции. Функции, у которых отсутствует это свойство, уязвимы для атак нахождения первого прообраза.
2. **Сопротивление поиску второго прообраза**: при наличии сообщения m1, должно быть трудно найти другое сообщение m2 (не равное m1) такое, что hash(m1) = hash(m2). Это свойство иногда называют слабым сопротивлением поиску коллизий. Функции, у которых отсутствует это свойство, уязвимы для атак поиска второго прообраза.
3. **Стойкость к коллизиям** Коллизией для хэш-функции называется такая пара значений m1 и m2, $m1 \neq m2$, для которой hash(m1) = hash(m2). Так как количество возможных открытых текстов больше числа возможных значений свёртки, то для некоторой свёртки найдётся много прообразов, а следовательно, коллизии для хэш-функций обязательно существуют. Например, пусть длина хэш-прообраза 6 битов, длина свёртки 4 бита. Тогда число различных свёрток — 2^4, а число хэш-прообразов — 2^6, то есть в 4 раза больше, значит хотя бы одна свёртка из всех соответствует 4 прообразам.
4. **Псевдослучайность**: должно быть трудно отличить генератор псевдослучайных чисел на основе хэш-функции от генератора случайных чисел.

На практике, построение криптографической функции с входом переменного размера задача не из простых. По этой причине большинство хеш-функций основаны на итерированной конструкции, в которой используется так называемая функция сжатия, чей вход имеет фиксированный размер. В данной работе мы рассматриваем именно односторонние хеш-функции построенные путем повторения функции сжатия. Поэтому для построения коллизии хэш функции, можно воспользоваться алгоритмом поиска одиночной коллизии для функции сжатия, а затем найти коллизии непосредственно самой хэш функции.

K – коллизией называется k открытых текстов, дающих одинаковое значение хэш функции. В идеальной хэш функции с n выходными битами для поиска k-коллизии, с учетом «парадокса дней рождений», потребуется времени $2^{n(k-1)/k}$, что при больших k ведет к 2^n. В данной работе будет показано, что время на поиск мультиколлизий для хэш функции «Мора» можно свести к $t \cdot 2^{\frac{n}{2}+1}$.

Таким образом, основная цель нашей выпускной квалификационной работы - разработка алгоритмов и реализация программного обеспечения для построения мультиколлизий хэш-функции «Мора».

Объектом исследования являются построения мультиколлизий для хэш функции.

Предметом исследования являются проблемы и перспективы использования построения атак на хэш-функции с контрольными суммами.

Для достижения указанных целей перед исследованием ставятся следующие основные задачи:

1. Провести обзор существующих алгоритмов построения коллизий и мультиколлизий для функций сжатия и хэш-функций на основе схемы Меркла-Дамгарда.

2. Провести обзор существующих алгоритмов и структур данных для построения атак на хэш-функции, использующие контрольные суммы.

3. Разработать алгоритмы построения мультиколлизий для хэш-функции «Мора». Оценить их сложности по времени и памяти.

4. Программная реализация разработанных алгоритмов.

5. Провести экспериментальные исследования с использованием разработанного программного обеспечения.

6. Обобщить результаты исследования и сформулировать выводы по всей работе.

Для решения поставленных в дипломной работе задач использованы следующие методы научного исследования: теоретические: анализ и систематизация теоретических и практических данных с целью определения актуальности и состояния разработки указанной проблемы; эмпирические: исследование, контрольный эксперимент, формирующий эксперимент.

Теоретическая значимость исследования заключается в том, что результаты работы могут послужить стимулом к новым практическим экспериментам в области исследований построения мультиколлизий. Практическая значимость заключается в исследовании алгоритма построения 2^k-мультиколлизий для хэш функции «Мора», которые можно успешно внедрять в способы идентификации цифровой информации.

Методологической и теоретической основой исследования послужили работы зарубежных и российских авторов в области алгоритмов построения коллизий и мультиколлизий для функции сжатия и хэш-функций.

Первая и вторая глава - теоретический анализ данной проблемы. Третья и четвертая глава – практическая часть построения мультиколлизий для хэш-функции «Мора».

ГЛАВА 1. ОБЩИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В данной главе приводятся обозначения, необходимые в дальнейшем исследовании.

1.1 Обозначения и общие сведения

V^* - множество всех двоичных векторов двоичной размерности (далее-векторы), включая пустую строку;

Z_2^n – кольцо вычетов по модулю 2^n;

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

$\oplus$ - операция покомпонентного сложения по модулю 2 двух двоичных векторов одинаковой размерности;

$\boxplus$ - операция сложения в кольце Z_2^n;

IV - инициализационный вектор функции хэширования

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

$Vec_n: Z_2^n \to V_n$ – биективное отображение, сопоставляющее элементу кольца Z_2^n, его двоичное представление, т. е. для любого элемента z кольца 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 $Vec_n(z) = z_{n-1} \| \ldots \| z_1 \| z_0$

$Int_n: V_n \to Z_2^n$ – отображение, обратное Vec_n;

$\coloneqq$ - операция присвоения

1.2 Конечные поля

В данном параграфе, пользуясь [1], введем понятие конечного поля и его размера. Перечислим основные свойства операций, заданных в конечном поле. Конечным полем GF(q) называется конечное множество, на котором определены аддитивная + и мультипликативная $*$ операции, удовлетворяющие следующим свойствам:

1) Коммутативность сложения: $\forall\, a, b \in \mathrm{GF(q)}:\ a + b = b + a$

2) Ассоциативность сложения: $\forall\, a, b, c \in \mathrm{GF(q)}:\ (a + b) + c = a + (b + c)$

3) Существование нулевого элемента: $\exists\, 0 \in \mathrm{GF(q)}, \forall\, a \in \mathrm{GF(q)}: a + 0 = 0 + a = a$

4) Существование противоположного элемента: $\forall\, a \in \mathrm{GF(q)}, \exists(-a) \in \mathrm{GF(q)}: a + (-a) = 0$

5) Коммутативность умножения: $\forall\, a, b \in \mathrm{GF(q)}:\ a * b = b * a$

6) Ассоциативность умножения: $\forall\, a, b, c \in \mathrm{GF(q)}:\ (a * b) * c = a * (b * c)$

7) Существование единичного элемента: $\exists\, e \in \mathrm{GF(q)} \setminus \{0\}, \forall\, a \in \mathrm{GF(q)}: a * e = a$

8) Существование обратного элемента для ненулевых элементов: $\forall\, a \in \mathrm{GF(q)}: a \neq 0\ , \exists a^{-1} \in \mathrm{GF(q)}: a * a^{-1} = e$

9) Дистрибутивность умножения относительно сложения: $\forall\, a, b, c \in \mathrm{GF(q)}:\ (a + b) * c = (a * c) + (b * c)$

Конечное поле будем обозначать GF(q), где q = p^n p – простое, n – натуральное.

1.3 Итеративная хэш функция

Все криптографические хэш-функции должны создавать дайджест фиксированного размера из сообщения переменного размера. Применять такую функцию лучше всего, используя итерацию. Вместо хэш-функции с вводом переменного размера создана и используется необходимое количество раз функция с вводом фиксированного размера, называемая функцией сжатия. Она сжимает n-битовую строку и создает m-битовую строку, где n обычно больше, чем m. Эта схема известна как итеративная криптографическая функция.

1.4 Струтура Меркла-Дамгарда

Структура Меркла-Дамгарда[2] - метод построения криптографических хеш-функций, предусматривающий разбиение входных сообщений произвольной длины на блоки фиксированной длины и работающий с ними по очереди с помощью функции сжатия, каждый раз принимая входной блок с выходным от предыдущего прохода.

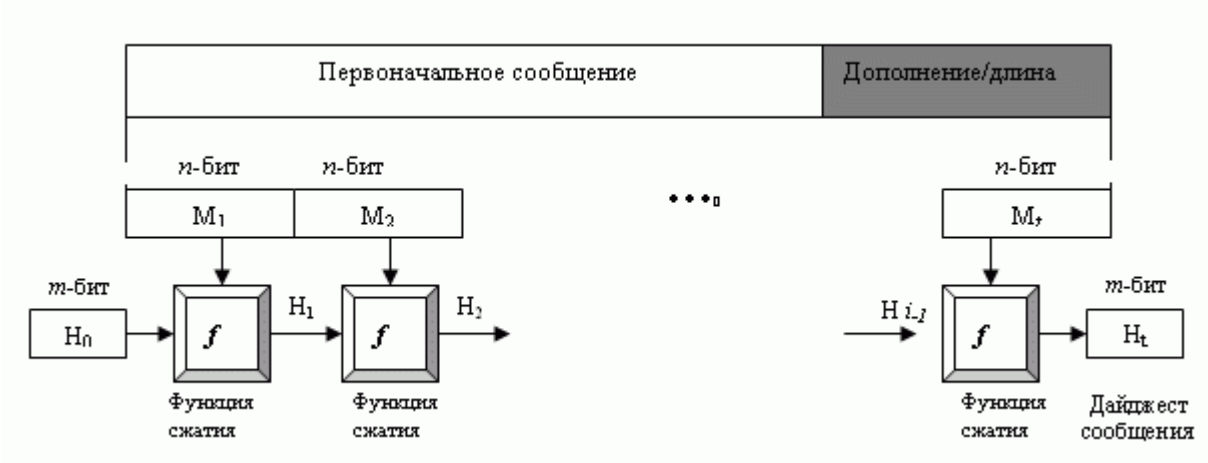

Рисунок 1 - Схема Меркла-Дамгарда

Схема использует следующие шаги:

1 Длина сообщения и дополнение добавляются в конец сообщения, чтобы создать увеличенное сообщение, которое может быть равномерно разделено на n -битовые блоки; здесь n - размер блока, который будет обработан функцией сжатия.

