МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»
Факультет информационных технологий
Кафедра информационных систем и технологий

УТВЕРЖДАЮ
Заведующий кафедрой
_____ В.В. Смелов
подпись инициалы и фамилия
“___”_____2022
г.

ЗАДАНИЕ

к курсовому проектированию по дисциплине

"Защита информации и надежность информационных систем"

Специальность 1-40 05 01 03 «Информационные системы и технологии (издательско-полиграфический комплекс)»

Группа: 1

Студент: Савельев Дмитрий Витальевич

Тема: «Реализация и исследование алгоритма хеширования Lyra2»

1. Срок сдачи студентом законченной работы: «10» мая 2022г.

2. Исходные данные к проекту:

2.1. Функционально должны быть выполнены следующие задачи:

- Реализовать методы для хеширования данных.
- Разработать программное средство реализации данного метода или алгоритма.
- Составить руководство пользователя.

2.2. Требования:

- Необходимо провести аналитический обзор литературы по теме проекта
- Необходимо описать сферу применимости метода
- Программное средство может быть разработано на любом языке
- Архитектура приложения выбирается разработчиком
- Листинги проекта должны содержать комментарии

3. Содержание расчетно-пояснительной записки:

- Введение
- Постановка задачи
- Описание метода
- Описание программного средства
- Тестирование программного средства
- Руководство пользователя
- Заключение
- Список используемых источников
- Приложения

4. Форма представления выполненного курсового проекта:

- Теоретическая часть курсового проекта должна быть представлена в формате MS Word.
- Оформление записки должно быть согласно правилам.
- Необходимые схемы, диаграммы и рисунки допускается делать в MS Office Visio или копии экрана (интерфейс).
- Полные листинги проекта представляются в приложении.
- К записке необходимо приложить CD (DVD), который должен содержать: пояснительную записку, листинги и файлы базы данных.

Календарный план

№ п/п	Наименование этапов курсового проекта	Срок выполнения этапов проекта	Примечание
1	Введение	12.02.2022	
2	Аналитический обзор литературы по теме проекта	28.02.2022	
3	Разработка метода	10.03.2022	
4	Разработка прототипа программного средства	24.03.2022	
5	Разработка программного средства	31.03.2022	
6	Тестирование программного средства	14.04.2022	
7	Написание руководства пользователя	21.04.2022	
8	Оформление пояснительной записки	28.04.2022	
9	Сдача проекта	10.05.2022	

5. Дата выдачи задания «15» февраля 2022г.

Руководитель _____ ____*В.О. Берников*_____

(подпись)

Задание принял к исполнению _15.03.2022_____

(дата и подпись студента)

Содержание

Введение

Аутентификация пользователей – это один из самых важных элементов в современной компьютерной безопасности. Даже несмотря на существование механизмов аутентификации, основанных на биометрических данных («что есть пользователь») или физические устройства, такие как смарт-карты («что есть у пользователя»), наиболее широко распространенный способ все равно основывается на секретных паролях («что пользователь знает»). Это происходит из-за того, что аутентификация, основанная на паролях, остается самым удобным и эффективным способом поддержания общего секрета между пользователем и компьютерной системой. К счастью или сожалению, и несмотря на существование множества предложений по их замене, это преобладание паролей в качестве одного и часто единственного способа аутентификации скорее всего не изменится в ближайшем будущем.

Системы, основанные на паролях, обычно используют некоторые криптографические алгоритмы, которые позволяют генерировать псевдослучайные строки бит на основе паролей, так называемая криптографические хеш-функции (англ. password hashing scheme – PHS) или функция формирования ключа (англ. key derivation function – KDF). Обычно, вывод хеш-функции используется одним из двух способов: он может храниться в форме «токена» для будущих проверок пароля или использоваться в качестве секретного ключа для зашифрования и/или аутентификации данных. В любом случае, такие решения используют внутренние односторонние (например, хеш) функции, так что восстановление пароля из вывода функции получения ключа вычислительно неосуществимо.

Несмотря на популярность аутентификации, основанной на пароле, тот факт, что большинство пользователей выбирают довольно коротки и простые строки в качестве паролей, приводит к серьезным проблемам: они обычно имеют гораздо меньшую энтропию нежели рекомендуется для криптографических ключей. Согласно анализу 5 миллионов слитых паролей Gmail, средняя энтропия их составила 21.6 бит, при том, что для соответствия категории «сильный пароль», он должен иметь энтропию более 60 бит, а «очень сильный» - более 128 бит. Исследование же 2007 года на основании 544960 паролей реальных пользователей получила среднюю энтропию порядка 40.54 бит. Такая слабость паролей сильно упрощает множество видов «брутфорс» атак, такие как атака со словарем и полный перебор, позволяя хакерам полностью обходить свойство неинверсивного процесса хеширования. Как пример, хакеры могут применять криптографические хеш-функции для списка наиболее используемых паролей пока результат не совпадет с локально хранимым токеном или действительным ключом шифрования/аутентификации. Осуществимость таких атак зависит в основном от количества ресурсов доступных для хакеров, кто может

ускорить процесс распараллеливая множество тестов. Такие атаки обычно получают выгоду от платформ, оснащенных большим количеством процессорных ядер, например современных видеокарт или специализированного оборудования.

Прямолинейный подход для решения этой проблемы: заставлять пользователей выбирать сложные пароли. Однако это плохой подход, так как такие пароли сложнее запоминать, а значит пользователи должны их где-то записать, что полностью перечеркивает весь смысл аутентификации. Для решения этих проблем современные решения хеширования паролей обычно используют механизмы для увеличения «стоимости» брутфорс атак. Такие методы как PBKDF2 и bcrypt, к примеру, включают конфигурируемые параметры, которые контролируют количество осуществляемых итераций, позволяя пользователям изменять время, требуемое для процесса хеширования пароля. Более актуальный алгоритм scrypt позволяет пользователям контролировать помимо времени обработки использование памяти, усложняя тем самым восстановление пароля. Существует, однако, значительный интерес в исследование и развитии новых и лучших алгоритмов хеширования.

В результате этих проблем М. Симплисио младший, Л. К. Альмейда, Э. Р. Андраде, П. К. Ф. Сантуш и П. С. Л. М. Баррето из Политехнической школы Университета Сан-Паулу разработали на основе своего алгоритма Lyra улучшенную версию – Lyra2. Lyra2 предлагает безопасность, эффективность и гибкость Lyra, включает: возможность настраивать желаемое количество используемой памяти, время выполнения и использование параллелизма алгоритмом, возможность обеспечения большего использования памяти с временем обработки, аналогичным таковому, полученному с помощью scrypt.

Lyra2 использует в своей реализации Sponge функции. Sponge функции представляют собой хеш-функции, которые могут итеративно обрабатывать входные и выходные данные произвольной длины. Они включают в себя перестановку f фиксированное длины, которая работает с внутренним состоянием представленным последовательностью размером b+c битов, состоящим из битрейта длиной b и емкостью c, в сочетании с входными данными M, разрезанными на b-битные блоки. Sponge функция включает в себя операции absorb, которая заключается в итеративном применении f к внутреннему состоянию после применения операции XOR битрейта с каждым из b-битных входных битов. Операция squeeze, в свою очередь, представляет собой применение f ко всему внутреннему состоянию и последующую выдачу битрейта на выход, эта операции будет выполняться пока заданное пользователем количество битов не будет предоставлено в качестве вывода.

1 Основы криптографических хеш функций и конкуренты

1.1 Sponge-функция

В основе алгоритма Lyra2, как и многих других хеш-функций, лежит концепция Sponge-функций. Элегантность дизайна губки также мотивировала к созданию более общих структур, таких как семейство функций Parazoa. Возможно, гибкость была одной из причин, по которой Keccak был избран в качестве нового SHA-3.

Вкратце, sponge-функции предлагают интересный путь построения хеш-функций с произвольной длиной входа и выхода. Такие функции построены на так называемых sponge-конструкциях, итеративном способе операций, который использует фиксированное число перестановок f и дополняющее правило pad. Более подробно, как изображено на рис. 2.1, sponge-функции зависят от внутреннего состояния w = b+c битов, изначально установленных в нули, и оперируют с дополненным входом M, разделенного на b-битные блоки. Это выполняется за счет итеративного применения f к внутреннему состоянию губки, операция чередуется с вводом входных битов (впитывание – (англ.) absorbing) или последующим поиском выходных битов (выжимание – (англ.) squeezing). Процесс останавливается, когда весь вход пройдет фазу абсорбции и образует результирующую l-битную выходную строку. Обычно, f-перестановка сама по себе итеративна, параметризуется количеством раундов (например, 24 для Keccak, оперирующего с 64-битными словами).

