КОМП’ЮТЕРНИЙ ПРАКТИКУМ No2

з курсу

ГЕШ-ФУНКЦІЇ ТА КОДИ АВТЕНТИЧНОСТІ

Реалізація атаки Хеллмана на геш-функції

Приходько Юрій ФБ-12, Варіант 12.

Мета роботи

Опанувати методи оптимізованого перебору для побудови атак на геш-функції, експериментально визначити параметри методу Хеллмана.

Хід роботи

0. Аналіз завдання

За варіантом була використана функція Blake2b.
Розглянемо основні моменти що були визначені при аналізі завдання та його виконанні.

0.1 Функція редукції

Для функції редукції було використане рекомендоване перетворення, вигляду $R(x) = r||x$, де x - довільне вхідне значення, а r випадково згенерований вектор довжини $128 - 32 = 96$ бітів. r генерується один на таблицю, і зберігається разом з нею для використання при атаці.

0.2 Простір вихідних значень

Геш-функція була обмежена в розмірі за допомогою функції $truncate$ що повертає останніх $n = 32$ бітів. Функція $truncate$ обмежує простір вихідних значень геш-функції нерівномірно, тобто у нас можуть виникнути ситуації в яких певні значення геш-функції в принципі не будуть існувати на наборі вхідних повідомлень.
Це потрібно буде враховувати при виконанні безпосередньо атаки, за методичними вказівками до виконання лабораторної роботи рекомендовано генерувати повідомлення довжини 256 біт, гешувати їх і після цього намагатись знайти прообраз за допомогою передобчислених таблиць. Випадкове значення довжини 256 біт, не знаходитиметься в просторі повідомлень сформованих функцією редукції, а отже без виникнення

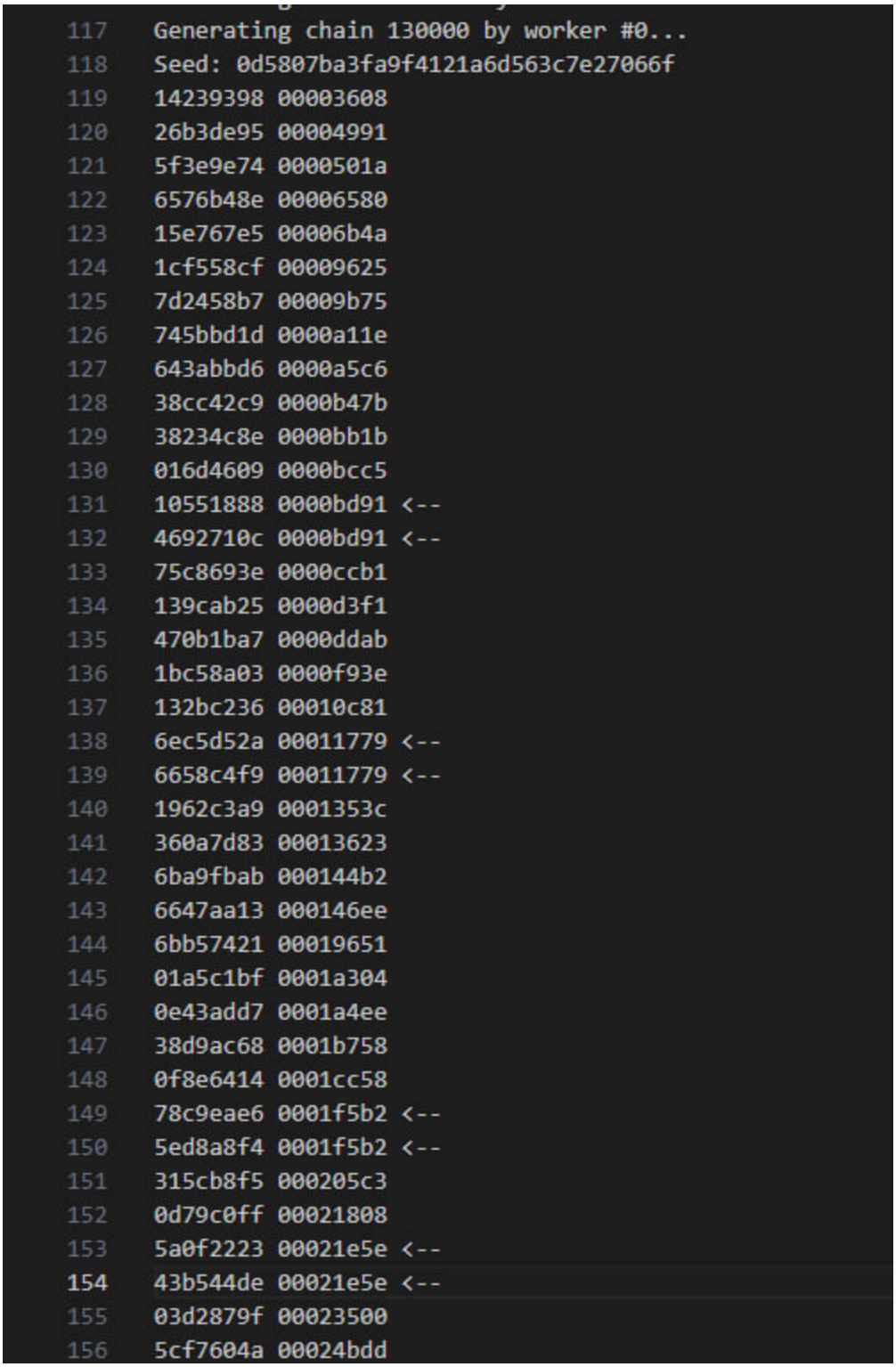
безпосередньої випадкової колізії на повідомленні ми не матимемо змоги знайти прообраз.