2 Сообщение тогда рассматривают как t блоков, размер каждого состоит из n бит. Мы обозначим каждый блок M_1,..., M_t. Мы обозначаем дайджест, созданный при t итерациях, - H_1, H_2,...., H_t

3 Перед стартом итерации дайджест H_0 устанавливается на фиксированное значение, обычно называемое IV (начальное значение или начальный вектор).

4 Функция сжатия при каждой итерации обрабатывает H_{i-1} и M, создавая новый H_i. Другими словами, мы имеем $H_i = f(H_{i-1}, M_i)$, где f - функция сжатия.

5 H_t - функция криптографического хэширования первоначального сообщения, то есть h(M).

Если функция сжатия в схеме Меркла-Дамгарда устойчива к коллизии, хэш-функция также устойчива к коллизии.

1.5 Хэш функция «Мора

Описание хэш функции «Мора» было взято из работы [3]

Параметры хэш функции, следующие:

- Длина блока обрабатываемых данных n = 64 бит
- Количество раундов r = 9 (выбор количества раундов опциональная опция, в практической части работы выбрано 8 раундов)
- Итерационные константы C задаются таблицей 1.3.1

Значение счётчика	Итерационная константа
0000000000000001	c0164633575a9699
0000000000000002	925b4ef49a5e7174
0000000000000003	86a89cdcf673be26
0000000000000004	1885558f0eaca3f1
0000000000000005	dcfc5b89e35e8439
0000000000000006	54b9edc789464d23
0000000000000007	f80d49afde044bf9
0000000000000008	8cbbdf71ccaa43f1
0000000000000009	cb43af722cb520b9

Таблица 1 Итерационные константы

Матрица линейного преобразования (L) имеет вид:

(0x3a22, 0x8511, 0x4b99, 0x2cdd,
0x483b, 0x248c, 0x1246, 0x9123,
0x59e5, 0xbd7b, 0xcfac, 0x6e56,
0xac52, 0x56b1, 0xb3c9, 0xc86d);

Перестановка полубайт (P) задаётся массивом:

τ = (15, 9, 1, 7, 13, 12, 2, 8, 6, 5, 14, 3, 0, 11, 4, 10)

Нелинейное биективное преобразование задается подстановкой:

$\pi = Vec_4\pi' Int_4 : V_4 \to V_4$,

где $\pi = (15, 9, 1, 7, 13, 12, 2, 8, 6, 5, 14, 3, 0, 11, 4, 10)$

Преобразования, используемые при вычислении хэш-кода:

1. $X[k] : V_{64} \to V_{64}, X[k] = k \oplus a, k, a \in V_{64}$

2. $S : V_{64} \to V_{64}, S(a) = S(a_{15} || \dots || a_0) = \pi(a_{15}) || \dots || \pi(a_0)$, где $a = a_{15} || \dots || a_0 \in V_{64}, a_j \in V_4, j = 0, \dots, 15$

3. $P : V_{64} \to V_{64}, P(a) = P(a_{15} || \dots || a_0) = a_{\tau(15)} || \dots || a_{\tau(0)}$, где $a = a_{15} || \dots || a_0 \in V_{64}, a_j \in V_4, j = 0, \dots, 15$

4. $L : V_{64} \to V_{64}, L(a) = L(a_3 || \dots || a_0) = l(a_3) || \dots || l(a_0)$, где $a = a_3 || \dots || a_0 \in V_{64}, a_j \in V_4, j = 0, \dots, 3$

Функция сжатия функции хэширования задается следующим образом:

$$g_N : V_{64} \, x \, V_{64} \to V_{64}, N \in V_{64}$$

$$g_N(h, m) = E(LPS(h \oplus N), m) \oplus h \oplus m, \text{где}$$

$$E(K, m) = X[K_9]LPSX[K_8] \dots LPSPX[K_1](m)$$

Значения $K_i \in V_{64}, i = 2, \dots, 9$ вычисляются следующим образом:

$K_1 = K$;

$K_i = LPS\ (K_{i-1} \oplus C_{i-1})$, $i = 2, \dots, 9$.

Алгоритм вычисления функции хэширования.

Этап 0. На вход функции хэширования подается начальный вектор инициализации и данные, подлежащие хешированию.

Этап 1:

Присвоить начальные значения текущих величин:

1.1 $h := IV$;

1.2 $N := 0^{64} \in V_{64}$;

1.3 $\Sigma := 0^{64} \in V_{64}$;

1.4 Перейти к этапу 2.

Этап 2:

2.1 Проверить условие $|M| < 64$.

При положительном исходе перейти к этапу 3.

В противном случае выполнить действия 2.2 – 2.6.

2.2 Вычислить подвектор $m \in V_{64}$ сообщения $M : M = M'||m$. Далее выполнить последовательность вычислений:

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

2.4 $N := Vec_{64}(Int_{64}(N) \boxplus 64)$

2.5 $\Sigma := Vec_{64}\left(Int_{64}(\Sigma) \boxplus Int_{64}(\mathrm{m})\right)$

2.6 Перейти к шагу 2.1

Этап 3:

3.1 $m := 0^{63-|M|}||1||M$

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

3.3 $N := Vec_{64}(Int_{64}(N) \boxplus |\mathrm{M}|)$

3.4 $\Sigma := Vec_{64}\left(Int_{64}(\Sigma) \boxplus Int_{64}(\mathrm{m})\right)$

3.5 $h := g_0(h, N)$

3.6 $h := g_0(h, \Sigma)$

3.7 Конец работы алгоритма. Значение величины h, полученное на шаге 3.5, является значением функции хэширования H(M).

1.6 Парадокс дней рождений

Так называемый "парадокс дней рождений" [4] состоит в следующем.

Первая задача. Каким должно быть число k, чтобы для данного значения X и значений Y_1, …, Y_k, каждое из которых принимает значения от 1 до n, вероятность того, что хотя бы для одного Y_i выполнялось равенство X=Y

$P(X = Y) \geq 0.5$

Для одного значения Y вероятность того, что X=Y, равна 1/n.

$P(X = Y) = 1/n$

Соответственно, вероятность того, что $X \neq Y$, равна 1 – 1/n.

$P(X \neq Y) = 1 - 1/n$

Если создать k значений, то вероятность того, что ни для одного из них не будет совпадений, равна произведению вероятностей, соответствующих одному значению, т.е. $(1 – 1/n)^{k.}$

Следовательно, вероятность по крайней мере одного совпадения равна

$P (X = Y_i) = 1 – (1 – 1/n)^k$

По формуле бинома Ньютона

$$(1-a)k = 1 - ka + \frac{k(k-1)}{2!}a^2 - \cdots \approx 1 - ka$$

$$1 - \left(1 - \frac{k}{n}\right) = \frac{k}{n} = 0.5$$

$$k = n/2$$

Таким образом, для хэш-кода длиной m бит достаточно выбрать 2m-1 сообщений, чтобы вероятность совпадения хэш-кодов была больше 0,5.

Теперь рассмотрим вторую задачу. Обозначим P(n,k) вероятность того, что в множестве из k элементов, каждый из которых может принимать n значений, есть хотя бы два с одинаковыми значениями. Чему должно быть равно k, чтобы P(n,k) была бы больше 0,5?

Число различных способов выбора элементов таким образом, чтобы при этом не было дублей, равно

$$n(n-1) \ldots \frac{(n-k+1)n!}{(n-k)!}$$

Всего возможных способов выбора элементов равно

$$n^k$$

Вероятность того, что дублей нет, равна

$$\frac{n!}{(n-k)!\,n^k}$$

Вероятность того, что есть дубли, соответственно равна

$$1 - \frac{n!}{(n-k)!\,n^k}$$

$$P(n,k) = 1 - \frac{n!}{(n-k)! \times n^k} = 1 - \frac{n \times (n-1) \times \ldots (n-k-1)}{n^k}$$

$$= 1 - \left[\frac{n-1}{n} \times (n-2) \times \ldots \times \frac{n-k+1}{n}\right]$$

$$= 1 - [\left(1 - \frac{1}{n}\right) \times \left(1 - \frac{2}{n}\right) \times \ldots \times \left(1 - \frac{k-1}{n}\right)]$$

Известно, что если хэш-код имеет длину m бит, т.е. принимает 2^m значений, то

$$1 - x \leq e^{-x}$$

$$P(n,k) > 1 - [e^{-\frac{1}{n}} \times e^{-\frac{2}{n}} \times \ldots \times e^{-\frac{k}{n}}]$$

$$P(n,k) > 1 - e^{-k(k-1)/n}$$

$$\frac{1}{2} = 1 - e^{-k(k-1)/n}$$

$$2 = e^{k(k-1)/n}$$

$$ln2 = k(k-1)/2n$$

$$k(k-1) \approx k^2$$

$$k = \sqrt{2n\ x\ ln2} = 1{,}17\sqrt{n} \approx \sqrt{n} = \sqrt{2^m} = 2^{m/2}$$

Подобный результат называется "парадоксом дней рождений", потому что в соответствии с приведенными выше рассуждениями для того, чтобы вероятность совпадения дней рождения у двух человек была больше 0,5, в группе должно быть всего 23 человека. Этот результат кажется удивительным, возможно, потому что для каждого отдельного человека вероятность того, что с его днем рождения совпадет день рождения у кого-то другого в группе, достаточно мала.