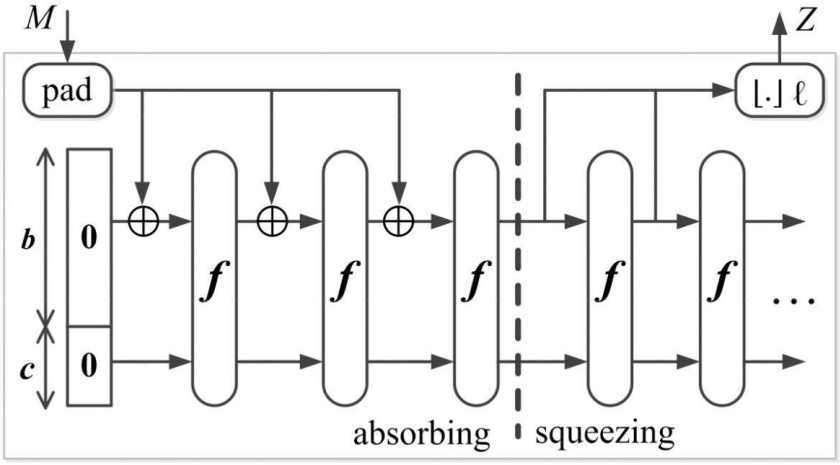

Рисунок 1.1 – Обзор sponge-конструкции Z = [f, pad, b](M, l)

Внутреннее состояние губки состоит из двух частей: b-битная внешняя часть, которая взаимодействует напрямую с входом губки, и c-битной внутренней части, которая зависит только от входа посредством f-перестановки. Параметры w, b и c называются, соответственно, шириной, битрейтом и емкостью губки.

1.2 Дуплексная конструкция

Похожая структура, полученная из концепции губок – дуплексная конструкция, представленная на рисунке 2.2.

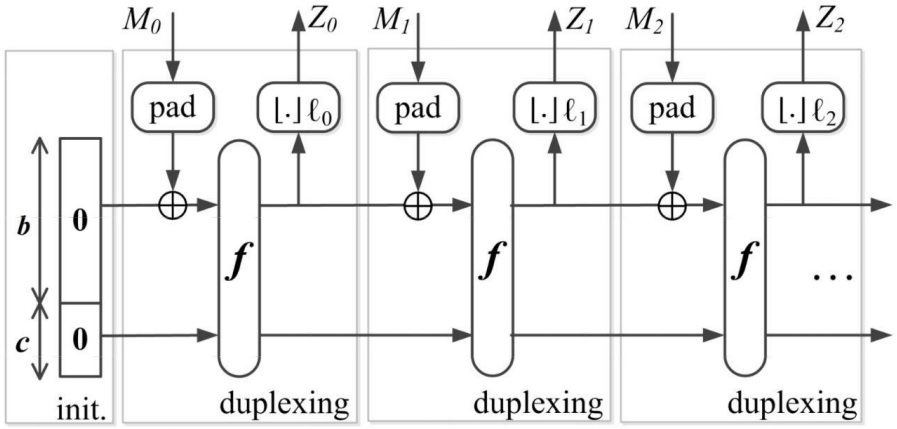

Рисунок 1.2 – Обзор дуплексной конструкции

В отличие от обычных губок, которые не имеют состояния между вызовами, дуплексные же принимают входные строки переменной длины и предлагают выход переменной длины, который зависит от всех полученных входных данных. Другими словами, хотя внутреннее состояние дуплексной функции заполнено нулями во время инициализации, оно сохраняется после каждого вызова дуплексного объекта, а не повторно сбрасывается. С этой стороны входная строка M должна быть достаточно короткая, чтобы поместиться в один b-битный блок после расширения, и длина выхода l должна удовлетворять $l \leq b$.

Для алгоритма Lyra2 предлагается выбрать одну из трех sponge-функций: blake2b, blamka и blamka с уменьшенным числом раундов. В программе я реализовал губки с помощью наследования. Каждый алгоритм представляет собой отдельный класс, который наследуют общий класс Sponge (приложение А), в котором выполняются общие для всех губок действия.

На листинге 1.1 показано, как с помощью смены порядка с little-endian на big-endian, операциями XOR и сдвига влево происходит установка состояния для губки на основе blake2b.

```
public void G(final int a, final int b, final int c, final int
d) {
    state[a] = mem.flip(mem.flip(state[a]) +
mem.flip(state[b]));
    state[d] = rotl64(state[d] ^ state[a], 32);

    state[c] = mem.flip(mem.flip(state[c]) +
mem.flip(state[d]));
    state[b] = rotl64(state[b] ^ state[c], 24);
```

Окончание листинга 1.1

```
    state[a] = mem.flip(mem.flip(state[a]) +
mem.flip(state[b]));
    state[d] = rotl64(state[d] ^ state[a], 16);

    state[c] = mem.flip(mem.flip(state[c]) +
mem.flip(state[d]));
    state[b] = mem.flip(rotr64(mem.flip(state[b] ^ state[c]),
63));
}
```

Листинг 1.1 – Преобразование с помощью blake2b

На основе губки blamka можно выбрать одну из двух спецификаций: с полным или уменьшенным количеством раундов (листинг 1.2).

```
private void diagonalize() {
    long t0, t1, t2;
    t0 = state[4];
    state[4] = state[5];
    state[5] = state[6];
    state[6] = state[7];
    state[7] = t0;
    t0 = state[8];
    t1 = state[9];
    state[ 8] = state[10];
    state[ 9] = state[11];
    state[10] = t0;
    state[11] = t1;
    t0 = state[12];
    t1 = state[13];
    t2 = state[14];
    state[12] = state[15];
    state[13] = t0;
    state[14] = t1;
    state[15] = t2;
}
@Override
public void sponge_lyra(final int rounds) {
    for (int round = 0; round != rounds; ++round) {
        G(0, 4,  8, 12);
        G(1, 5,  9, 13);
        G(2, 6, 10, 14);
        G(3, 7, 11, 15);
        diagonalize();
    }
}
@Override
public void sponge_lyra() {
    sponge_lyra(FULL_ROUNDS);
}
```

Листинг 1.2 – Преобразование с помощью blaMka с половиной раундов

BlaMka с половиной раундов, по сути, представляет собой измененную версию оригинальной blaMka, только использующую предварительную перестановку состояний, но использующая функцию G основной blaMka.

Сама blaMka – это измененный blake2b. Реализация G-функции губки blaMka представлена в листинге 1.3.

```
private long fBlaMka(final long x, final long y) {
    long lessX = 0x00000000FFFFFFFFL & x;
    long lessY = 0x00000000FFFFFFFFL & y;

    lessX *= lessY;

    lessX <<= 1;

    return lessX + x + y;
}

@Override
public void G(final int a, final int b, final int c, final int
d) {
    state[a] = mem.flip(fBlaMka(mem.flip(state[a]),
mem.flip(state[b])));
    state[d] = rotl64(state[d] ^ state[a], 32);

    state[c] = mem.flip(fBlaMka(mem.flip(state[c]),
mem.flip(state[d])));
    state[b] = rotl64(state[b] ^ state[c], 24);

    state[a] = mem.flip(fBlaMka(mem.flip(state[a]),
mem.flip(state[b])));
    state[d] = rotl64(state[d] ^ state[a], 16);

    state[c] = mem.flip(fBlaMka(mem.flip(state[c]),
mem.flip(state[d])));
    // Cannot use the left rotation trick here: 63 % 8 != 0, so
    // individual bytes do not stay the same, they change too.
    state[b] = mem.flip(rotr64(mem.flip(state[b] ^ state[c]),
63));
}
```

Листинг 1.3 – Инициализация состояний с помощью blaMka

Цель использования blaMka вместо blake2b – добавление умножения в алгоритм. Это позволяет еще больше улучшить диффузию битов и увеличить зависимость между этими битами, почти не влияя на скорость.

2 Криптографические хеш-функции

Как описывалось ранее, главное требование к криптографическим хеш-функциям – однонаправленность, что делает восстановление пароля из выхода функции вычислительно невозможным. Более того, выход хороших хеш-функций ожидается неразличимым со случайной последовательностью битов, предотвращая отбрасывание злоумышленником части пространства пароля на основе предполагаемых шаблонов. В принципе, эти рекомендации могут быть легко выполнены простым использованием безопасных хеш-функций, которые сами по себе гарантируют, что лучшая стратегия против сформированного ключа лежит через «брутфорс».

Поэтому современные включают технологии, позволяющие увеличить стоимость «брутфорс» атак. Первая стратегия по осуществлению этого – брать в качестве входа не только подлежащие запоминанию пользовательские пароли, но также и последовательность случайных битов, так называемую соль. Существование подобных случайных переменных мешают атакам, основанных на заранее созданных таблицах с типичными паролями, так как злоумышленник вынужден создавать такие таблицы для каждой возможной соли. Таким образом, соль можно рассматривать как индекс большого набора возможных ключей, полученных из пароля, и ее не нужно запоминать или держать в секрете.