0.3 Простір вхідних значень

Простір вхідних значень генерується як 32 бітові числа, отримані або при випадковій генерації на початку ланцюга, або як результат геш функції, розширені до 128 біт за допомогою статичного випадкового префіксу функцією редукції, що ніяк не впливає на обмежений розмір простору 2^{32}. Кількість унікальних значень повідомлень в таблиці залишається та сама.

0.4 Зклейки в ланцюгах таблиці

Так як простір вхідних значень обмежений 2^{32} у нас доволі часто будуть траплятись колізії та цикли які заважатимуть адекватній роботі алгоритму. Приклад з емпіричного досліду наведено на мал. 1.



Мал. 1, бачимо 4 колізії за перші 50 значень в згенерованій таблиці

Також колізія може відбутись значенням в ланцюгу таблиці, та ланцюгу що отримується під час атаки безпосередньо з атакованого гешу. Наприклад:
Ми обрахували певну кількість гешувань вхідного гешу шукаючи чи співпаде він на поточній ітерації з якимось значенням з другої колонки таблиці, і ми дійсно знаходимо таке значення. Але прообраз отримуємо неправильний, тобто отриманий x такий що $h(x) \neq y$, де y геш для якого ми шукали прообраз. Це стається у випадку колізії в ряді гешувань y та

ланцюгу таблиці. На практиці ми можемо зробити бектрейс обох ланцюгів і побачити місце де все "пішло не за планом".

```
chain: 6076f686 | hash: 6076f686
chain: a53b8760 | hash: a53b8760
chain: f0960915 | hash: f0960915
chain: dbb46cbc | hash: dbb46cbc
chain: 93403060 | hash: 93403060
chain: c6679e42 | hash: c6679e42
chain: 30707b4d | hash: 30707b4d
chain: 48b601f4 | hash: 48b601f4
chain: 99486aad | hash: 99486aad
chain: 17327488 | hash: 17327488
chain: 03ec391a | hash: 03ec391a
chain: d099c1ca | hash: d099c1ca
chain: 008a88ed | hash: 008a88ed
chain: 6a9fe040 | hash: 6a9fe040
chain: caee0b47 | hash: caee0b47
chain: 3a049ae9 | hash: 3a049ae9
chain: 05fe4075 | hash: 05fe4075
chain: 3870bd48 | hash: 3870bd48
chain: 29e20246 | hash: 29e20246
chain: 30a3c530 | hash: 30a3c530
chain: b39dc459 | hash: b39dc459
chain: 6ec002f9 | hash: 6ec002f9
chain: ad81a249 | hash: ad81a249
chain: 5d33a317 | hash: 5d33a317
chain: 8b132474 | hash: 8b132474
chain: 3b888911 | hash: 3b888911
chain: 6569db36 | hash: 6569db36
chain: 879d40b6 | hash: 879d40b6
chain: 989592d7 | hash: 989592d7
chain: e0968fba | hash: e0968fba
chain: 81d388e7 | hash: 81d388e7
chain: 37c959fd | hash: 37c959fd
chain: 53e48015 | hash: 53e48015
chain: 69fcddee | hash: 69fcddee
chain: 17e97d0a | hash: 17e97d0a
chain: 0a36e6e3 | hash: 0a36e6e3
chain: a23b813d | hash: a23b813d
```

```
chain: 23deb6ef | hash: 23deb6ef
chain: 607c0309 | hash: 607c0309
chain: 8e36d278 | hash: 8e36d278
chain: 317f4a19 | hash: 317f4a19
chain: 3b992721 | hash: 3b992721
chain: cdf6b08a | hash: cdf6b08a
chain: bb08e410 | hash: bb08e410
chain: cf847d13 | hash: 5c3d6dbf
```