ГЛАВА 2. АЛГОРИТМЫ ПОСТРОЕНИЯ КОЛЛИЗИЙ И МУЛЬТКОЛЛИЗИЙ ДЛЯ ФУНКЦИЙ СЖАТИЯ И ХЭШ-ФУНКЦИЙ

В данной главе будут рассмотрены способы построения коллизий и мультиколлизий.

2.1 Методы построения одиночной коллизии

2.1.1 Метод «грубой силы»

Метод грубой силы подразумевает полный перебор всех возможных значений хэш функции. Например, пусть функция сжатия генерирует 128 бит, тогда для поиска коллизии нам потребуется перебрать 2^{128} вариантов, к тому же нам потребуется еще больше памяти, чтобы хранить все эти значения. Хорошо, мы можем улучшить алгоритм, воспользовавшись парадоксом «дней рождений», т.е. взяв 2^{64} значений, мы с вероятностью более 0.5 получим совпадение, тем не менее, даже распараллелив вычисления, нам все равно придется хранить более 2^{64} байт данных. Далее будут рассмотрены более оптимальные методы поиска одиночных коллизий.

2.1.2 Атака дней рождений

Предположим, что используется 64-битный хэш-код. В среднем противник должен перебрать 2^{63} сообщений для того, чтобы найти другое сообщение, у которого хэш-код равен перехваченному сообщению. Тем не менее, возможны различного рода атаки, основанные на "парадоксе дня рождения". Возможна следующая стратегия:

1. Противник создает 2m/2 вариантов сообщения, каждое из которых имеет некоторый определенный смысл. Противник подготавливает такое же количество сообщений, каждое из которых является поддельным и предназначено для замены настоящего сообщения.
2. Два набора сообщений сравниваются в поисках пары сообщений, имеющих одинаковый хэш-код. Вероятность успеха в соответствии с "парадоксом дня рождения" больше, чем 0,5. Если соответствующая

пара не найдена, то создаются дополнительные исходные и поддельные сообщения до тех пор, пока не будет найдена пара.

Таким образом, если используется 64-битный хэш-код, то для подбора двух сообщений с одинаковым хэш-кодом в среднем необходимо перебрать 2^{32} сообщений.

2.1.3 Алгоритм Флойда

Алгоритм Флойда предназначен для поиска цикла в последовательности значений итеративной функции. Алгоритм также носит название «Заяц и черепаха».

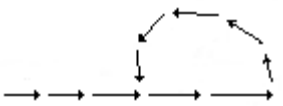

Рисунок 2 Алгоритм Флойда

Надо найти начало петли. Для решения поставленной задачи, пустим 2 итератора по списку, один будет двигаться в два раза быстрее (будет делать на каждый такт по два шага). Когда заяц (быстрый итератор) догонит черепаху (медленный итератор), мы знаем, что:

- Черепаха либо дошла до петли, либо уже по ней движется. Пройти петлю целиком она не могла
- Заяц пробежал по петле как минимум один раз
- Заяц пробежал ровно в два раза больше

Теперь ставим черепаху в начало списка и пускаем два итератора с одинаковой скоростью, пока они не пересекутся.

Причем же здесь хэш функция? Пусть наш список будет выглядеть так:

X_0 – произвольное число

$X_{i+1} = hash(x_i)$

Для медленного итератора на каждой итерации вычисляем хэш значение от предыдущего значения один раз, для быстрого вычисляем hash(hash(x)). Найденная петля и будет наша коллизия. Плюс данного алгоритма в том, что он требует O (1) затрат памяти.

```
def floyd(f, x0):
    # Основная часть алгоритма: находим повторение x_i = x_2i.
    # Заяц движется вдвое быстрее черепахи,
    # и расстояние между ними увеличивается на единицу от шага к шагу.
    # Однажды они окажутся внутри цикла, и тогда расстояние между ними
    # будет делиться на λ.
    tortoise = f(x0) # f(x0) является элементом, следующим за x0.
    hare = f(f(x0))
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(f(hare))

    # В этот момент позиция черепахи v,
    # которая равна расстоянию между черепахой и зайцем,
    # делится на период λ. Таким образом, заяц, двигаясь
    # по кольцу на одну позицию за один раз,
    # и черепаха, опять начавшая движение со стартовой точки x0 и
    # приближающаяся к кольцу, встретятся в начале кольца
    # Находим позицию μ встречи.
    mu = 0
    tortoise = x0
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(hare)   # Заяц и черепаха двигаются с одинаковой скоростью
        mu += 1

    # Находим длину кратчайшего цикла, начинающегося с позиции x_μ
    # Заяц движется на одну позицию вперёд,
    # в то время как черепаха стоит на месте.
    lam = 1
    hare = f(tortoise)
    while tortoise != hare:
        hare = f(hare)
        lam += 1

    return lam, mu
```

Листинг 1 Алгоритм «Флойда»

2.1.4 Алгоритм Брента

Ричард Брент описал альтернативный алгоритм нахождения цикла, которому, подобно алгоритму черепахи и зайца, требуется лишь два указателя

на последовательность. Однако он основан на другом принципе — поиске наименьшей степени 2i числа 2. По утверждению Брента, алгоритм работает на 36 % быстрее алгоритма Флойда.

```
def brent(f, x0):
    # Основная фаза: ищем степень двойки
    power = lam = 1
    tortoise = x0
    hare = f(x0)  # f(x0) — элемент/узел, следующий за x0.
    while tortoise != hare:
        if power == lam:  # время начать новую степень двойки?
            tortoise = hare
            power *= 2
            lam = 0
        hare = f(hare)
        lam += 1

    # Находим позицию первого повторения длины λ
    mu = 0
    tortoise = hare = x0
    for i in range(lam):
    # range(lam) образует список со значениями 0, 1, ... , lam-1
        hare = f(hare)
    # расстояние между черепахой и зайцем теперь равно λ.

    # Теперь черепаха и заяц движутся с одинаковой скоростью, пока не встретятся
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(hare)
        mu += 1

    return lam, mu
```

Листинг 2 Алгоритм «Брента»

2.1.5 Лямбда-алгоритм Полларда

Если у нас есть функция с коллизиями, и мы начнём её итерировать (т.е. как в прошлый раз: $X_{i+1} = h(X_i)$) от двух разных случайных значений, скорее всего, цепочки пересекутся, после чего сольются.

На этой идее основан следующий алгоритм:

- Берём случайное x_0, итерируем от него функцию Q раз, получаем x_q.
- Берём случайное y_0, и итерируем от него до тех пор, пока не получим то же самое x_q.

Теперь, зная количество шагов, которые пришлось пройти по первому и по второму пути, можно пустить по ним двух черепах, дав одной из них такую фору, чтобы они встретились.

2.1.6 Параллельный лямбда-алгоритм Полларда

В работе [5] был представлен способ ускорения поиска коллизий, с помощью распараллеливания вычислений по следующему принципу:

- Пусть у нас есть T компьютеров.
- Среди множества значений функции выберем «волшебные». Выберем так, чтобы до попадания в такое значение нужно было бы сделать в среднем $2^{n/2}/T$ попыток. Например, посчитаем X = $\log_2 2^{n/2}/T = \frac{n}{2} - \log_2 T$ и скажем, что первые X бит должны быть нулями.
- Сгенерируем T случайных чисел, раздадим их компьютерам.
- Каждый компьютер начинает итерировать функцию со своего значения и продолжает до тех пор, пока не получат «волшебное» значение. Получив, присылает в центр результат, своё начальное значение и количество шагов.
- Центр сравнивает результаты, у кого-нибудь да совпадёт. Из двух начальных значений и двух длин путей несложно найти коллизию.

Конечно, может ни у кого не совпасть. Но, по парадоксу «дней рождений», совпадёт с вероятностью больше 0,5, нам этого достаточно**.**

2.2 Методы построения мультиколлизий

Как правило, трудоемкость алгоритмов построения мультиколлизий для функций хэширования является довольно высокой и не допускает их практическую реализацию. В то же время они имеют несомненную ценность с теоретической точки зрения, так как характеризуют конструкцию Меркля – Дамгорда как не идеальную и изменяют наш взгляд на теоретическую стойкость хэш-функций, используемых на практике. Принципы, лежащие в основе

методов построения мультиколлизий, часто используются при криптографическом анализе конкретных хэш-функций.