Вторая стратегия – это целенаправленное увеличение стоимости проверки каждого пароля в рамках ресурсов компьютера, таких как процессорное время и/или использование памяти. Это также увеличивает стоимость аутентификации обычного пользователя, вводящего правильный пароль, означая что алгоритму нужно быть сконфигурированным таким образом, чтобы нагрузка на целевую платформу была наименее заметная для людей. Поэтому легитимные пользователи и их платформы – это именно то, что накладывает верхний лимит на вычислительную стоимость хеш-функции. Например, люди, запускающие единственный экземпляр функции получения ключа, вряд ли будут считать проблемой, если хеширование пароля займет одну секунду и 20 МБ ОЗУ. С другой стороны благодаря этому, предполагая, что процесс хеширования не может быть разделен на более маленькие распараллеленные задачи, достигается требования 20 ГБ ОЗУ и 1000 процессорных единиц для хеширования 1000 паролей с сопоставимой с пользовательской скоростью.

И, наконец, третья стратегия, особенно полезная, когда функция получения ключа использует и процессорное время, и использование памяти – использование архитектуры с низкой параллелизуемостью. Причины следующие. Для злоумышленника с доступом к p процессорным ядрам, обычно нет разницы между разделение каждого пароля на отдельное ядро или распараллеливанием хеширования одного пароля, предполагая, что это выполнится в p раз быстрее: в обоих случаях общая пропускная способность угадывания пароля одинакова. Однако, последовательная архитектура,

которая влияет на конфигурационное использование памяти, налагает интересный штраф злоумышленникам, у которых нет достаточного количества памяти для запуска p процессов параллельно. К примеру, предполагая, что тестирование пароля использует m байтов памяти и выполняется за n инструкций. Также предполагая, что у злоумышленника есть устройство с 100m байтов памяти и 1000 ядер, каждое из которых выполняет n инструкций в секунду. В этом случае до 100 попыток может быть протестировано за секунду против строго последовательного алгоритма (один на ядро), остальные 900 ядер остаются неиспользованными, так как им не хватает памяти для работы.

2.1 Scrypt

Основные решения для хеширования паролей: PBKDF2, bcrypt и scrypt. Но из них лишь scrypt использует и память, и стоимость вычислений, становясь таким образом напрямую сравнимым с Lyra2.

Архитектура scrypt фокусируется на связи стоимости по памяти и по времени. Для этого scrypt использует концепцию функций «последовательной памяти»: алгоритм, который асимптотически использует столько памяти, сколько требуется для операций и для которых параллельная реализация не может асимптотически давать значительно меньшую стоимость. В результате, если количество операций и количество используемой памяти в обычных операций оба алгоритма имеют стоимость O(R), сложность атаки, для которой использование памяти снижается до O(1), становится равной $O(R^2)$, где R – параметр системы.

Следующие шаги составляют операции алгоритма scrypt (рис. 3.1). Сначала он инициализирует p блоков B_i длиной b. Это делается с использованием алгоритма PBKDF2 с HMAC-SHA256 лежащей в основе хеш-функции и единичной итерации. Затем каждый B_i обрабатывается (итерационно или параллельно) функцией «последовательной памяти» ROMix. Обычно ROMix инициализируется массивом M из R элементов длины b итеративно хешируемых B_i. Затем он посещает R позиций M в случайном обновляющемся внутреннем состоянии переменной X в течение процесса в порядке установления этого эти позиции, наконец, доступны в памяти. Хеш-функция, использованная ROMix называется BlockMix, которая эмулирует функцию, имеющую произвольной длины (размера b) вход и выход. Это достигается с помощью поточного шифра Salsa20/8, чья выходная длина h = 512. Когда p процессов ROMix закончены, B_i блоки используются в качестве соли в одной финальной итерации алгоритма PBKDF2, выводя ключ K.

```
PARAM: h   ▷ BlockMix's internal hash function output length
INPUT: pwd   ▷ The password
INPUT: salt   ▷ A random salt
INPUT: k   ▷ The key length
INPUT: b   ▷ The block size, satisfying b = 2r · h
INPUT: R   ▷ Cost parameter (memory usage and processing time)
INPUT: p   ▷ Parallelism parameter
OUTPUT: K   ▷ The password-derived key
(B_0...B_{p-1}) ←PBKDF2_{HMAC-SHA-256}(pwd, salt, 1, p · b)
for i ← 0 to p − 1 do
    B_i ←ROMIX(B_i, R)
end for
K ←PBKDF2_{HMAC-SHA-256}(pwd, B_0 ‖ B_1 ‖ ... ‖ B_{p-1}, 1, k)
return K   ▷ Outputs the k-long key
function ROMIX(B, R)   ▷ Sequential memory-hard function
    X ← B
    for i ← 0 to R − 1 do   ▷ Initializes memory array M
        V_i ← X   ;   X ←BLOCKMIX(X)
    end for
    for i ← 0 to R − 1 do   ▷ Reads random positions of M
        j ← Integerify(X) mod R
        X ←BLOCKMIX(X ⊕ M_j)
    end for
    return X
end function
function BLOCKMIX(B)   ▷ b-long in/output hash function
    Z ← B_{2r−1}   ▷ r = b/2h, where h = 512 for Salsa20/8
    for i ← 0 to 2r − 1 do
        Z ← Hash(Z ⊕ B_i)   ;   Y_i ← Z
    end for
    return (Y_0, Y_2, ..., Y_{2r−2}, Y_1, Y_3, Y_{2r−1})
end function
```

Рисунок 2.1 – Псевдокод алгоритма scrypt

Scrypt представляет очень интересную архитектуру, будучи одним из немногих решений, которые позволяют настраивать стоимость как по памяти, так и по времени. Один из главных недостатков — это, вероятно, тот факт, что это сильно связывает требования памяти и вычисления для обычных пользователей. Конкретно, архитектура scrypt предотвращает пользователей от затрат по времени обработки во время поддержания фиксированного количества используемой памяти, пока они готовы увеличивать параметр p и позволяют в дальнейшем распараллеливать для использования злоумышленниками. Следующее неудобство с scrypt – это факт того, что в своей основе алгоритм использует 2 разные хеш-функции HMAC-SHA-256 (для алгоритма PBKDF2) и Salsa20/8 (как основа функции BlockMix), используемых для увеличения сложности реализации. Наконец, даже несмотря на известные уязвимости Salsa20/8 не представляют опасности для scrypt, использование лучших альтернатив было бы как минимум рекомендовано, особенно учитывая, что структура схемы не претерпела серьезных изменений во внутреннем алгоритме хеширования BlockMix. С этой стороны sponge-функция может сама по себе быть альтернативой, поскольку она позволяет использовать вход и выход любой

длины, что позволяет полностью заменить BlockMix.

Невзирая на эти недостатки алгоритм scrypt нашел свое применение в криптовалютах. В частности, его используют такие криптовалюты, как Litecoin и немало известный Dogecoin. Майнинг этих криптовалют происходит с помощью видеокарт, так как в таких задачах они дают большую производительность нежели процессоры.

Вдохновленная архитектурой scrypt, Lyra2 построена на свойствах губок, чтобы обеспечить не только более простое, но и более безопасное решение. Конечно, Lyra2 стоит на стороне концепции жесткой памяти: вычислительная стоимость для злоумышленника, использующего меньше памяти, чем определено алгоритмом, способствует ее более чем квадратичному росту, превосходящему наилучшие результаты, полученные с помощью scrypt, а также срывая атаку компромисса между временем/памятью/данными. Эта характеристика должна отговорить злоумышленников от размена используемой памяти на вычислительное время, что является целью функций получения ключа, в которых использование обоих ресурсов является настраиваемым. В дополнение Lyra2 увеличивает количество используемой памяти за такое же процессорное время, увеличивая также и стоимость обычных нападений злоумышленников по сравнению с scrypt.

3 Разбор Lyra2

Как и любая другая функция получения ключа, Lyra2 принимает в качестве входа соль и пароль, создавая псевдослучайный выход, который может использоваться в качестве ключевого материала для криптографических алгоритмов или для строк аутентификации. Внутренне структура памяти организована в виде матрицы, которая должна оставаться в памяти во время всего процесса хеширования пароля: пока ее ячейки итеративно считывают и записывают в них данные, выбрасывание ячейки для сохранения памяти ведет к необходимости пересчитывать ее значение каждый раз, когда происходит обращение. Конструкция и просмотр матрицы выполняется с использованием комбинаций с состоянием операций впитывания, выжимания и дуплексирования лежащей в основе губки (то есть ее внутреннее состояние никогда не сбрасывается в нули), убеждая в последовательной природу всего процесса. Также количество повторных посещений ячеек матрицы после инициализации определяется пользователем, позволяя времени выполнения Lyra2 быть хорошо адаптированным под ресурсы целевой платформы.