Мал. 2, Зліва значення гешів для ланцюга в таблиці починаючи з кінця, зправа те саме але для ланцюга геш значення

На Мал. 2, в кінці бачимо колізію $cf847d13 \neq 5c3d6dbf$ але при тому $H(R(cf847d13)) = H(R(5c3d6dbf)) = bb08e410$ і це призводить до неправильного прообразу.

Таким чином дизайн атаки повинен перевіряти чи дійсно прообраз правильний, перед тим як його повернути і "зараховувати" успішність атаки тільки якщо дійсно повернений x такий що $H(x) = y$.

0.5 Особливості прикладної реалізації атаки мовою C

Для реалізації алгоритмів атаки було обрано мову програмування C, так як вона швидше за рахунок своєї низькорівневості.
Система де виконується атака, `Linux archlinux 6.12.1-arch1-1`, кількість потоків та ядер процесора відображено на Мал. 3.

```
$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          48 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   12
  On-line CPU(s) list:    0-11
Vendor ID:                AuthenticAMD
  Model name:             AMD Ryzen 5 5500U with Radeon Graphics
    CPU family:           23
    Model:                104
    Thread(s) per core:   2
    Core(s) per socket:   6
<SNIP>
```

Мал. 3, характеристики системи

Таким чином для розпаралелювання алгоритмів нам доступні 12 потоків, використання більшої кількості навпаки буде призводити до сповільнення роботи.
Було отримано наступні показники часу, при генерації таблиці $K = 2^{20}, L = 2^{10}$.

```
20x10
1:  163.54s user 0.02s system 99%   cpu 2:43.83 total
2:  163.65s user 0.02s system 199%  cpu 1:22.03 total
4:  163.80s user 0.01s system 397%  cpu 41.206  total
8:  233.15s user 0.05s system 737%  cpu 31.641  total
10: 268.45s user 0.09s system 951%  cpu 28.220  total
12: 303.90s user 0.13s system 1153% cpu 26.368  total
14: 318.24s user 0.08s system 1158% cpu 27.488  total
```

Мал. 4, результати часу на генерування однієї таблиці при використанні різної кількості потоків

Таким чином було обрано 8 потоків як оптимальне значення.

0.6 Оцінка затрат часу при генерації таблиць для завд. 2

Розглянемо варіант, в якому необхідно згенерувати $K = 2^{20}$ таблиць $K = 2^{20}, L = 2^{10}$, кожна з яких генерується з різним $seed$ значенням для функції редукції. Ця таблиця генерується найшвидше, адже має найменшу кількість геш-знечень необхідних для її створення. Приблизний час генерації 30 секунд. Нехитрими обчисленнями отримуємо

$$2^{20} \cdot 30 = 31457280с = 524288хв = 8737год = 364дн$$

Що трішки не вкладається в час виділений мною на цей комп'ютерний практикум...
Тому для виконання другого завдання, було прийняте рішення про зменшення кількості таблиць відповідно до часових та обчислювальних вимог.

Повні вихідні коди програмної реалізації можна знайти за посиланнямна репозиторій лабораторної роботи на моєму github.