Мультиколлизия (r-коллизия) произвольной хэш функции H : $V^* \rightarrow V_n$ может быть построена с использованием парадокса дней рождения с трудоемкостью порядка $(r!)^{1/r} \cdot 2^{n(r-1)/r}$ операций вычисления значения функции H [6, 7].

Позднее появились более эффективные методы построения мультиколлизий для итеративных хэш функций.

Впервые более эффективная атака с использованием мультиколлизий предложена в работе [8] Жуксом. Данная атака была предложена для хэш-функций, использующих каскадирование, и основанных на усиленной конструкции Меркла-Дамгарда. Согласно этой конструкции, информационное сообщение M дополняется до длины, кратной длине блока данных, разбивается на l частей и дополняется блоком m_{l+1}, который содержит длину оригинального сообщения M, а хэширование происходит согласно формуле:

$$h_i := f(h_{i-1}, m)$$

где h_i – промежуточное хэш-значение, полученное на i -ом шаге;

$f(h, m)$– функция сжатия, обеспечивающая фиксированную длину результата.

Жукс в работе [8] предложил метод конструирования 2^t-коллизий для итеративной структуры с трудоемкостью всего $O(t\ 2^{n/2})$. Как показано на рисунке 3, атакующий сначала генерирует t различных попарных коллизий {(B1, B1*), (B2, B2*), …, (Bt, Bt*)}. Затем атакующий может сразу получить 2^t-коллизию вида (b1, b2,…, bt), где bi – один из двух блоков Bi и Bi*. Жукс предложил следующий алгоритм:

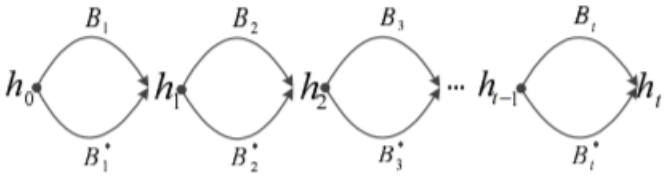

Рисунок 3 Алгоритм Жукса

1. Пусть h_0 будет равно вектору инициализации
2. Для i от 1 до t выполняем
    - Ищем такие B_i и B_i^*, для которых $g_N(h_{i-1}, B_i) = g_N(h_{i-1}, B_i^*)$
    - Составление результирующих 2^t сообщений в форме $(b_1,\ldots, b_t)$, где b_i – это один из двух блоков: B_i или B_i^*.

2.3 Методы построения мультиколлизий для хэш функций с контрольными суммами

Некоторое время в качестве одного из метода противодействия мультиколлизиям рассматривалось дополнение исходного сообщения различного рода контрольными суммами. Например, в рассматриваемой хэш функции все блоки складываются друг с другом по модулю 2^{64}. В связи с этим, помимо поиска коллизий функции сжатия, также нужно найти коллизию в Σ .

Однако в работах [9, 10] были предложены методы построения мультиколлизий для таких хэш функций.

Метод, предложенный в работе [9] для 2^b коллизий, имеет трудоемкость порядка $b \times \left(\left(\frac{b}{2}\right) + 1\right) \times 2^{\frac{t}{2}}$ операций вычисления функции сжатия (рисунок 4).

Variables:

1. i, j, k : integers.
2. $chunk[i]$: a pair of $(b/2)+1$-block sequences denoted by (e_i^0, e_i^1).
3. H_0 : initial state.
4. H_j^i : the intermediate state on the iterative chain.
5. (M_j^i, N_j^i) : a pair of message blocks each of b bits.
6. T : Table with three columns: a $(b/2)+1$-collision path, addition modulo 2^b of message blocks in that path and a value of 0 or 1.

Steps:

1. For $i = 1$ to b:
   - For $j = 1$ to $(b/2)+1$:
     - Find M_j^i and N_j^i such that $f(H_{j-1}^i, M_j^i) = f(H_{j-1}^i, N_j^i) = H_j^i$ where $H_0^1 = H_0$. That is, build a $(b/2)+1$-block multicollision where each block yields a collision on the iterative chain and there are $2^{(b/2)+1}$ different $(b/2)+1$-block sequences of blocks all hashing to the same intermediate state $H_{(b/2)+1}^i$ on the iterative chain.
   - Find a pair of paths from the different $(b/2)+1$-block sequences whose additive checksum differs by 2^{i-1} as follows:
     - T = empty table.
     - for $j = 1$ to $2^{(b/2)+1}$
       - $C_j^i \equiv \sum_{k=1}^{(b/2)+1} X_k^i \bmod 2^b$ where X_k^i can be M_k^i or N_k^i.
       - Add to T: $(C_j^i,\ 0,\ X_1^i || X_2^i || \ldots X_{(b/2)+1}^i)$
       - Add to T: $(C_j^i + 2^{i-1},\ 1,\ X_1^i || X_2^i || \ldots X_{(b/2)+1}^i)$.
     - Search T to find colliding paths between the entries with 0 and 1 in the second column of T. Let these paths of $(b/2)+1$ sequence of blocks be e_i^1 and e_i^0 where $e_i^1 \equiv e_i^0 + 2^{i-1} \bmod 2^b$.
   - $chunk[i] = (e_i^0, e_i^1)$.
2. Construct CCS by concatenating individual chunks each containing a pair of $(b/2)+1$ blocks that hash to the same intermediate state on the iterative chain. The CCS is $chunk[1] \,||\, chunk[2] \ldots \,||\, chunk[b]$.
3. The checksum at the end of the 2^b $(b/2)+1$-block collision can be forced to the desired checksum by choosing either of the sequences e_i^0 or e_i^1 from the

Рисунок 4 Алгоритм поиска мультиколлизий для хэш функции с контрольной суммой

Однако, в работе [10] был предложен более быстрый метод построения мультиколлизий для хэш функций с контрольными суммами:

- Пусть завершающее преобразование выглядит следующим образом $f(h_t, M) = g(h_t\ ,\ (m1 +\ m2 +\ m3 \ldots)\ mod\ 2^n)$, где $m_1 \ldots m_i$ – блоки исходного сообщения
- Определяем функцию $g'(h, M) = g(g(h, m),\ 2^n - m)$, где n – количество выходных бит
- Последовательно строим коллизии для сообщений (m_1^0, m_1^1), $(m_2^0, m_2^1) \ldots (m_t^0, m_2^1)$, так что $g'^{(h_{2i-2}, m_i^0)} = g'^{(h_{2i-2}, m_i^1)} = h_{2i}$
- Получаем 2^t сообщений в форме $(m_1,\ 2^n - m_1, \ldots, m_t,\ 2^n - m_t)$, где m_i – это один из двух блоков: m_i^0 или m_i^{1*}.

Трудоемкость предложенного метода составляет порядка $O(t\ 2^{\frac{n}{2}+1})$ операций вычисления функции сжатия.

ГЛАВА 3. ОПИСАНИЕ ПРОГРАММНОЙ РЕАЛИЗАЦИИ ЭКСПЕРИМЕНТА

3.1 Общее описание практической реализации

Для проведения экспериментальной верификации описанных теоретических результатов согласно общей схемы атаки (глава 2.3) было разработано соответствующее программное обеспечение, состоящее из:

- программной реализации алгоритма хэширования «Мора»;
- программной реализации параллельного лямбда-алгоритма Полларда для поиска одиночной колизии
- программной реализации алгоритма, реализующего построение мультиколлизий для данной хэш функции согласно общей схеме.

Программная реализация алгоритмов написана на языке C# с использование .Net Framework версии 4.7.2. Для распараллеливания расчетов использовалась community версия библиотеки Alea GPU v3.0.4, которая представляет собой .Net обертку над языком cuda. Данная библиотека позволила значительно ускорить процесс поиска коллизии, используя ресурсы

графического процессора. Исходный код программного обеспечения приведен в Приложении А. Все исходные коды также находятся в репозитории по пути [1].