3.1 Структура алгоритма

Весь алгоритм Lyra2 состоит из нескольких этапов, таких как: Bootstrapping, Setup, Wandering и Wrap-up.

На стадии Bootstrapping происходит инициализация внутренних переменных sponge-функции. На вход подаются пароль, соль, затраты по времени, по памяти.

Матрица памяти инициализируется на стадии Setup.

Во время стадии Wandering ячейки матрицы памяти случайным образом перезаписываются. Количество итераций зависит от выбранного коэффициента затрат по времени.

И, наконец последний этап Wrap-up применяет операции absorb и squeeze. В результате мы получаем псевдослучайную последовательность заданного размера.

Алгоритм Lyra2 в виде псевдокода представлен в приложении Б.

А теперь более подробно про каждый из этапов.

3.1.1 Bootstrapping

Как уже отмечалось выше, Bootstrapping представляет собой установку начальных значений губки и внутренних переменных (строки 1-6 прил. А). Набор переменных {gap, stp, wnd, sqrt, $prev^0$, row^1, $prev^1$} инициализированные в строках 5 и 6 используются для следующей стадии алгоритма.

Губка Lyra2 инициализируется впитыванием дополненного пароля и соли вместе с битовой строкой params. Паттерн дополнения, используемый

Lyra2, такой же, как и у Keccak – дополнение 10*1. Он означает, что мы добавляем к сообщению единичный бит, а затем столько нулей сколько необходимо, а в конце добавляем еще один единичный бит. В результате мы добавляем к сообщению минимум 2 единичных бита. После этого этапа пароль больше не нужен для алгоритма, и он удаляется из памяти (например, замещается нулями). Данный этап представлен в листинге 3.1.

```
int    gap = 1;
int   step = 1;
int window = 2;
int   sqrt = 2;

int  row0 = 3;
int prev0 = 2;
int  row1 = 1;
int prev1 = 0;
```

Листинг 3.1 – Этап Bootstrapping

В этой первой операции выжимки цель битовой строки params в избегании коллизий, используя простые комбинации солей и паролей: например, для любых (u, v | u + v = a), мы получим коллизию если pwd = 0^u, соль = 0^v и params пустая строка. Однако, это не может произойти если params явно включает u и v. Поэтому params можно назвать дополнение соли, включающее любое количество дополнительной информации, такой как список параметров, пришедших в криптографическую хеш-функцию (включая длину соли, пароля и выхода), идентификационную строку пользователя, доменное имя в направлении, в котором пользователь аутентифицирует себя (полезно в сценариях удаленной аутентификации).

3.1.2 Setup

Когда внутреннее состояние губки проинициализировано, Lyra2 входит в фазу Setup (строки с 7 по 24 приложения А). Эта фаза включает построение матрицы памяти размерностью R*C, чьи ячейки – блоки длиной b, а R и C - параметры, определенные пользователем и b – это битрейт губки в битах.

```
    // Setup phase:
sponge.reduced_squeeze_row0(matrix, offsets[0]);

sponge.reduced_duplex_row1_and_row2(matrix, offsets[0],
offsets[1]);
sponge.reduced_duplex_row1_and_row2(matrix, offsets[1],
offsets[2]);

// Setup phase: filling loop:
for (row0 = 3; row0 != m_cost; ++row0) {
    sponge.reduced_duplex_row_filling(
```

Окончание листинга 3.1

```
            matrix,
            offsets[row1],
            offsets[prev0],
            offsets[prev1],
            offsets[row0]
    );

    prev0 = row0;
    prev1 = row1;

    row1 = (row1 + step) & (window - 1);

    if (row1 == 0) {
        window *= 2;
        step = sqrt + gap;
        gap = -gap;

        if (gap == -1) {
            sqrt *= 2;
        }
    }
}
```

Листинг 3.2 – Этап Setup

Для лучшей производительности для работы с потенциально огромной, занимающей много памяти матрицы, Setup полагается на губку с уменьшенным числом раундов, то есть операции губки выполняются с функцией f с уменьшенным количеством раундов, обозначаемой f_p, для отображения, что p раундов выполняются быстрее, чем обычное количество раундов p_{max}. Преимущества использования функции f с уменьшенным количеством раундов в том, что этот подход ускоряет операции губки и, конечно, это позволяет использовать большее количество памяти нежели функция с полным количеством раундов за одно и то же время.

3.1.3 Wandering

Это самый затратный этап по времени (строки с 27 по 37 приложения А). Она запускается после окончания этапа Setup и не сбрасывает внутренне состояние губки. Так же, как и в Setup, основа этой фазы состоит из уменьшенного дублирования строк, которые были добавлены вместе (строка 32) для вычисления похожего на случайный выход rand, который затем с помощью операции XOR складывается со строками, взятыми в качестве входа. Одно существенное отличие фазы Wandering ссылается на то, как происходит обращение к входам и выходам губки. А именно кроме взятия 4 строк вместо 3, в качестве входа губки, теперь не все строки выбираются детерминировано, но все они включают некоторого рода

псевдослучайные, зависящие от пароля переменные в процессе обращения и посещения:

- row^d (d = 0,1): рассчитанные индексы в строке 28 из первого и второго слов внешнего состояния губки, то есть из rand[0] и rand[1] для d = 0 и d = 1, соответственно. Эта деталь вычислений убеждает, что каждый индекс row^d соответствует псевдослучайному значению, находящегося в интервале [0, R - 1], что узнается только когда все столбцы до этого посещенных строк дублированы. Это дает широкий спектр возможностей, эти строки вряд ли будут в кеше, однако после того как они последовательно посещены, их столбцы могут быть предварительно выбраны процессором для ускорения их вычислений.

- $prev^d$ (d = 0,1): устанавливается в строке 36 в индексы недавно измененных строк. Так же, как и на стадии Setup, эти строки, вероятно, будут находиться в кеше. Пользуясь преимуществами этого факта, посещение его столбцов не последовательно, но на самом деле контролируется псевдослучайными переменными, зависящими от пароля, $(col_0, col_1) \in [0, C - 1]$. Более точно каждый индекс col^d (d=0,1) вычислен из выходного состояния губки. К примеру, для w = W, это берется из rand[d+2] прямо перед каждой операцией дублирования (строка 31). В результате индекс соответствующего столбца остается в памяти на время всей операции дублирования для лучшей производительности и срывая построение простых пайплайнов для их посещения.

```
// Wandering phase:
for (int i = 0; i != t_cost * m_cost; ++i) {
    row0 = (int)
Long.remainderUnsigned(mem.flip(sponge.state[0]), m_cost);
    row1 = (int)
Long.remainderUnsigned(mem.flip(sponge.state[2]), m_cost);

    sponge.reduced_duplex_row_wandering(matrix,
offsets[row0], offsets[row1], offsets[prev0],
offsets[prev1]);

    prev0 = row0;
    prev1 = row1;
}
```

Листинг 3.3 – Этап Wandering

Обращение, данное выходам губки затем довольно похоже на таковое в Setup фазе: выходы, предоставленные губкой, последовательно складываются с $M[row^0]$ посредством операции XOR (строка 33) и после поворачиваются с $M[row^1]$ (строка 34). Однако на стадии Wandering выход губки складывался с $M[row^0]$ с первого до последнего индекса, прямо как $M[row^1]$. Это архитектурное решение было использовано, потому что она позволяет ускорить вычисления, поскольку прочитанные столбцы также перезаписываются; в то же время последовательное чтение этих столбцов в

псевдослучайном порядке уже срывает стратегию злоумышленников.

3.1.4 Wrap-up

Наконец, после (R·T) операций дублирования, произведенных во время фазы Wandering, алгоритм входит в стадию Wrap-up. Эта стадия состоит из полно-раундовых операций впитывания (строка 39) одной ячейки матрицы памяти M[row0][0]. Цель этого финального вызова впитываний в основном для убеждения в том, что выжимание битовой строки ключа начнется только после выполнения полно-раундового впитывания в строке 4, состояние было обновлено только несколькими вызовами функции f с уменьшенным количеством раундов. Операция впитывания продолжается полно-раундовой операцией выжимания для генерации k бит, снова без сброса внутреннего состояния губки в нули.

```
    /**
 * Absorb words into the sponge.
 * src    a source array of words to absorb
 * len    a number of words to absorb from src
 * offset an index into src to start from
 */
public void absorb(final long[] src, final int len, final int
offset) {
    for (int i = 0; i != len; ++i) {
        state[i] ^= src[offset + i];
    }

    sponge_lyra();
}

/**
 * Squeeze bytes from the sponge.
 * dst a destination array to squeeze bytes into
 * len a number of bytes to squeeze into dst
 */
public void squeeze(byte[] dst, final int len) {
    final int div = len / BLOCK_LEN_BYTES;
    final int mod = len % BLOCK_LEN_BYTES;

    // Assume block size is a multiple of 8 bytes
    for (int i = 0; i != div; ++i) {
        final int offset0 = i * BLOCK_LEN_BYTES;

        for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
            final int offset1 = offset0 + 8 * j;

            byte[] bytes = pack.bytes(state[j]);

            for (int k = 0; k != 8; ++k) {
                dst[offset1 + k] = bytes[k];
```

Окончание листинга 3.4

```
            }
        }

        sponge_lyra();
    }
```

Листинг 3.4 – Этап Wrap-Up

В результате эта последняя стадия выполняет лишь обычные операции лежащей в основе губки, построенной на своей безопасности для убеждения в том, что весь процесс не обратим и выход непредсказуем. После всего нарушение таких основных свойств Lyra2 равноценно нарушению общих основных свойств нижележащей полно-раундовой губки.