0.7 Таблиці

За для економії часу (і нервів) було вирішено після генерації зберігати таблиці у вигляді файлів разом з випадковим значенням використаним для функції редукції. Таблиці відсортовані за другим стовпчиком, для підвищення ефективності при атаці завдяки використанню бінарного пошуку.
Деякі помічені властивості при аналізі найменшої з таблиць:

- Кількість унікальних значень в лівій колонці (стартових повідомлень) згенерованих випадково в найменшій таблиці 99.97% тобто пару сотень з них повторюється зменшуючи ефективну кількість рядків
- В той же час кількість унікальних значень в кінці ланцюга (другому стовпчику) складає орієнтовно 88.98%.

1. Проведення атаки за допомогою однієї таблиці передобчислень

1.1 Генерація таблиць

В результаті реалізації програмного коду атаки, було отримано функціонал що дозволяє доволі ефективно генерувати таблиці обраного розміру (K, L) та зберігати їх у вигляді файлу для подальшого використання в атаках.
В основній версії практикуму необхідно згенерувати 9 таблиць, назви яких я присвоюю використовуючи степені двійки для їх параметрів (K, L). Таким чином було отримано наступні таблиці:

$$20 \times 10,\ 20 \times 11,\ 20 \times 12,\ 22 \times 10,\ 22 \times 11,\ 22 \times 12,\ 24 \times 10,\ 24 \times 11,\ 24 \times 12$$

При генерації таблиць було обраховано час витрачений на їх генерацію, що необхідно для оцінки генерації великої кількості таблиць такого розміру, та безпосередній розмір таблиць, для оцінки пам'яті необхідної для паралельного утримання певної їй кількості під час атаки.
Отримали наступні результати:

table	time	size
20×10	0:32.209	8388624
20×11	1:00.42	8388624
20×12	1:47.93	8388624
22×10	1:57.51	33554448
22×11	3:51.25	33554448
22×12	7:11.87	33554448
24×10	7:11.45	134217744
24×11	14:53.67	134217744
24×12	30:02.07	134217744