Эксперименты были проведены на домашней рабочей станции, имеющей следующие характеристики:

- CPU Intel Core i5-8250 (4 ядра, 8 потоков, 1,8 ГГц)
- RAM - 8GB DDR4
- GPU - NVIDIA GeForce GTX1660
- Операционная система – Windows 10 x64

На экспериментальную часть исследовательской работы были поставлены следующие задачи:

- Разработка реализации хэш функции «Мора»
- Реализация метода поиска одиночной коллизии
- Реализация метода построения мультиколлизий
- Показать на практике использования вышеперечисленных методов за реальное время

3.1.1 Хэш «Мора»

Constants класс содержит значения постоянных переменных, а именно:

- L – матрица, используемая для умножения в линейном преобразовании множества двоичных векторов.
- C – итерационные константы
- SBox - нелинейное биективное преобразование множества двоичных векторов
- Tau – перестановка полубайт

ByteUtils предоставляет методы для работы с массивами байт:

- Xor – производит операцию исключающего или для 2х массивов
- RingSum – операция сложения 2ух элементов в поле $GF(2^4)$

- JoinBytes – объединяет 2 4х битных значение в одно 8ми битное
- SplitByte – разбивает 8ми битное значение на два 4 битных (значение дополняется до байта лидирующими нулями)

HashFunction реализация хэш функции «Мора»

- GetHash - функция, которая непосредственно производит хэширование
- G_n – функция сжатия
- L – функция линейного преобразования
- P – функция перестановки полубайт
- S – функция нелинейного преобразования
- KeySchedule – вызов LPSX от итерационных констант
- E - $E(k, m) = X[K9]LPSX[K8] \dots LPSX[K1](m)$
- ComputeHash – метод для вызова расчета хеша
- StringRepresentation – представление массива байт в виде строки

3.1.2 Алгоритм построения мультиколлизий

Program содержит метод main, являющийся точкой входа в программу

MagicInput используется для передачи параметров в/из ГПУ

Utils содержит вспомогательные методы

- Solve – генерирует 2^t сообщений из t входящих массивов сообщений
- GetRandomByteArray – генерирует случайный массив байт
- G_nGPU (G_n) - функция сжатия
- LGPU (L) - функция линейного преобразования
- PGPU (P) - функция перестановки полубайт
- SGPU (S) - функция нелинейного преобразования
- KeyScheduleGPU (KeySchedule) – вызов LPSX от итерационных констант

- SplitRightByte – возвращает последние 4 бита
- SplitLeftByte - возвращает первые 4 бита
- JoinBytes – объединяет два 4-х битных значение в одно 8-ми битное
- E - $E(k, m) = X[K9]LPSX[K8] \dots LPSX[K1](m)$

MultiCollisions описывает алгоритм построения атаки

- FindCollisions - выполняет все действия для поиска мультиколлизий
- MagicPoints - ищет одиночную коллизию
- FunctionGPU (Function) – производит вычисления вида $g(g(h, m, n),\ 2^{64} - m, n + 64)$
- GetMagicGPU (GetMagic) – вычисляет «магическое» значение, для поиска одиночной коллизии
- MagicCycleGPU (MagicCycle) – параллельно выполняет метод GetMagicGPU (GetMagic)

3.1.3 Тесты

TestMoraHash тестирование реализации хэш функции «Мора»

- TestG_n – проверяет равенство функций сжатия в реализации хэш функции и в реализации мультиколлизий
- TestHash – проверяет корректность реализованной хэш функции на основе контрольных значений из [3]

TestMultiCollisions тестирует алгоритм поиска мультиколлизий

- TestCollisions – проверяет корректность найденных мультиколлизий (мультиколлизии найдены заранее)

3.2 Описание используемых библиотек

В данном разделе описываются внешние библиотеки, задействованные в практической реализации.

3.2.1 Распараллеливание программы

Для ускорения поиска одиночной коллизии был выбран параллельный лямбда алгоритм Полларда, рассмотренный в главе x. Программа имеет возможность вычислений как на ресурсах центрального, так и на ресурсах графического процессоров. Для распараллеливания на центральном процессоре использовалась стандартная библиотека .Net. Для распараллеливания на графическом процессоре, использовалась библиотека Alea GPU v3.0.4, которая позволила значительно ускорить вычисления на домашней станции. Графический процессор не может работать напрямую с памятью центрального процессора, поэтому пришлось продублировать все функции, выполняемые в разных потоках. Для исключения ситуаций, при которых могли возникнуть ошибки, связанные с состоянием гонки (race condition), все переменные, для которых выполняются операции записи, были объявлены локально для каждого потока. Постоянные переменные используются только для чтения, поэтому не требуют синхронизации.

Для существенного ускорения эксперимента, был распараллелен метод поиска «магической» точки, для параллельного лямбда алгоритма Полларда – GetMagicGPU(GetMagic)

Так как данный метод несет в себе основную вычислительную нагрузку, его распараллеливание позволило ускорить вычисления в T раз, где T – количество потоков выполнения. Также, было принято решение использовать ресурсы графического процессора, т.к. это позволило выполнять одновременно намного больше вычислений, по сравнению с центральным процессором, немного потеряв в скорости выполнения одиночного вычисления.

3.2.2 Тестирование реализации

Проведение экспериментов требует корректности и точности полученных результатов. Для проверки программного обеспечения было реализовано тестирование. В качестве фреймворка тестирования была использована .Net библиотека NUnit версии 3.12.0. Для проведения тестирования был использован

подход AAA (arrange, act, assert), т.е. заполнение исходных данных (arrange), выполнение действий, которые хотим протестировать (act), проверка утверждений (assert), выражений, проверяющих истинность условия. Утверждения используются для проверки поведения кода. Результатом выполнения тестов может быть:

- Успех (все утверждения истинны)
- Неудача (одно или несколько утверждений ложны)
- Ошибка (во время выполнения теста, произошла критическая ошибка)

Если возникает ошибка, которую мы не учитываем в тесте, то считаем, что тест завершился неудачно и выводим сообщение об ошибке. Если результатом является неудача, то выводим утверждение, которое было ложно. В случае успеха, просто выводим сообщение, что тест завершен успешно.

ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

В данном разделе подробно рассмотрим детали реализации конкретных структур, классов, функций, используемых в программном обеспечении. Для более компактного изложения будем описывать используемые структуры группируя их по функциональному назначению.

Точкой входа в приложение является класс Program, на вход программе подаются параметры:

-h – получение справки

-t – количество циклов на поиск коллизий

-o – выходной файл, в который записываются полученные коллизии

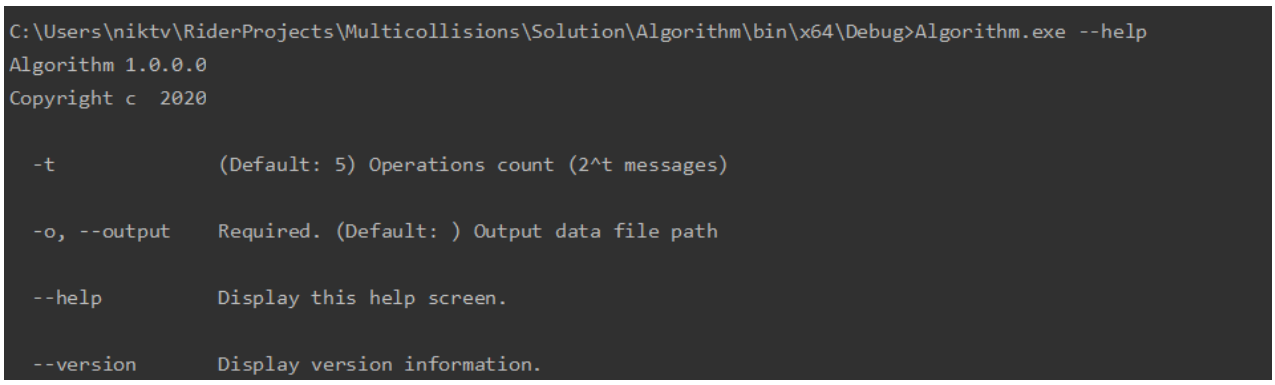

Рисунок 5 - получение справки

Рисунок 6 – Запуск с параметрами

Реализация хэш функции «Мора» находится в классе HashFunction, входной точкой является метод GetHash, который принимает на вход массив байт любой длины, на выходе выдает хэш - массив байт длиной 8. Для поиска мультиколлизий, в связи с особенностями работы графического процессора, в классе Utils были переписаны преобразования LPSX и функция сжатия, вместо массива байт, они работают с ulong значениями, т.к. размер ulong также составляет 8 байт. В тесте TestMoraHash.TestG_n выполнена проверка корректности переписанной функции сжатия.

Для реализации алгоритма поиска мультиколлизий используется класс MultiCollisions, с входной точкой FindCollisions, которая на вход принимает глубину поиска мультиколлизий (t), на выходе выдает конечное значение h, n и массив сообщений размером 2^t в виде массива байт. Для поиска одиночной колиизи используется параллельный лямбда алгоритм Полларда (метод MagicPoints). Он принимает предыдущие значения n и h, вычисленные на предыдущем шаге. Будем получать значения хешей для случайно заданных массивов байт в методе MagicCycleGPU и заносить их в ассоциативный массив, в котором ключ – хэш, значение – пара из начально переданного значения и счетчика. Заносить будем до тех пор, пока не встретим повторяющийся ключ, если встретили, то выравниваем значения по счетчикам повторно вызывая

функцию и возвращаем пару сообщений, образующих коллизию. Общая итерация была разбита на 2, для получения информационных сообщений о времени работы. На каждом новом цикле выводится сообщение о начале новой итерации «Begin», время поиска коллизии в миллисекундах, промежуточное значение n, h и пара найденных сообщений. В среднем время на поиск одной коллизии составляет около одного часа.