3.3 Настройка потребления памяти и времени вычислений

Общее количество памяти, занимаемое матрицей памяти Lyra2 составляет b·R·C битов, где b зависит от битрейта лежащей в основе sponge-функции. С таким выбором b нет необходимости в дополнении входящих блоков по мере того, как они обрабатываются дуплексной конструкцией, которая служит для упрощения и потенциального ускорения реализации. Параметры R и C, с другой стороны, могут быть определены пользователем, тем самым позволяя настраивать необходимое количество памяти во время выполнения алгоритма.

Игнорируя вспомогательные операции, вычислительная стоимость Lyra2 обычно определена количеством вызовов лежащей в основе губки функции f. Аппроксимированная стоимость этого составляет [(|pwd| + |salt| + |params|)/b] вызовов на стадии Bootstrapping, плюс $R \cdot C \cdot p/p_{max}$ в фазе Setup, плюс $T \cdot R \cdot C \cdot p/p_{max}$ на стадии Wandering, плюс [k/b] на этапе Wrap-Up, ведущее примерно к $(T+1) \cdot R \cdot C \cdot p/p_{max}$ вызовов функции f для маленьких длин pwd, salt и k. Поэтому пока количество используемой памяти алгоритмом налагает нижний предел на общее время работы, в последствие оно может быть увеличено без влияния предшественника выбором подходящего параметра T. Это позволяет пользователям исследовать наиболее изобилующий ресурс легитимной платформы с несбалансированной возможностью памяти и вычислительной мощности. Эта архитектура также позволяет Lyra2 использовать больше памяти чем scrypt за одно и то же вычислительное время: пока scrypt вызывает полно-раундовый хеш для вычисления каждого его элемента, Lyra2 вызывает уменьшенное количество раундов, ускоряя операции одной и той же задачи.

4 Программная реализация

Для написания приложения, реализующего хеш-функцию Lyra2, я использовал язык программирования Java, а для создания интерфейса я использовал платформу JavaFX. Эта платформа основана на Java и позволяет создавать как настольные приложения, так и интернет-приложения с графическим интерфейсом. Для разметки интерфейса в JavaFX используется декларативный язык разметки JavaFX Script.

Чтобы можно было работать с этой платформой необходимо скачать JavaFX SDK с официального сайта. Также для удобства работы с языком разметки можно дополнительно установить SceneBuilder, которое представляет из себя конструктор приложения, где можно настраивать размеры и месторасположение множества элементов управлений, доступных для JavaFX Script.

В результате я получил интерфейс, представленный на рисунке 4.1.

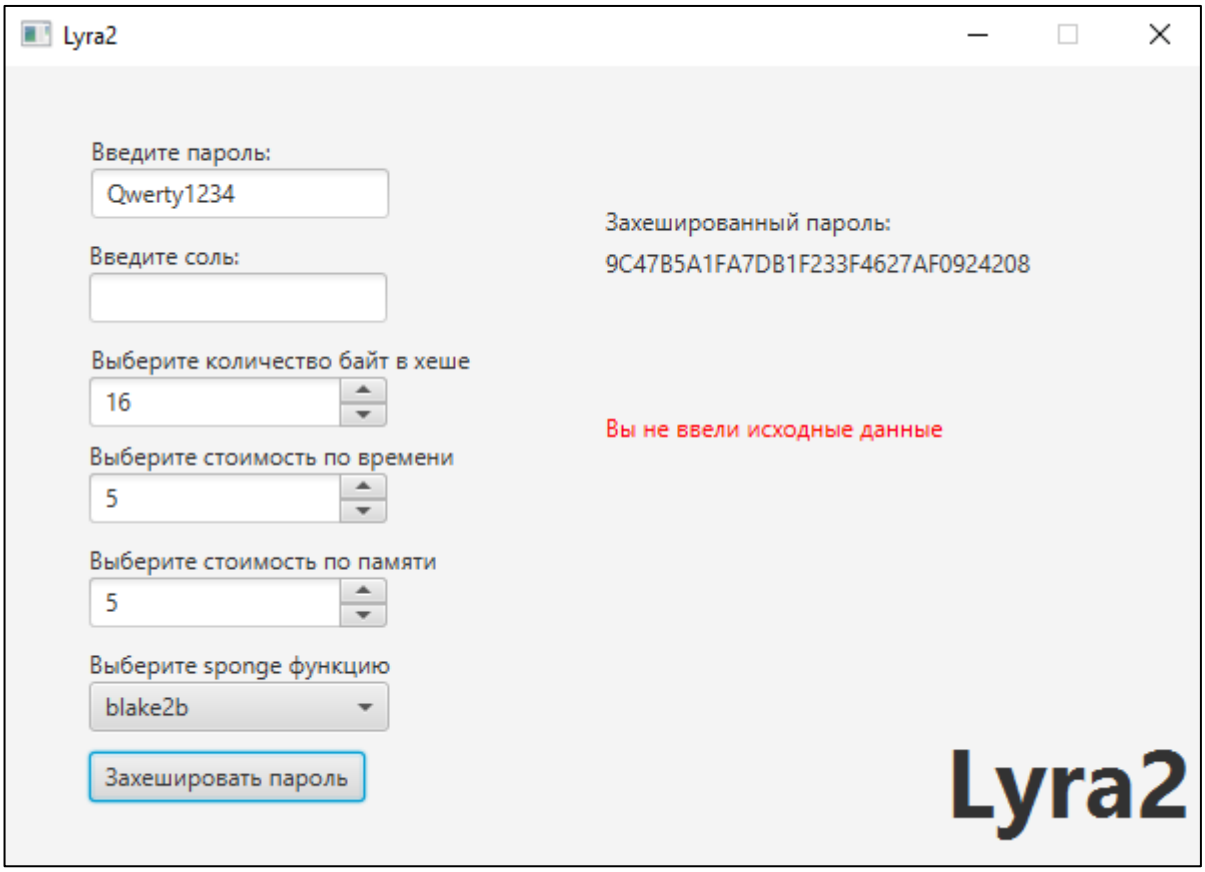

Рисунок 3.1 – Графический интерфейс приложения

Данное приложение позволяют пользователю задавать свои пароль и соль, задавать затраты по времени и памяти, а также sponge-функцию, а на выходе получить захешированный пароль с заданным количество байт.

5 Безопасность

Архитектура Lyra2 такая, что полученный ключ устойчив к коллизиям и односторонен, что достигается начальными и конечными полными операциями хеширования, в комбинации с операциями с уменьшенным числом раундов в середине алгоритма; злоумышленники не смог распараллелить алгоритм, используя несколько экземпляров криптографической губки H, поэтому они не могут существенно ускорить процесс тестирования пароля средствами многоядерного процессора; однажды инициализировавшись, матрица памяти остается в памяти на время большинства процессов хеширования пароля, означая что оптимальная операция Lyra2 требует достаточно (быстрой) памяти, чтобы хранить ее содержимое.

Для лучшей производительности легитимный пользователь хранит целую матрицу во временной памяти, способствуя доступа к ней в каждой итерации алгоритма. Злоумышленник, запускающий множественные экземпляры Lyra2, с другой стороны, может решить не делать то же самое, а держать в памяти небольшую часть матрицы в быстрой памяти, с целью уменьшения затрат памяти за каждый пароль. Даже несмотря на это альтернативный подход неизбежное уменьшение пропускной способности каждого конкретного экземпляра Lyra2, цель этой стратегии позволить выполнять больше независимых тестов паролей одновременно, что потенциально увеличивает среднюю пропускную способность. Вот два основных метода для достижения этого. Первый так называемая атака с нехваткой памяти, которая состоит в размене памяти на вычислительное время, то есть удаление части матрицы и пересчете удаленной информации с нуля, когда (и только когда) это становится необходимым. Второй метод состоит в использовании малозатратных (а, следовательно, более медленной) запоминающих устройств, таких как магнитный жесткий диск и называется атака с медленной памятью.

Также стоит упомянуть так называемую кеш-атаку по времени, которая вызывает шпионский процесс, совмещенный с функцией получения ключа и, наблюдая за последним выполнением, получит возможность восстановить пользовательский пароль без необходимости выполнения утомительного поиска.