Табл. 1, час і розмір генерації таблиць завдання 1

1.2 Proof of Concept

Продемонструємо приклад атаки на найменшій таблиці.

```
$ ./hellman
Hellman's time-memory tradeoff
Parameters: K = 1048576, L = 1024
Using BLAKE2b with 32-byte output
Table exists, loading...
Searching for preimage...
Iteration 0
Searching for 6a1b7466 preimage...
y: 6a1b7466, input: f1c4a7015d90901c168dea5f6a1b7466
y: 474edfe8, input: f1c4a7015d90901c168dea5f474edfe8
y: 12e31149, input: f1c4a7015d90901c168dea5f12e31149
y: 3148b510, input: f1c4a7015d90901c168dea5f3148b510
y: e6afc6f9, input: f1c4a7015d90901c168dea5fe6afc6f9
y: b76da46a, input: f1c4a7015d90901c168dea5fb76da46a
y: d2a7695a, input: f1c4a7015d90901c168dea5fd2a7695a
y: 6492f4bc, input: f1c4a7015d90901c168dea5f6492f4bc
...
<SNIP>
...
f1c4a7015d90901c168dea5ffe1eab53y: 4c51d3b5, input:
f1c4a7015d90901c168dea5f4c51d3b5y: c9950391, input:
f1c4a7015d90901c168dea5fc9950391y: 36be98f5, input:
f1c4a7015d90901c168dea5f36be98f5Found chain 121404 with value at 781
Calculating preimage...
Current y(j): 1da2a398(243)
Chain: [5ee419e9] -> [1da2a398]
[0] x: 5ee419e9, reduced input:f1c4a7015d90901c168dea5f5ee419e9 Current hash:
3b89760f -> target hash 6a1b7466
[1] x: 3b89760f, reduced input:f1c4a7015d90901c168dea5f3b89760f Current hash:
1a51a496 -> target hash 6a1b7466
[2] x: 1a51a496, reduced input:f1c4a7015d90901c168dea5f1a51a496 Current hash:
1f901748 -> target hash 6a1b7466
[3] x: 1f901748, reduced input:f1c4a7015d90901c168dea5f1f901748 Current hash:
666420a5 -> target hash 6a1b7466
[4] x: 666420a5, reduced input:f1c4a7015d90901c168dea5f666420a5 Current hash:
ea40b0f8 -> target hash 6a1b7466
[5] x: ea40b0f8, reduced input:f1c4a7015d90901c168dea5fea40b0f8 Current hash:
b115087b -> target hash 6a1b7466
[6] x: b115087b, reduced input:f1c4a7015d90901c168dea5fb115087b Current hash:
f7c716e4 -> target hash 6a1b7466
[7] x: f7c716e4, reduced input:f1c4a7015d90901c168dea5ff7c716e4 Current hash:
581c7af7 -> target hash 6a1b7466
```

```
[8] x: 581c7af7, reduced input:f1c4a7015d90901c168dea5f581c7af7 Current hash:
5d039833 -> target hash 6a1b7466
[9] x: 5d039833, reduced input:f1c4a7015d90901c168dea5f5d039833 Current hash:
af415979 -> target hash 6a1b7466
[10] x: af415979, reduced input:f1c4a7015d90901c168dea5faf415979 Current hash:
601f7a21 -> target hash 6a1b7466
[11] x: 601f7a21, reduced input:f1c4a7015d90901c168dea5f601f7a21 Current hash:
3bdd5edc -> target hash 6a1b7466
[12] x: 3bdd5edc, reduced input:f1c4a7015d90901c168dea5f3bdd5edc Current hash:
d80020ad -> target hash 6a1b7466
...
<SNIP>
...
[768] x: 81d267b3, reduced input:f1c4a7015d90901c168dea5f81d267b3 Current
hash: 9bc3bf80 -> target hash 6a1b7466
[769] x: 9bc3bf80, reduced input:f1c4a7015d90901c168dea5f9bc3bf80 Current
hash: d7794cc3 -> target hash 6a1b7466
[770] x: d7794cc3, reduced input:f1c4a7015d90901c168dea5fd7794cc3 Current
hash: 380a3e80 -> target hash 6a1b7466
[771] x: 380a3e80, reduced input:f1c4a7015d90901c168dea5f380a3e80 Current
hash: 01b1293e -> target hash 6a1b7466
[772] x: 01b1293e, reduced input:f1c4a7015d90901c168dea5f01b1293e Current
hash: 27960d20 -> target hash 6a1b7466
[773] x: 27960d20, reduced input:f1c4a7015d90901c168dea5f27960d20 Current
hash: f06ec887 -> target hash 6a1b7466
[774] x: f06ec887, reduced input:f1c4a7015d90901c168dea5ff06ec887 Current
hash: b34584bc -> target hash 6a1b7466
[775] x: b34584bc, reduced input:f1c4a7015d90901c168dea5fb34584bc Current
hash: c7a2f8dc -> target hash 6a1b7466
[776] x: c7a2f8dc, reduced input:f1c4a7015d90901c168dea5fc7a2f8dc Current
hash: f866f01f -> target hash 6a1b7466
[777] x: f866f01f, reduced input:f1c4a7015d90901c168dea5ff866f01f Current
hash: 9cd99ff7 -> target hash 6a1b7466
[778] x: 9cd99ff7, reduced input:f1c4a7015d90901c168dea5f9cd99ff7 Current
hash: 2793ba0d -> target hash 6a1b7466
[779] x: 2793ba0d, reduced input:f1c4a7015d90901c168dea5f2793ba0d Current
hash: a61c23ab -> target hash 6a1b7466
y: 1da2a398, input: f1c4a7015d90901c168dea5f1da2a398
y: d38535c8, input: f1c4a7015d90901c168dea5fd38535c8
...
<SNIP>
...
Current y(j): b11559d9(1019)
Chain: [091e1fd9] -> [b11559d9]
[0] x: 091e1fd9, reduced input:f1c4a7015d90901c168dea5f091e1fd9 Current hash:
450d7f8e -> target hash 6a1b7466
```

```
[1] x: 450d7f8e, reduced input:f1c4a7015d90901c168dea5f450d7f8e Current hash:
8dac73ee -> target hash 6a1b7466
[2] x: 8dac73ee, reduced input:f1c4a7015d90901c168dea5f8dac73ee Current hash:
94048c21 -> target hash 6a1b7466
[3] x: 94048c21, reduced input:f1c4a7015d90901c168dea5f94048c21 Current hash:
b02722e4 -> target hash 6a1b7466
y: b11559d9, input: f1c4a7015d90901c168dea5fb11559d9
Found chain 620611 with value at 4
Calculating preimage...
Current y(j): 979c46fa(1020)
Chain: [04ed4be1] -> [979c46fa]
[0] x: 04ed4be1, reduced input:f1c4a7015d90901c168dea5f04ed4be1 Current hash:
a7feb6d5 -> target hash 6a1b7466
[1] x: a7feb6d5, reduced input:f1c4a7015d90901c168dea5fa7feb6d5 Current hash:
dddd27ec -> target hash 6a1b7466
[2] x: dddd27ec, reduced input:f1c4a7015d90901c168dea5fdddd27ec Current hash:
3b15f3df -> target hash 6a1b7466
y: 979c46fa, input: f1c4a7015d90901c168dea5f979c46fa
Found chain 409852 with value at 3
Calculating preimage...
Current y(j): 642eb706(1021)
Chain: [351d2540] -> [642eb706]
[0] x: 351d2540, reduced input:f1c4a7015d90901c168dea5f351d2540 Current hash:
69ade379 -> target hash 6a1b7466
[1] x: 69ade379, reduced input:f1c4a7015d90901c168dea5f69ade379 Current hash:
2b1630a6 -> target hash 6a1b7466
y: 642eb706, input: f1c4a7015d90901c168dea5f642eb706
Found chain 267336 with value at 2
Calculating preimage...
Current y(j): 415cc98f(1022)
Chain: [468ee0cf] -> [415cc98f]
[0] x: 468ee0cf, reduced input:f1c4a7015d90901c168dea5f468ee0cf Current hash:
eda477f7 -> target hash 6a1b7466
y: 415cc98f, input: f1c4a7015d90901c168dea5f415cc98f
Found chain 91752 with value at 1
Calculating preimage...
Current y(j): 1650e87d(1023)
Chain: [7d76929b] -> [1650e87d]
y: 1650e87d, input: f1c4a7015d90901c168dea5f1650e87d
```