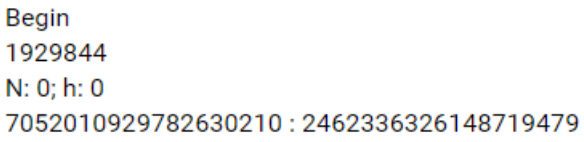

Рисунок 7 – Вывод итерации

Метод MagicCycleGPU параллельно вызывает метод GetMagicGPU, который вызывает метод FunctionGPU и увеличивает счетчик, пока результат работы функции не будет содержать 32 – sizelидирующих нулей, где $size = \log_2 T$, T – количество процессоров.

Метод FunctionGPU представляет собой вызов функции вида $g'(h, M, n) = g\_n(n + 64, g\_n(n, h, m),\ \ 2^n - m + 1)$.

После нахождения t пар сообщений (m_i^0, m_i^1), 2^t колллизий, которые дают одинаковое значение хэш функции, вида $(2^n - m_t, m_t, \ldots, 2^n - m_0, m_0)$, где m_i – это один из двух блоков: m_i^0или m_i^1 .

Эти сообщения записываются в файл, который был указан в параметре -o.

ЗАКЛЮЧЕНИЕ

В настоящей работе был рассмотрен алгоритм построения 2^k-мультиколлизий для хэш функции «Мора».

В практической реализации было показано, что, используя приведенные алгоритмы, можно строить 2^k-мультиколлизии для хэш функции «Мора» за эффективное время на обычной домашней рабочей станции.

По эффективному методу поиска мультиколлизии можно строить эффективные методы поиска второго прообраза и (второго) мультипрообраза функции хэширования [9, 11].

СПИСОК ЛИТЕРАТУРЫ

[1] Глухов М. М., Елизаров В. П., Нечаев А. А. Алгебра. Учебник. 2-е

изд., испр. и доп. -СПб.:Лань, 2015 г.

[2] Damgaard I. A design principle for hash functions // CRYPTO’89. Lect. Notes

Comput. Sci. — 1990. — V. 435. — P. 416–427.

[3] Бондакова О.С. Описание низкоресурсной хэш функции «Мора», г.Москва. 2020

[4] M. Bellare and T. Kohno. Hash function balance and its impact on birthday attacks. Lecture Notes in Computer Science 3027 (2004), 401–418 (Eurocrypt 2004)

[5] Paul C. van Oorschot and Michael J. Wiener Parallel Collision Search with Cryptanalytic Applications 1996 September 23, http://cr.yp.to/bib/1999/vanoorschot.pdf

[6] Колчин В. Ф., Севастьянов Б. А., Чистяков В. П. Случайные размещения. — М.: Наука, 1976.

[7] Aumasson J.-P. Faster multicollisions // INDOCRYPT’08. Lect. Notes

Comput. Sci. — 2008. — V. 5365. — P. 67–77.

[8] Antonie Joux Multicollisions in iterated hash functions. Application to cascade construction, DSSCI Crypto Lab, France https://www.iacr.org/archive/crypto2004/31520306/multicollisions.pdf

[9] Praveen Gauravaram and John Kelsey Linear-XOR and Additive Checksums Don’t Protect Damgard-Merkle Hashes from Generic Attacks

[10] Д. В. Матюхин, В. А. Шишкин, Некоторые методы анализа функций хэширования и их применение к алгоритму ГОСТ Р 34.11-94, Матем. вопр. криптогр., 2012, том 3, выпуск 4, 71–89

[11] Kelsey J., Kohno T. Herding hash functions and the Nostradamus attack //

EUROCRYPT’06. Lect. Notes Comput. Sci. — 2006. — V. 4004. — P. 183–200.

[11] https://github.com/nekitos911/Multicollisions

ПРИЛОЖЕНИЕ А. ИСХОДНЫЕ КОДЫ ПО

Листинг А.1 – ByteUtils.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using MoreLinq.Extensions;

namespace MoraHash
{
   public static class ByteUtils
   {
      public static byte[] Xor(this IEnumerable<byte> l, IEnumerable<byte> r) => l.Zip(r, (bl, br) => (bl, br)).Select(b => (byte)(b.bl ^ b.br)).ToArray();

      public static byte[] AddModulo64(this byte[] a, byte[] b)
      {
         byte[] temp = new byte[8];
         int i = 0, t = 0;
         byte[] tempA = new byte[8];
         byte[] tempB = new byte[8];
         Array.Copy(a, 0, tempA, 8 - a.Length, a.Length);
         Array.Copy(b, 0, tempB, 8 - b.Length, b.Length);
         for (i = 7; i >= 0; i--)
         {
            t = tempA[i] + tempB[i] + (t >> 8);
            temp[i] = (byte)(t & 0xFF);
```

```
        }
        return temp;
    }

    public static byte JoinBytes(byte hi, byte low) => (byte)((hi << 4) | (low & 0xffffffffL));

    public static (byte hi, byte low) SplitByte(byte val) => ((byte) (val >> 4), (byte) (val & 0xf));

    public static BitArray ToBitArray (this byte[] bytes, int bitCount) {
        BitArray ba = new BitArray (bitCount);
        for (int i = 0; i < bitCount; ++i) {
            ba.Set (i, ((bytes[i / 8] >> (i % 8)) & 0x01) > 0);
        }
        return ba;
    }
  }


}
```

Листинг А.2 – Constants.cs

```
namespace MoraHash
{
  public static class Constants
  {
    public static readonly ulong[] C =
    {
      0xc0164633575a9699,
      0x925b4ef49a5e7174,
      0x86a89cdcf673be26,
      0x1885558f0eaca3f1,
      0xdcfc5b89e35e8439,
      0x54b9edc789464d23,
      0xf80d49afde044bf9,
      0x8cbbdf71ccaa43f1,
      0xcb43af722cb520b9
    };

    // Нелинейное биективное преобразование множества двоичных векторов
    public static readonly int[] SBox =
    {
      15, 9, 1, 7, 13, 12, 2, 8, 6, 5, 14, 3, 0, 11, 4, 10
```

```
        };

        // Перестановка полубайт.
        public static readonly int[] Tau =
        {
            0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15
        };

        // Линейное преобразование множества двоичных векторов.
        public static readonly int[] L =
        {
            0x3a22, 0x8511, 0x4b99, 0x2cdd,
            0x483b, 0x248c, 0x1246, 0x9123,
            0x59e5, 0xbd7b, 0xcfac, 0x6e56,
            0xac52, 0x56b1, 0xb3c9, 0xc86d
        };
    }
}
```

Листинг А.3 – HashFunction.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using MoreLinq;

namespace MoraHash
{
    public class HashFunction
    {
        private static readonly int BlockSize = 8;

        private byte[] _n = new byte[BlockSize];
        private byte[] _sigma = new byte[BlockSize];
        private byte[] _iv = new byte[BlockSize];

        public byte[] P(byte[] state)
        {
            var tmp = new byte[16];
            var ret = new byte[8];
            for (int i = 0; i < state.Length; i++)
```

```
        {
            var splitted = ByteUtils.SplitByte(state[i]);
            tmp[i * 2] = splitted.hi;
            tmp[i * 2 + 1] = splitted.low;
        }

        var res = Enumerable.Range(0, tmp.Length).Select(i => tmp[Constants.Tau[i]]).ToArray();
        for (int i = 0; i < ret.Length; i++)
        {
            ret[i] = ByteUtils.JoinBytes(res[i * 2], res[i * 2 + 1]);
        }

        return ret;
    }

    public byte[] S(byte[] state)
    {
        var tmp = new byte[16];
        var ret = new byte[8];

        for (int i = 0; i < state.Length; i++)
        {
            var splitted = ByteUtils.SplitByte(state[i]);
            tmp[i * 2] = splitted.hi;
            tmp[i * 2 + 1] = splitted.low;
        }

        var res = Enumerable.Range(0, tmp.Length).Select(i => (byte)Constants.SBox[tmp[i]]).ToArray();

        for (int i = 0; i < ret.Length; i++)
        {
            ret[i] = ByteUtils.JoinBytes(res[i * 2], res[i * 2 + 1]);
        }

        return ret;
    }

    public byte[] L(byte[] state)
    {
        byte[] result = new byte[BlockSize];
//          Parallel.For(0, 4, i =>
        for (int i = 0; i < 4; i++)
```

```
        {
            int t = 0;
            byte[] tempArray = new byte[2];
            Array.Copy(state, i * 2, tempArray, 0, 2);
            tempArray = tempArray.Reverse().ToArray();
            var tempBits1 = tempArray.ToBitArray(16);
            bool[] tempBits = new bool[BlockSize * 2];
            tempBits1.CopyTo(tempBits, 0);
            tempBits = tempBits.Reverse().ToArray();

            for (int j = 0; j < tempBits.Length; j++)
            {
                if (tempBits[j])
                {
                    t ^= Constants.L[j];
                }
            }
            Array.Copy(BitConverter.GetBytes(t).Reverse().ToArray(), 2, result, i * 2, 2);
        }

        return result;
    }

    private byte[] GetHash(byte[] message)
    {
        var h = new byte[BlockSize];
        _n = new byte[BlockSize];
        _sigma = new byte[BlockSize];
        Array.Copy(_iv, h, BlockSize);

        byte[] n0 = new byte[BlockSize];

        IEnumerable<IEnumerable<byte>> blocks = message.Batch(BlockSize).ToArray();

        blocks.Where(block => block.Count() >= BlockSize).Reverse().ForEach(block =>
        {
            h = G_n(_n, h, block);
            _n = _n.AddModulo64(BitConverter.GetBytes((long)64).Reverse().ToArray());
            _sigma = _sigma.AddModulo64(block.ToArray());
        });

        var lastBlockSize = blocks.Last().Count();
```

```
            byte[] pad = MoreEnumerable
                .Append(new byte[lastBlockSize < BlockSize ? BlockSize - 1 - lastBlockSize : BlockSize - 1], (byte)
1).ToArray();

            byte[] m = pad
                .Concat(blocks.Where(block => block.Count() < BlockSize).DefaultIfEmpty(new
byte[0]).First()).ToArray();

            h = G_n(_n, h, m);

            var msgLen = BitConverter.GetBytes((long)(message.Length * 8)).Reverse();

            _n = _n.AddModulo64(msgLen.ToArray());

            _sigma = _sigma.AddModulo64(m);

            h = G_n(n0, h, _n);
            h = G_n(n0, h, _sigma);

            return h;
        }

        public byte[] G_n(IEnumerable<byte> N, IEnumerable<byte> h, IEnumerable<byte> m)
        {
            return E(L(P(S(h.Xor(N)))), m.ToArray()).Xor(h).Xor(m);
        }

        public byte[] E(byte[] k, byte[] m)
        {
            byte[] state = k.Xor(m);
            for (int i = 0; i < Constants.C.Length - 1; i++)
            {
                k = KeySchedule(k, i);
                state = L(P(S(state))).Xor(k);
            }
            return state;
        }

        private byte[] KeySchedule(byte[] k, int i) => L(P(S(k.Xor(BitConverter.GetBytes(Constants.C[i]).Reverse()))));

        public byte[] ComputeHash(byte[] message)
```

```
        {
            return GetHash(message.ToArray());
        }

        public static string StringRepresentation(byte[] input)
        {
            return BitConverter.ToString(input.ToArray()).Replace("-", string.Empty);
        }

    }
}
```