5.1 Атака с нехваткой памяти

Для того чтобы оценить устойчивость Lyra2 к атакам с нехваткой памяти, нужно понять, как такие атаки могут быть совершены против ROMix структуры scrypt. Причина в том, что его последовательная жесткая архитектура памяти в основном служит для обеспечения защиты от конкретно этого вида угроз.

Чтобы это рассмотреть авторы Lyra2 сформулировали следующую теорему:

Пока память и вычислительные затраты scrypt имеют сложность O(R) для системного параметра R, один из них может достичь затрат памяти O(1) (например, в атаке свободной памятью), но увеличивая затраты на вычисления до $O(R^2)$.

Они также приводят доказательство этой теоремы, которое я приводить здесь не буду.

Для сравнения, злоумышленник пытающийся использовать такую же атаку с нехваткой памяти против Lyra2 будет подвергнуть дополнительным испытаниям. Во-первых, во время фазы Setup недостаточно просто хранить одну строку в памяти для вычисления следующей, каждая строка требует хранения трех предыдущих строк для своего вычисления.

В любом случае это создает сложную сеть зависимостей, которая растет в размере, пока выполнение алгоритма движется вперед и все большее количество строк подвергается изменению, ведущее к нескольким рекурсивным вызовам. Этот эффект даже более впечатлителен на стадии Wandering, в связи с дополнительными усложняющими факторами: каждая операция дублирования влияет на случайный (зависимый от пароля_ индекс строки, который не может быть определен до конца предыдущего дублирования. Поэтому выбор, какие строки хранить в памяти и какие строки удалить просто спекулятивный и не может быть просто оптимизирован для всех тестируемых паролей.

5.2 Атака с медленной памятью

По сравнению с атакой с нехваткой памяти, предоставление защиты против атаки с медленной памятью это более сложная задача. Это происходит из-за того, что злоумышленники действуют почти как легитимные пользователи во время операций алгоритма, сохраняя в памяти все необходимую информацию. Главным отличием является полоса пропускания и задержка, предлагаемая используемым запоминающим устройством, которое в конечном счете влияет на время, требуемое для тестирования каждого тестируемого пароля.

Lyra2 равно как и scrypt исследует свойства устройств дешевой памяти посещением мест памяти, следуя псевдослучайному шаблону во время стадии Wandering. В частности, эта стратегия увеличивает задержку последовательных запоминающих устройств, таких как жесткий диск, особенно если злоумышленник использует множественные экземпляры, одновременно обеспечивая доступ к разным секциям памяти. Более того, эти псевдослучайные шаблоны в комбинации с маленьким параметром C может также уменьшить ускорение, требуемое механизмами, таким как кеширование и предварительная выборка, даже когда злоумышленники используют дешевые чипы памяти RAM. Даже так задержка может быть

(частично) спрятана в параллельной атаке предварительной выборкой строк, необходимых для одного потока пока остальные потоки запущены, как минимум злоумышленник должен заплатить тратами за часто изменяемый контекст каждого потока. Этот подход частично губителен против старых моделей видеокарт, чья внутренняя структура была обычно оптимизирована против детерминированного доступа к маленьким порциям данных.

В сравнении с scrypt, небольшое количество улучшений внедренных в Lyra2 против таких атак состоит в том, что ячейки памяти не только повторяюще считываются, но и записываются. В результате Lyra2 требует повторяющегося перемещения данных вверх и вниз по иерархии памяти. Общее влияние этих особенностей на производительность атак с медленной памятью зависит, однако, от точной архитектуры системы. К примеру, это как увеличить трафик на общей шине памяти, пока механизмы кеширования могут потребовать более сложную схему/планирование, чтобы справится с продолжительным потоком информации от/к медленному уровню памяти. Это использование большой пропускной способности также препятствует конструкции высокопроизводительного разделенного аппаратного обеспечения для тестирования множества паролей параллельно.

Другая особенность Lyra2 тот факт, что во время фазы Wandering столбцы недавно обновленных строк ($M[prev^0]$ и $M[prev^1]$) читаются в псевдослучайном стиле. Так как ожидается, что эти строки будут в кеше во время обычного выполнения Lyra2, легитимный пользователь, который адекватно настраивает C должен суметь прочитать эти строки примерно с такой скоростью, как если бы они считывались последовательно. Злоумышленник, использующий платформу с меньшим размером кеша, однако, должен получить более низкую производительность, в связи с потерей кеша. В дополнение этот псевдослучайный шаблон затрудняет создание простых пайплайнов в аппаратном обеспечении для посещения этих строк: даже если злоумышленник хранит все столбцы в быстрой памяти для избежания проблем с задержкой, некоторые функции выборки будут необходимы, чтобы выбирать среди этих столбцов на лету.

Наконец, в архитектуре Lyra2 выход губки всегда проходит операцию XOR со значением существующих строк, предотвращая ячейки памяти, относящиеся к этим строкам, от становления быстро заменяемыми. Это свойство, кстати, затрудняет емкость повторно используемых областей памяти злоумышленников в параллельном потоке.

Очевидно, все особенности показанные Lyra2 для предоставления защиты против атак с медленной памятью может также повлиять на производительность алгоритма для легитимных пользователей. В конце концов, они также мешают законным возможностям платформы использовать свои собственные методы кеширования и предварительной выборки. Поэтому крайне важно, чтобы настройка алгоритма была оптимизирована под характеристики платформы, учитывая такие аспекты как количество свободной RAM, длина строки кеша и так далее. Это должно

позволить выполнению Lyra2 запускаться более плавно на компьютерах легитимных пользователей, и давать более внушительные штрафы злоумышленникам, использующих платформы с определенными характеристиками.

5.3 Кеш-атака

Кеш-атака – это тип атаки по сторонним каналам, в которой злоумышленник может наблюдать за поведением тайминга устройства, отслеживая его доступ к кеш-памяти. Этот класс атак был представлен, чтобы быть эффективным, к примеру против некоторых реализаций AES и RSA, позволяя восстанавливать секретный ключ, используемый в алгоритме.

В контексте хеширования паролей эта атака может быть угрозой против hard-memory решений, которые используют операции, для которых порядок посещения зависит от пароля. Причина в том, что как минимум в теории, шпионские процессы, которые наблюдают за поведением кеша правильного пароля, могут отфильтровать пароли, которые не совпадают с этим шаблоном после нескольких итераций, скорее, чем продолжительность всего алгоритма. Тем не менее кеш-атаки вряд ли могут быть причиной значительного беспокойства в сценариях, когда функция получения ключа запускает одиночный сценарий пользователя, такой как локальная аутентификация или удаленная аутентификация, произведенная на разделенном сервере: после всего, если злоумышленник способен вставить такой процесс в эти окружения, то вполне вероятно, что он может вставить и более сильное шпионское обеспечение, чтобы получить пароль более прямым способом.

С другой стороны, эти атака могут быть интересным подходом в сценариях, когда запущенное на физическом аппаратном средстве функция формирования ключа разделена на процессы разным пользователям, такие как виртуальные серверы, размещенные в публичном облаке. Это происходит, потому что такие окружения потенциально создают требуемые условия для измерения кеш-таймингов, но ожидается не допущение установки вредоносного ПО достаточно сильного, чтобы обойти способность изоляции гипервизора для доступа к данным из других виртуальных машин.

В этом контексте подход, использованный в Lyra2 для предоставления защиты против кеш-атак только во время стадии Setup, в которой индексы строк считываются и записываются независимо от пароля, когда стадии Wandering и Wrap-Up восприимчивы к таким атакам. В результате, даже хотя Lyra2 не имеет полного иммунитета к кеш-атакам, алгоритм гарантирует, что злоумышленники будут запускать целую фазу Setup и как минимум часть фазы Wandering прежде, чем они смогут использовать информацию о расчете времени кеша для фильтрации паролей. Поэтому такие атаки все равно влияют на использование памяти как минимум R/2 строк.

Обоснование такого архитектурного решения по предоставлению

частичной защиты от кеш атак тройное. Первое обоснование, создание посещения памяти зависимым от пароля – это один из главных способов защиты Lyra2 против атак с медленной памятью, так как это препятствует кешированию и механизмов предварительной выборки, что может ускорить эту угрозу. Поэтому защита против атак с нехваткой памяти и защита против кеш-атак по времени в некотором роде конфликтуют требованиями. Так как эти атаки подходят для широкого набора сценариев, начиная от локальной и удаленной аутентификации, это кажется более важным для защиты против них чем полное предотвращение кеш-атак.