Мал. 5, приклад неуспішної атаки

Як ми можемо побачити трюк з випадковим 256 бітовим повідомленням не працює, по причині яку я пояснив в аналізі завдання - цього повідомлення немає серед множини входів, відповідно його геш може і взагалі не мати фізичної можливості з'явитись в

табличці.
Якщо ми проведемо 10 тисяч таких атак на геші що отримані з абсолютно випадкових повідомлень, то побачимо сумні результати.

```
$ ./hellman
Hellman's time-memory tradeoff
Parameters: K = 1048576, L = 1024
Using BLAKE2b with 32-byte output
Table exists, loading...
Searching for preimage...
Iteration 0
Iteration 1000
Iteration 2000
Iteration 3000
Iteration 4000
Iteration 5000
Iteration 6000
Iteration 7000
Iteration 8000
Iteration 9000
Success rate: 212/10000
```

Мал. 6, атака на 10000 гешів абсолютно випадкових повідомлень

Така поведінка зумовлена нерівномірністю розподілу нашої "обрізаної" геш функції. При побудові цієї атаки на повноцінну версію, проблема неоднорідності виходів не виникала би, так як геш-функція за побудовою рівномірне відображення, і ми б могли сподіватись що геш будь-якого випадкового повідомлення матиме шанс бути серед таблиці, але в нашому випадку це не так, адже множини повідомлень серед яких ми шукаємо, і серед яких ми будуємо таблицю - кардинально різні, і скоріш за все не матимуть перетину на множині виходів нашої "обрізної" геш функції.

Адекватна робота атаки над гешами випадкових 256 бітних повідомлень передбачається тільки з використанням певної кількості різних таблиць, з різними функціями редукції, що дозволить більш-менш рівномірно покрити простір виходів "обрізаної" геш-функції. Результати такого експерименту ми побачимо в другій частині практикуму.