Листинг А.4 – TestMoraHash.cs

```
using System;
using System.Linq;
using Algorithm;
using NUnit.Framework;

namespace MoraHash.Tests
{
    public class TestMoraHash
    {
        [Test]
        public void TestG_n()
        {
            // h1 из контрольного примера
            var expected = new byte[] {0x1c, 0x9b, 0xea, 0x78, 0xab, 0x26, 0x32, 0x56};
            // дополненное сообщение из контрольного примера
            var m = new byte[] {0x01, 0xd4, 0x44, 0x90, 0x7e, 0xfb, 0x8c, 0xf7};
            var mora = new HashFunction();

            var result = mora.G_n(new byte[8], new byte[8], m);
            var resultMultiCol = MultiCollisions.G_n(0, 0, BitConverter.ToUInt64(m.Reverse().ToArray(), 0));

            Assert.True(expected.SequenceEqual(result));
            Assert.True(expected.SequenceEqual(BitConverter.GetBytes(resultMultiCol).Reverse()));
        }


        [Test]
        public void TestHash()
        {
```

```
            //h из контрольного примера
            var expected = new byte[] {0xb1, 0x7e, 0xb3, 0xf6, 0x0f, 0x29, 0x0e, 0xfd};
            // сообщение из контрольного примера
            var m = new byte[] { 0xd4, 0x44, 0x90, 0x7e, 0xfb, 0x8c, 0xf7};
            var mora = new HashFunction();
            var result = mora.ComputeHash(m);

            Assert.True(expected.SequenceEqual(result));
        }
    }
}
```

Листинг А.5 – Utils.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;
using Alea;
using MoraHash;

namespace Algorithm
{
    public class Utils
    {
        public static readonly GlobalArraySymbol<ulong> ConstC =
Gpu.DefineConstantArraySymbol<ulong>(Constants.C.Length);
        public static readonly GlobalArraySymbol<int> ConstSbox =
Gpu.DefineConstantArraySymbol<int>(Constants.SBox.Length);
        public static readonly GlobalArraySymbol<int> ConstTau =
Gpu.DefineConstantArraySymbol<int>(Constants.Tau.Length);
        public static readonly GlobalArraySymbol<int> ConstL =
Gpu.DefineConstantArraySymbol<int>(Constants.L.Length);
        private static RNGCryptoServiceProvider _rng = new RNGCryptoServiceProvider();

        /// <summary>
        /// Возвращает в solutions 2^t сообщений из t пар
        /// </summary>
        /// <param name="list"></param>
        /// <param name="solutions"></param>
        /// <param name="solution"></param>
        public static void Solve(List<List<ulong>> list, List<ulong[]> solutions, ulong[] solution)
        {
            if (solution.All(i => i != 0) && !solutions.Any(s => s.SequenceEqual(solution)))
```

```
            solutions.Add(solution);
        for (int i = 0; i < list.Count; i++)
        {
            if (solution[i] != 0)
                continue; // a caller up the hierarchy set this index to be a number
            for (int j = 0; j < list[i].Count; j++)
            {
                if (solution.Contains(list[i][j]))
                    continue;
                var solutionCopy = solution.ToArray();
                solutionCopy[i] = list[i][j];
                Solve(list, solutions, solutionCopy);
            }
        }
    }

    public static int SplitLeftByte(ulong data, int byteNum)
    {
        int shift = (8 * byteNum);
        var b = (data >> shift) & 0xff;
        return (int)(b >> 4);
    }

    public static int SplitRightByte(ulong data, int byteNum)
    {
        int shift = (8 * byteNum);
        var b = (data >> shift) & 0xff;
        return (int)(b & 0xf);
    }

    public static ulong JoinBytes(int hi, int low)
    {
        return (ulong)((hi << 4) | (low & 0xffffffffL));
    }

     #region G_n

    public static ulong P(ulong state)
    {
        ulong ret = 0;
        for (int i = 0; i < 8; i++)
        {
```

```
            int shift = (64 - 8 - 8 * i);
            var l = Constants.Tau[i * 2] % 2 == 0 ? SplitLeftByte(state, 7 - Constants.Tau[i * 2] / 2) :
SplitRightByte(state, 7 - Constants.Tau[i * 2] / 2);
            var r = Constants.Tau[i * 2 + 1] % 2 == 0 ? SplitLeftByte(state, 7 - Constants.Tau[i * 2 + 1] / 2) :
SplitRightByte(state, 7 - Constants.Tau[i * 2 + 1] / 2);
            ret |= JoinBytes(l, r) << shift;
        }

        return ret;
    }

    public static ulong S(ulong state)
    {
        ulong ret = 0;
        for (int i = 0; i < 8; i++)
        {
            int shift = (8 * i);
            ret |= JoinBytes(Constants.SBox[SplitLeftByte(state, i)], Constants.SBox[SplitRightByte(state, i)]) << shift;
        }

        return ret;
    }

    public static ulong L(ulong state)
    {
        ulong result = 0;

        for (int i = 0; i < 4; i++)
        {
            int t = 0;
            for(int k = 0; k < 16; k++) {
                if ((state & (1UL << (k + i * 16))) != 0)
                {
                    t ^= Constants.L[16 - k - 1];
                }
            }

            var data = (ulong) t;
            result |= data << (i * 16);
        }

        return result;
```

```
}

public static ulong E(ulong k, ulong m)
{
    ulong state = k ^ m;
    for (int i = 0; i < Constants.C.Length - 1; i++)
    {
        k = KeySchedule(k, Constants.C[i]);
        state = L(P(S(state))) ^ k;
    }
    return state;
}

public static ulong KeySchedule(ulong k, ulong c)
{
    return L(P(S(k ^ c)));
}

public static ulong G_n(ulong N, ulong h, ulong m)
{
    return E(L(P(S(h ^ N))), m) ^ h ^ m;
}

public static ulong Function(ulong N, ulong h, ulong m)
{
    var n1 = N + 64UL;
    return G_n(n1, G_n(N, h, m), ulong.MaxValue - m + 1);
}

#endregion

#region GPU G_n
public static ulong G_nGPU(ulong N, ulong h, ulong m)
{
    return EGPU(LGPU(PGPU(SGPU(h ^ N))), m) ^ h ^ m;
}

public static ulong PGPU(ulong state)
{
    ulong ret = 0;
    for (int i = 0; i < 8; i++)
    {
```