Второе, для практических причин (а именно масштабируемость) это может быть интересным для разгрузки процесса хеширования паролей для пользователей, распределяя лежащую в основе стоимость среди устройств клиентов, скорее, чем концентрация их на сервере, даже в случае удаленной аутентификации. Эта главная идея позади протокола освобождения сервера, в соответствии с которой сервер отправляет только соль клиенту (предпочтительно используя безопасный канал), кто отвечает с x = PHS(pwd, salt); затем сервер вычисляет только локально y = H(x) и сравнивается его со значением, хранимым в его собственной базе данных. Результатом этого подхода является то, что серверная сторона вычислений во время аутентификации сводится к вычислению одного хеша, пока операции с интенсивным использованием памяти и вычислений, связанные с процессом хешированием пароля, выполняется клиентом, в окружении в котором кеш-тайминг вероятно станет менее критической заботой.

Третье, последние достижения технологий программного и аппаратного обеспечения могут (частично) воспрепятствовать осуществимости кеш-тайминга и связанных с ним атак из-за количества «шума» переданных их скрытой сложности. Эти технологические ограничения также усиляются фактом того, что занимающиеся безопасностью облачные провайдеры ожидают усиления контрмер против такого рода атак, защищая своих пользователей с помощью: гарантия того, что процесс запуска разными пользователями не влияет кеш друг друга (или, как минимум, что это влияние не будет полностью предсказуемым); или делая его более сложным для злоумышленников, чтобы размещать шпионский процесс на той же физической машине, где и процессы чувствительные к безопасности, в особенности относящегося к аутентификации пользователя. Поэтому, даже эти контрмеры, добавляющие сложности, принесенной ими может быть достаточным, чтобы заставить злоумышленников запускать большие части фазы Wandering, платя соответствующую стоимость, пока угадывание пароля будет надежно стерто.

Заключение

В результате анализа алгоритма хеширования Lyra2 можно заключить, что данная функция формирования ключа является довольно гибкой, а что более важно – безопасной. Lyra2 отлично подходит для использования в процессе хеширования и обеспечивает хорошую устойчивость от самых разных атак, которые обычно применяются против алгоритмов, использующих sponge-функции.

Данную криптографическую функцию можно модернизировать и подстраивать под совершенно разные задачи. Есть расширенный алгоритм Lyra2p, позволяющий выполнять алгоритм параллельно.

Отдельного упоминания заслуживает Lyra2REv2. Данный алгоритм используется в таких криптовалютах Vertcoin, MonaCoin, что в лишний раз доказывает хорошую продуманность, безопасность и эффективность данной функции получения ключа.

На независимом открытом конкурсе Password Hashing Competition, который, к слову, выиграл Argon2, Lyra2 и еще 3 алгоритма были удостоены специального признания.

Таким образом, можно сказать с уверенностью, что Lyra2 заняла свое достойное в месте ряду других функций формирования ключа таких как Argon2, bcrypt, PBKDF2, scrypt.

Список использованных источников

1. Lyra2: Efficient Password Hashing with High Security against Time-Memory Trade Offs [Электронный ресурс]. – Режим доступа: http://eprint.iacr.org/2015/136. – Дата доступа: 02.04.2022.

2. TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme [Электронный ресурс]. – Режим доступа: TwoCats-v0.pdf (password-hashing.net). – Дата доступа: 05.05.2022.

Приложение А

Листинг общей реализации губок в классе Sponge

```
     /**
 * Представляет губку и реализует общие методы
 */
public abstract class Sponge {
    static final long[] blake2b_IV = {
            0x6a09e667f3bcc908L, 0xbb67ae8584caa73bL,
            0x3c6ef372fe94f82bL, 0xa54ff53a5f1d36f1L,
            0x510e527fade682d1L, 0x9b05688c2b3e6c1fL,
            0x1f83d9abfb41bd6bL, 0x5be0cd19137e2179L
    };
    public long[] state;
    public final int N_COLS;
    public final int FULL_ROUNDS;
    public final int HALF_ROUNDS;
    public final int BLOCK_LEN_INT64;
    public final int BLOCK_LEN_BYTES;
    /**
     * Строит губку и инициализирует ее состояние
   * Инициализирует первую половину состояния нулями, а вторую с
     * помощью blake2b_IV. Это подразумевает, что состояние
имеет 16 * 8 = 128 байтов
     */
    public Sponge(LyraParams params) {
        // initialize the sponge state:
        state = new long[16];
        // first 8 words are zeroed out
        for (int i = 0; i != 8; ++i) {
            state[i] = 0;
        }
        // second 8 words are blake2b_IV's
        for (int i = 0; i != 8; ++i) {
            state[8 + i] = mem.flip(blake2b_IV[i]);
        }
        this.N_COLS = params.N_COLS;
        this.FULL_ROUNDS = params.FULL_ROUNDS;
        this.HALF_ROUNDS = params.HALF_ROUNDS;
        this.BLOCK_LEN_INT64 = params.BLOCK_LEN_INT64;
        this.BLOCK_LEN_BYTES = params.BLOCK_LEN_BYTES;
    }
    /**
     * Поглощает слова в губку
     * src    исходный массив слов для поглощения
     * len    количество слов для поглощения из src
     * offset индекс элемента из src, с которого начинать
     */
    public void absorb(final long[] src, final int len, final
```

```
int offset) {
        for (int i = 0; i != len; ++i) {
            state[i] ^= src[offset + i];
        }
        sponge_lyra();
    }
    /**
     * Выжимает байты из губки
     * dst целевой массив для выжимки
     * len количество байт для выжимания в dst
     */
    public void squeeze(byte[] dst, final int len) {
        final int div = len / BLOCK_LEN_BYTES;
        final int mod = len % BLOCK_LEN_BYTES;
        // Assume block size is a multiple of 8 bytes
        for (int i = 0; i != div; ++i) {
            final int offset0 = i * BLOCK_LEN_BYTES;

            for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
                final int offset1 = offset0 + 8 * j;
                byte[] bytes = pack.bytes(state[j]);
                for (int k = 0; k != 8; ++k) {
                    dst[offset1 + k] = bytes[k];
                }
            }
            sponge_lyra();
        }
        final int div8 = mod / 8;
        final int mod8 = mod % 8;
        final int offset0 = div * BLOCK_LEN_BYTES;
        for (int i = 0; i != div8; ++i) {
            final int offset1 = offset0 + 8 * i;
            final byte[] bytes = pack.bytes(state[i]);
            for (int j = 0; j != 8; ++j) {
                dst[offset1 + j] = bytes[j];
            }
        }
        final int offset1 = offset0 + 8 * div8;
        for (int i = 0; i != mod8; ++i) {
            final byte[] bytes = pack.bytes(state[div8]);
            for (int j = 0; j != mod8; ++j) {
                dst[offset1 + j] = bytes[j];
            }
        }
    }
    /**
     * Свдиг слов на определенное количество битов вправо
     * Rotate a word by several bits to the right.
     * word слово для сдвига
     * b    величина сдвига
     * a    результат сдвига
     */
```

```
    public static long rotr64(final long word, final int b) {
        return (word << (64 - b)) | (word >>> b);
    }
    /**
     * Сдвиг слов на определенное количество битов влево
     * word слово для свдига
     * b    величина сдвига
     * a    результат сдвига
     */
    public static long rotl64(final long word, final int b) {
        return (word << b) | (word >>> (64 - b));
    }
    public abstract void G(final int a, final int b, final int
c, final int d);
    /**
     * Обноавляет состояние губки
     * rounds количество перестановок
     */
    public void sponge_lyra(final int rounds) {
        for (int round = 0; round != rounds; ++round) {
            G(0, 4,  8, 12);
            G(1, 5,  9, 13);
            G(2, 6, 10, 14);
            G(3, 7, 11, 15);
            G(0, 5, 10, 15);
            G(1, 6, 11, 12);
            G(2, 7,  8, 13);
            G(3, 4,  9, 14);
        }
    }
    /**
     * Обновляет состояние губки, запускает полное количество
раундов по умолчанию.
     */
    public void sponge_lyra() {
        sponge_lyra(FULL_ROUNDS);
    }
    public void reduced_squeeze_row0(long[] dst, final int
offset) {
        int word = (N_COLS - 1) * BLOCK_LEN_INT64;
        for (int i = 0; i != N_COLS; ++i) {
            System.arraycopy(state, 0, dst, offset + word,
BLOCK_LEN_INT64);
            word -= BLOCK_LEN_INT64;
            sponge_lyra(HALF_ROUNDS);
        }
    }
    public void reduced_duplex_row1_and_row2(long[] dst, final
int offset1, final int offset2) {
        int word1 = 0;
        int word2 = (N_COLS - 1) * BLOCK_LEN_INT64;
        for (int i = 0; i != N_COLS; ++i) {
```

```
            for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
                state[j] ^= dst[offset1 + word1 + j];
               }

            sponge_lyra(HALF_ROUNDS);
            for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
                dst[offset2 + word2 + j] = dst[offset1 + word1 +
j] ^ state[j];
            }
            word1 += BLOCK_LEN_INT64;
            word2 -= BLOCK_LEN_INT64;
        }
    }
    /**
     * Операция дублирования
     * Все смещения указывают на матрицу dst и обозначают начало
некоторых строк байт
     * All of the offsets point into {@code dst} and denote a
start of some *row* of bytes.
     * dst     матрица, которая предоставляет и получает байты
     * offset0 строка, которая предоставляет и получает байты
     * offset1 строка, которая предоставляет байты (последняя
проинициализированная строка)
     * offset2 строка, которая предоставляет байты (последняя
повторно посещенная и обновленная строка)
     * offset3 строка,которая получает байты
     */
    public void reduced_duplex_row_filling(long dst[], final int
offset0, final int offset1, final int offset2, final int
offset3) {
        int word0 = offset0;
        int word1 = offset1;
        int word2 = offset2;
        int word3 = offset3 + (N_COLS - 1) * BLOCK_LEN_INT64;
        for (int i = 0; i != N_COLS; ++i) {
            for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
                state[j] ^= mem.flip(
                      mem.flip(dst[word0 + j])
                    + mem.flip(dst[word1 + j])
                    + mem.flip(dst[word2 + j])
                );
            }
            sponge_lyra(HALF_ROUNDS);
            for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
                dst[word3 + j] = dst[word1 + j] ^ state[j];
            }
            for (int j = 0; j != BLOCK_LEN_INT64 - 2; ++j) {
                dst[word0 + j] ^= state[j + 2];             }
            dst[word0 + BLOCK_LEN_INT64 - 2] ^= state[0];
            dst[word0 + BLOCK_LEN_INT64 - 1] ^= state[1];
            word0 += BLOCK_LEN_INT64;
            word1 += BLOCK_LEN_INT64;
```

```
            word2 += BLOCK_LEN_INT64;
            word3 -= BLOCK_LEN_INT64;
        }
    }
    /**
     * Операция дублирования
     * Все смещения указывают на dst и обозначают начало
некоторой строки байт
     * dst     матрица, котоаря предоставляет и получает байты
     * offset0 строка, которая предоставляет  и получает байты
     * offset1 строка, котоаря предоставляет и получает байты
после сдвига
     * offset2 строка, которая предоставляет байты
     * offset3 строка, которая предоставляет байты
     */
    public void reduced_duplex_row_wandering(long[] dst, final
int offset0, final int offset1, final int offset2, final int
offset3) {
        int word0 = offset0;
        int word1 = offset1;
        for (int i = 0; i != N_COLS; ++i) {
            final int rndcol0 = (int)
Long.remainderUnsigned(mem.flip(state[4]), N_COLS) *
BLOCK_LEN_INT64;
            final int rndcol1 = (int)
Long.remainderUnsigned(mem.flip(state[6]), N_COLS) *
BLOCK_LEN_INT64;
            final int word2 = offset2 + rndcol0;
            final int word3 = offset3 + rndcol1;
            for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
                state[j] ^= mem.flip(
                      mem.flip(dst[word0 + j])
                    + mem.flip(dst[word1 + j])
                    + mem.flip(dst[word2 + j])
                    + mem.flip(dst[word3 + j])
                );
            }
            sponge_lyra(HALF_ROUNDS);
            for (int j = 0; j != BLOCK_LEN_INT64; ++j) {
                dst[word0 + j] ^= state[j];
            }
            for (int j = 0; j != BLOCK_LEN_INT64 - 2; ++j) {
                dst[word1 + j] ^= state[j + 2];
            }
            dst[word1 + BLOCK_LEN_INT64 - 2] ^= state[0];
            dst[word1 + BLOCK_LEN_INT64 - 1] ^= state[1];
            word0 += BLOCK_LEN_INT64;
            word1 += BLOCK_LEN_INT64;
        }
    }
}
```