Таким чином, тут і в подальших атаках геш-значення які ми атакуємо, додатково до випадкових, будуть обчислені з повідомлень які належать множині вхідних, а саме:

- Буде згенероване випадкове 32 бітне число
- Це число пройде через функцію редукції для цієї таблиці

- Отриманий результат - 128 бітне повідомлення, для гешу якого ми будемо намагатись знайти прообраз.
  Таким чином геші для яких ми намагатимемось знайти прообраз, будуть хоча б мати шанс опинитись в таблиці.
  Приклад для такого вибору вхідних повідомлень і гешів:

```
$ ./hellman
Hellman's time-memory tradeoff
Parameters: K = 1048576, L = 1024
Using BLAKE2b with 32-byte output
Table exists, loading...
Searching for preimage...
Iteration 0
Iteration 1000
Iteration 2000
Iteration 3000
Iteration 4000
Iteration 5000
Iteration 6000
Iteration 7000
Iteration 8000
Iteration 9000
Success rate: 8969/10000
```

Мал. 7, атака на 10000 гешів повідомлень що належать простору входів

Отримуємо вже більш адекватний результат, якого ми і хотіли б очікувати від подібної атаки на зменшену версію геша.

1.3 Розгортання атаки на 10000 гешів за допомогою кожної за таблиць

Проведемо експеримент, в якому ми намагатимемось знайти прообраз для гешу за допомогою обраної таблиці передобчислень. Для кожної попередньо згенерованої таблиці проведемо 10000 спроб, запишемо кількість з них яка була успішною.

Table	Succ. Rate on 256 random	Succ. Rate on Input value space
20×10	$212/10000$	$8969/10000$
20×11	$177/10000$	$7984/10000$
20×12	$99/10000$	$6907/10000$
22×10	$350/10000$	$6976/10000$
22×11	$260/10000$	$5318/10000$

Table	Succ. Rate on 256 random	Succ. Rate on Input value space
22×12	$242/10000$	$3647/10000$
24×10	$481/10000$	$3900/10000$
24×11	$382/10000$	$2538/10000$
24×12	$277/10000$	$1573/10000$

Табл. 2, результати атаки з використанням однієї таблиці для різних типів вхідних повідомлень і таблиць

Можемо бачити як падає кількість успіхів в залежності від розмірів таблиці, це пов'язано з збільшенням кількості колізій. Таким чином вже на таблиці 22×12 ми маємо наступні показники:

```
In [20]: len(values)
Out[20]: 4194304

In [21]: len(set(values))
Out[21]: 1376605
```

Мал. 8, кількість унікальних значень в другому стовпчику таблиці 22x12

Бачимо що в таблиці залишилось лише близько 30% унікальних значень - все інше ланцюжки які злиплись. В той же час в таблиці 20×10 кількість унікальних значень близько 89%. На найбільшій таблиці 24×12 унікальних значень лишається всього 10%.

2. Проведення атаки за допомогою декількох таблиць передобчислень

2.1 Оцінки часу на генерацію певної кількості таблиць

Знаючи час та розмір таблиць, ми можемо попередньо обрахувати приблизні значення для часу необхідного на генерацію N таблиць, а також визначити скільки таблиць ми зможемо одночасно тримати в пам'яті для синхронного пошуку за допомогою потоків.

2.2 Особливості технічної реалізації

В пояснені завдання в методичних вказівках наголошено що обчислення на декількох таблицях мають виконуватись синхронно. Тобто логічно розділити пошук прообразу на потоки, що збільшить швидкість. Таким чином я прийшов наступних висновків.