```
            int shift = (64 - 8 - 8 * i);
            var l = ConstTau[i * 2] % 2 == 0 ? SplitLeftByte(state, 7 - ConstTau[i * 2] / 2) : SplitRightByte(state, 7 - ConstTau[i * 2] / 2);
            var r = ConstTau[i * 2 + 1] % 2 == 0 ? SplitLeftByte(state, 7 - ConstTau[i * 2 + 1] / 2) : SplitRightByte(state, 7 - ConstTau[i * 2 + 1] / 2);
            ret |= JoinBytes(l, r) << shift;
        }

        return ret;
    }

    public static ulong SGPU(ulong state)
    {
        ulong ret = 0;
        for (int i = 0; i < 8; i++)
        {
            int shift = (8 * i);
            ret |= JoinBytes(ConstSbox[SplitLeftByte(state, i)], ConstSbox[SplitRightByte(state, i)]) << shift;
        }

        return ret;
    }

    public static ulong LGPU(ulong state)
    {
        ulong result = 0;

        for (int i = 0; i < 4; i++)
        {
            int t = 0;
            for(int k = 0; k < 16; k++) {
                if ((state & (1UL << (k + i * 16))) != 0)
                {
                    t ^= ConstL[16 - k - 1];
                }
            }

            var data = (ulong) t;
            result |= data << (i * 16);
        }

        return result;
```

```
        }

        public static ulong EGPU(ulong k, ulong m)
        {
            ulong state = k ^ m;
            for (int i = 0; i < ConstC.Length - 1; i++)
            {
                k = KeyScheduleGPU(k, ConstC[i]);
                state = LGPU(PGPU(SGPU(state))) ^ k;
            }
            return state;
        }

        public static ulong KeyScheduleGPU(ulong k, ulong c)
        {
            return LGPU(PGPU(SGPU((k ^ c))));
        }



        #endregion

        public static IEnumerable<byte> GetRandomByteArray(int arraySize)
        {
            byte[] buffer = new byte[arraySize];
            _rng.GetBytes(buffer);
            return buffer;
        }
    }
}
```

Листинг А.6 – MagicInput.cs

```
namespace Algorithm
{
    /**
     * Используется для передачи параметров в/из gpu
     */
    public struct MagicInput
    {
        public ulong X;
        public ulong X0;
        public long Counter;
```

```
    }
}
```

Листинг А.7 – MagicInput.cs

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Security.Cryptography;
using System.Threading.Tasks;
using Alea;
using Alea.CSharp;
using Alea.Parallel;
using MoraHash;
using MoreLinq.Extensions;
using ServiceStack;
using static Algorithm.Utils;

namespace Algorithm
{
    public class MultiCollisions
    {
        private async Task<ulong> AsyncG_n(ulong N, ulong h, ulong m)
        {
            return await Task.Run(() => G_n(N, h, m));
        }

        private MagicInput GetMagic(MagicInput input, ulong n, ulong h, int size)
        {
            var x = input.X0;
            var x0 = input.X0;
            var counter = 0;
            //if first 32 - size bits are zero, x - magic num
            while (x >> (32 + size) != 0)
            {
                x = Function(n, h, x);
                counter++;
            }

            return new MagicInput() {X = x, X0 = x0, Counter = counter};
        }
```

```
private MagicInput GetMagicGPU(MagicInput input, ulong n, ulong h, int size)
{
    var x = input.X0;
    var x0 = input.X0;
    var counter = 0;
    //if first 32 - size bits are zero, x - magic num
    while (x >> (32 + size) != 0)
    {
        x = FunctionGPU(n, h, x);
        counter++;
    }

    return new MagicInput() {X = x, X0 = x0, Counter = counter};
}

private MagicInput[] MagicCycle(MagicInput[] input, int maxSize, ulong n, ulong h)
{
    var t = (int)Math.Log(maxSize, 2);
    var result = new MagicInput[input.Length];

    Parallel.For(0, result.Length, (i) =>
    {
        result[i] = GetMagic(input[i], n, h, t);
    });
    return result;
}

[GpuManaged]
private MagicInput[] MagicCycleGPU(MagicInput[] input, int maxSize, ulong n, ulong h)
{
    var t = (int)Math.Log(maxSize, 2);
    var result = new MagicInput[input.Length];

    Gpu.Default.For(0, result.Length, i =>
    {
        result[i] = GetMagicGPU(input[i], n, h, t);
    });

    return result;
}
```

```
    private static ulong FunctionGPU(ulong N, ulong h, ulong m)
    {
        var n1 = N + 64UL;
        return G_nGPU(n1, G_nGPU(N, h, m), ulong.MaxValue - m + 1);
    }

    private (ulong, ulong) MagicPoints(ulong n, ulong h)
    {
        (ulong, ulong) retVal = (0, 0);
        var dict = new ConcurrentDictionary<ulong, (ulong, long)>();
        var maxSize = 1_050_000;
        var step = 525_000;

        while (true)
        {
            Console.WriteLine("Begin");
            var input = new ulong[maxSize].AsParallel().Select(data =>
BitConverter.ToUInt64(GetRandomByteArray(8).ToArray(), 0)).Select(data => new MagicInput() {X0 =
data}).ToArray();
            for (int i = 0; i < maxSize; i+= step)
            {
                var st = new Stopwatch();
                st.Start();
                var result = MagicCycleGPU(input.Skip(i).Take(step).ToArray(), maxSize, n, h);
                st.Stop();
                var time = st.ElapsedMilliseconds;
                Console.WriteLine(time);

                Parallel.ForEach(result, (res, state) =>
                {
                    var x = res.X;
                    var x0 = res.X0;
                    var count = res.Counter;

                    if (dict.TryGetValue(x, out var value))
                    {
                        var x1 = value.Item1;
                        var count1 = value.Item2;

                        while (count1 > count)
                        {
                            x1 = Function(n, h, x1);
```

```
                    count1--;
                }

                while (count > count1)
                {
                    x0 = Function(n, h, x0);
                    count--;
                }

                while (true)
                {
                    var next = Function(n, h, x1);
                    var next2 = Function(n, h, x0);

                    if (next == next2) break;

                    x1 = next;
                    x0 = next2;
                }

                if (x1 != 0 && x0 != 0)
                {
                    retVal.Item1 = x1;
                    retVal.Item2 = x0;
                    state.Break();
                }
            }
            else
            {
                dict.TryAdd(x, (x0, count));
            }
        });

        if (retVal != (0, 0)) break;
    }
    if (retVal != (0, 0)) break;
  }

  return retVal;
}

private (ulong, ulong) Brent(int[] l, int[] sBox, int[] tau, ulong[] c)
```

```
{
    var power = 1;
    var lam = 1;
    var x0 = G_n(0, 0, BitConverter.ToUInt64(GetRandomByteArray(8).ToArray(), 0));

    var tortoise = x0;
    var hare = G_n(0, 0, x0);

    while (tortoise != hare)
    {
        if (power == lam)
        {
            tortoise = hare;
            power *= 2;
            lam = 0;
        }
        hare = G_n(0, 0, hare);
        lam++;
    }

    tortoise = hare = x0;

    for (int i = 0; i < lam; i++)
    {
        hare = G_n(0, 0, hare);
    }

    while (true)
    {
        var nextTask = AsyncG_n(0, 0, tortoise);
        var next2Task = AsyncG_n(0, 0, hare);

        Task.WaitAll(nextTask, next2Task);

        if (nextTask.Result == next2Task.Result) break;

        tortoise = nextTask.Result;
        hare = next2Task.Result;
    }

    return (tortoise, hare);
}
```

```
private (ulong, ulong) Floyd(int[] l, int[] sBox, int[] tau, ulong[] c)
{
    var x0 = G_n(0, 0, BitConverter.ToUInt64(GetRandomByteArray(8).ToArray(), 0));

    var hare = x0;
    var tortoise = x0;

    while (true)
    {
        hare = G_n(0, 0, G_n(0, 0, hare));
        tortoise = G_n(0, 0, tortoise);

        if (hare == tortoise)
        {
            break;
        }
    }

    tortoise = x0;

    while (true)
    {
        var nextTask = G_n(0, 0, tortoise);
        var next2Task = G_n(0, 0, hare);

        if (nextTask == next2Task) break;
    }

    return (tortoise, hare);
}

public (byte[][] messages, ulong h, ulong n) FindCollisions(int t)
{
    var gpu = Gpu.Default;

    gpu.Copy(Constants.C, ConstC);
    gpu.Copy(Constants.L, ConstL);
    gpu.Copy(Constants.SBox, ConstSbox);
    gpu.Copy(Constants.Tau, ConstTau);

    var lst = new List<List<ulong>>();
```

```
        var solutions = new List<ulong[]>();

        var h = 0UL;
        var n = 0UL;

        for (int i = 0; i < t; i++)
        {
            var r = MagicPoints(n, h);
            Console.WriteLine($"N: {n}; h: {h}");
            Console.WriteLine($"{r.Item1} : {r.Item2}");
            h = Function(n, h, r.Item1);
            n += 128UL;
            lst.Add(new List<ulong>() {r.Item1, r.Item2});
        }

        lst.Reverse();

        var solution = new ulong[lst.Count];
        Utils.Solve(lst, solutions, solution);

        var messages = solutions.Select(seq =>
        {
            IEnumerable<byte> ret = new byte[0];

            ret = seq.Aggregate(ret, (current, value) =>
                current.Concat(BitConverter.GetBytes(ulong.MaxValue - value + 1).Reverse())
                    .Concat(BitConverter.GetBytes(value).Reverse()));

            return ret.ToArray();
        }).ToArray();

        return (messages, h, n);
    }
  }
}
```