Приложение Б

Algorithm 2 The Lyra2 Algorithm.

PARAM: H ▷ Sponge with block size b (in bits) and underlying permutation f
PARAM: H_ρ ▷ Reduced-round sponge for use in the Setup and Wandering phases
PARAM: ω ▷ Number of bits to be used in rotations (recommended: a multiple of W)
INPUT: pwd ▷ The password
INPUT: $salt$ ▷ A salt
INPUT: T ▷ Time cost, in number of iterations ($T \geqslant 1$)
INPUT: R ▷ Number of rows in the memory matrix
INPUT: C ▷ Number of columns in the memory matrix (recommended: $C \cdot \rho \geqslant \rho_{max}$)
INPUT: k ▷ The desired hashing output length, in bits
OUTPUT: K ▷ The password-derived k-long hash

▷ **Bootstrapping phase**: Initializes the sponge's state and local variables
▷ Byte representation of input parameters (others can be added)
$params \leftarrow len(k) \parallel len(pwd) \parallel len(salt) \parallel T \parallel R \parallel C$
$H.absorb(\texttt{pad}(pwd \parallel salt \parallel params))$ ▷ Padding rule: 10^*1.
$gap \leftarrow 1$; $stp \leftarrow 1$; $wnd \leftarrow 2$; $sqrt \leftarrow 2$ ▷ Initializes visitation step and window
$prev^0 \leftarrow 2$; $row^1 \leftarrow 1$; $prev^1 \leftarrow 0$

▷ **Setup phase**: Initializes a $(R \times C)$ memory matrix, it's cells having b bits each
for $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M[0][C-1-col] \leftarrow H_\rho.squeeze(b)\}$ **end for**
for $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M[1][C-1-col] \leftarrow M[0][col] \oplus H_\rho.duplex(M[0][col], b)\}$ **end for**
for $(col \leftarrow 0$ **to** $C-1)$ **do** $\{M[2][C-1-col] \leftarrow M[1][col] \oplus H_\rho.duplex(M[1][col], b)\}$ **end for**
for $(row^0 \leftarrow 3$ **to** $R-1)$ **do** ▷ **Filling Loop**: initializes remainder rows
    ▷ **Columns Loop**: $M[row^0]$ is initialized; $M[row^1]$ is updated
    for $(col \leftarrow 0$ **to** $C-1)$ **do**
        $rand \leftarrow H_\rho.duplex(M[row^1][col] \boxplus M[prev^0][col] \boxplus M[prev^1][col], b)$
        $M[row^0][C-1-col] \leftarrow M[prev^0][col] \oplus rand$
        $M[row^1][col] \leftarrow M[row^1][col] \oplus \text{rot}(rand)$ ▷ rot(): right rotation by ω bits
    end for
    $prev^0 \leftarrow row^0$; $prev^1 \leftarrow row^1$; $row^1 \leftarrow (row^1 + stp) \bmod wnd$
    if $(row^1 = 0)$ **then** ▷ Window fully revisited
        ▷ Doubles window and adjusts step
        $wnd \leftarrow 2 \cdot wnd$; $stp \leftarrow sqrt + gap$; $gap \leftarrow -gap$
        if $(gap = -1)$ **then** $\{sqrt \leftarrow 2 \cdot sqrt\}$ **end if** ▷ Doubles $sqrt$ every other iteration
    end if
end for

▷ **Wandering phase**: Iteratively overwrites pseudorandom cells of the memory matrix
▷ **Visitation Loop**: $2R \cdot T$ rows revisited in pseudorandom fashion
for $(wCount \leftarrow 0$ **to** $R \cdot T - 1)$ **do**
    $row^0 \leftarrow \text{lsw}(rand) \bmod R$; $row^1 \leftarrow \text{lsw}(\text{rot}(rand)) \bmod R$ ▷ Picks pseudorandom rows
    for $(col \leftarrow 0$ **to** $C-1)$ **do** ▷ **Columns Loop**: updates $M[row^{0,1}]$
        ▷ Picks pseudorandom columns
        $col^0 \leftarrow \text{lsw}(\text{rot}^2(rand)) \bmod C$; $col^1 \leftarrow \text{lsw}(\text{rot}^3(rand)) \bmod C$
        $rand \leftarrow H_\rho.duplex(M[row^0][col] \boxplus M[row^1][col] \boxplus M[prev^0][col^0] \boxplus M[prev^1][col^1], b)$
        $M[row^0][col] \leftarrow M[row^0][col] \oplus rand$ ▷ Updates first pseudorandom row
        $M[row^1][col] \leftarrow M[row^1][col] \oplus \text{rot}(rand)$ ▷ Updates second pseudorandom row
    end for ▷ End of Columns Loop
    $prev^0 \leftarrow row^0$; $prev^1 \leftarrow row^1$ ▷ Next iteration revisits most recently updated rows
end for ▷ End of Visitation Loop

▷ **Wrap-up phase**: output computation
$H.absorb(M[row^0][0])$ ▷ Absorbs a final column with full-round sponge
$K \leftarrow H.squeeze(k)$ ▷ Squeezes k bits with full-round sponge
return K ▷ Provides k-long bitstring as output