- Використовуючи ресурси мого ноутбука генерувати K таблиць фактично не можливо, час необхідний на генерацію K найменших таблиць (тобто дефакто найшвидший набір даних який у нас є), описаний в аналізі завдання, і сильно перевищує часові рамки лабораторної роботи.
- Генерувати всі 9 типів таблиць, на ресурсах мого ноутбука займатиме неймовірно багато часу, не менше часу займатиме сама атака. Таким чином для порівняння роботи наборів з різних розмірів таблиць було обрано залишити лише один варіант для параметра L, а саме 2^{10}. Це було зроблено з огляду на те що в попередньому експерименті збільшення параметру L приводило до збільшення шансу колізій і гіршої роботи таблиці.
- Генерація відбувається з розбиттям на оптимальну кількість потоків, так як це було в першій частині лабораторної.
- Атака на декількох таблицях також може бути розбита на потоки. Виділити потік на кожну таблицю ми не можемо, адже у нас не має стільки ядер, а кількість потоків більша за кількість ядер просто змусить процесор стрибати між ними, і тим самим сповільнюватись. Тому, візьмемо визначену нами оптимальну кількість потоків t, і кожному потоку присвоїмо $\frac{N}{t}$ послідовних таблиць з набору. При необхідності знайти прообраз для гешу, кожен потік починає шукати його послідовно в кожній таблиці свого набору, і коли прообраз був знайдений, всі потоки зупиняються.
- З огляду на час необхідний на генерацію однієї таблиці певного розміру, ми можемо прикинути кількість таблиць яка вкладеться в приблизно 4 години генерації. Таким чином було визначено наступні числа:
  - Таблиці розміру 20×10 - 256 штук, приблизний очікуваний час генерації 2 години,
  - Таблиці розміру 22×10 - 128 штук, приблизний очікуваний час генерації 4 години,
  - Таблиці розміру 24×10 - 28 штук, приблизний очікуваний час генерації 4 години.

2.3 Результати роботи

Проведемо атаку так само як і в першій частині лабораторної, але з використанням декількох передгенерованих таблиць і алгоритмів розпаралелювання наведених в 2.2.

Table	Succ. Rate on 256 random	Succ. Rate on Input value space	Number of tables used
20×10	$9954/10000$	$9991/10000$	256
22×10	$9891/10000$	$9895/10000$	128
24×10	$7395/10000$	$8361/10000$	28

Табл. 3, результати атак з використанням декількох таблиць передобчислень

Отримані результати повністю співпадають з очікуваними. На великому наборі таблиць, кожна з яких має різний простір вхідних значень завдяки різним функціям редукції різниця між гешом абсолютно випадкового повідомлення, та гешом повідомлення з функції редукції стирається і стає не значною. Це відбувається тому що різні таблиці більш рівномірно покривають розподіл вихідних значень обрізаної геш-функції.

3. Аналіз отриманих результатів

3.1 Теоретичні оцінки складності та успіху

Складність обрахування таблиці $K \times L$, буде відповідно KL операцій гешування.
Складність пошуку прообразу по згенерованій таблиці буде
$L * \textit{складність бінарного пошуку по таблиці}$.
На зберігання таблиці необхідно $2K * 4$ байт, так як маємо дві колонки довжини K які містять 32 бітні значення.

З лекційного матеріалу курсу, а також оригінальної публікації Геллмана по TMTO
ми можемо побачити, що найменші обрані параметри K, L переважують оптимальну нижню границю, $KL^2 = N$. Далі $m = K, t = L$
Теорема Геллмана

$$p_{\text{succ}} \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{N}\right)^{j+1}$$

В нашому випадку значення

$$[(N - it)/N]^{j+1} = exp(-ijt/N)$$

Останній такий доданок буде оцінюватись як

$$exp(-\frac{mt^2}{N})$$

І бути супер малим, тобто збільшення розміру таблиці не суттєво збільшує ймовірність атаки, і трешхолд поставлений в $mt^2 = N$ не перевищуватиме ймовірність успіху
$P(S) = 1/(N^{1/3})$
що збільшуватиметься лише при генерації більшої кількості таблиць з різними функціями надлишковості.

3.2 Аналіз результатів отриманих практично

Практичні результати були обгрунтовані відразу під час їх отримання, і їх аналіз детально описаний в попередніх пунктах.
При порівнянні з теоретичними можна впевнено сказати що враховуючи умови лабораторної роботи, теоретичні оцінки і практичні результати корелюють.

3.3 Висновки

Для оптимальної роботи атаки, для таблиць треба класти менші значення K та L, а саме $KL^2 = N$, так як після цієї межі їх збільшення буде лише додавати ймовірності колізій в ланцюгах і не збільшувати ймовірність успіху. Також такі таблиці будуть швидше обчислюватись, ніж їх перебільшені версії, що дасть змогу ефективніше обчислити велику кількість таких таблиць з різними функціями редукції.

Також доцільним та цікавим може бути експеримент з використанням більш сучасних підходів то побудови TMTO атак, наприклад [https://hal.science/hal-04444552/file/ascending_Stepped_RT.pdf].