Time–Memory Trade-Off Attacks on Password Hashes using Rainbow Tables

Cryptography Course Project — IIIT Sri City

Dr Kamalakanta Sethi

Team:

Abhishek Sahay	**S20220010003**
Kummari Neeraj Kumar	**S20220010116**
Lavudi Nokesh	**S20220010125**

THE PROBLEM: CRACKING HASHES

- Imagine a server is breached. The attacker doesn't get your password, they get this:

```
>_

users.db (Leaked)
  alice: 5d41402abc4b2a76b9719d911017c592
  bob:   e175ea4ce0a553260a14bc5e922a935b
  charlie: 711185fa17de642583dfd5ecd13442e2
```

- A hash is a one-way function. You can't "un-hash" or "decrypt" it.

THE PROBLEM: CRACKING HASHES

The only way to crack it is to guess:

Is the password '12345'? -> HASH('12345') -> `827ccb...` -> No match.
Is the password 'ab1'? -> HASH('ab1') -> `e175ea...` -> Match!

TWO "OBVIOUS" SOLUTIONS

Option 1: Brute Force

(All Time, No Memory)

- Try 'a', hash it...
- Try 'aa', hash it...
- Try 'ab', hash it...

Problem: Infeasibly slow. The password space is enormous. Cracking one 6-character alphanumeric password could take days or weeks.

Option 2: Giant Lookup Table

(No Time, All Memory)

- Pre-compute every password and its hash.
- 'a': `0cc175...`
- 'aa': `96201b...`

Problem: Infeasibly large. A 6-char alphanumeric table has 56.8 billion entries. The table would be terabytes in size.

THE REAL SOLUTION: A TIME-MEMORY TRADE-OFF

We need a solution that is *smarter than brute-force but smaller than a lookup table*.

This is the "Time-Memory Trade-Off"

- Pre-computation (Time Cost): We will spend time once, up-front, to generate a special table.
- Storage (Memory Cost): We will store this table, which is much smaller than a full lookup table.
- Payoff (Success Rate): We can now crack hashes much faster, with a high probability of success.

This is exactly what a **Rainbow Table** does.

WHAT IS A RAINBOW TABLE?

- A Rainbow Table is a pre-computed data structure that cleverly compresses a giant lookup table.
- It does not store every password and hash.
- Instead, it stores only the START (Head) and END (Tail) of long "chains" of passwords and hashes.
- A single chain of 200 passwords and hashes is "compressed" into just two entries, dramatically saving space.

CORE CONCEPTS

1. Hash Function (H)

The one-way function we're attacking (e.g., `SHA1`).

Password -> H( ) -> Hash

'ab1' -> H() -> e175ea...

2. Reduce Function (R)

The "magic" function that turns a hash back into a password.

Hash -> REDUCE -> Password

(It's not "un-hashing", just a consistent way to map a big number to a short string. We use different functions, `R1`, `R2`, `R3`... at each step to prevent chains from merging.)

e175ea... -> R() -> 'Xy3zK'

3. Password Chain

A sequence of alternating Hash and Reduce functions:

(Head) 'aaaaa' --H()--> hash1 --R1()--> 'pw2' --H()--> hash2 --R2()--> ... --H()--> hash200 (Tail)

In our table, we only store: ('aaaaa', 'hash200')

HOW RAINBOW TABLES WORK: CRACKING

We have a Target Hash from the database. How do we find it?

1. Is Target Hash in our list of "Tails"? (This is a "perfect match", very rare).

2. If not:
- Run Target Hash through the last reduce function:
  `*R199(Target Hash) -> pw_guess*`
- Hash that: `*H(pw_guess) -> hash_guess*`
- Is `hash_guess` in our list of "Tails"?

3. If not:
- Run Target Hash through the last two reduce/hash functions:
  `*R198(...) -> ... -> hash_guess_2*`
- Is `hash_guess_2` in our list of "Tails"?

4. ...We get a HIT! Our `hash_guess_X` matches a "Tail".

5. We look up the "Head" password for that chain (e.g., 'aaaaa').

6. We regenerate the chain from 'aaaaa' until we find our Target Hash. The password right before it is our answer!

OUR PROJECT IMPLEMENTATION

We built a set of Python scripts to demonstrate this entire process:

- rainbowtable.py: The main Python class that handles all the logic.
  - hash_function(): Implements SHA1 and MD5.
  - reduce_function(): The key to our success! A robust function that maps hashes to the password keyspace and avoids collisions.
  - generate_table(): Creates the chains and stores the Head/Tail pairs.
  - lookup(): Implements the cracking logic from the previous slide.
- rainbowgen.py: A command-line tool to build and save new tables.
- rainbowcrack.py: A tool to load a table and crack a single hash.
- analyze_tradeoff.py: Our experiment script to run the tests and generate the final plots.

EXPERIMENTAL SETUP

To prove the trade-off, we designed an experiment to measure how "Memory Cost" affects "Success Rate".

- Algorithm: sha1
- Charset: alphanumeric (62 chars: a-z, A-Z, 0-9)
- Password Length: 1 to 6 characters
- Chain Length: 200 (Constant)

The Test:

1. We generated a fixed set of 100 random "target" hashes.
2. We built 5 different tables with varying numbers of chains: 1k, 5k, 10k, 20k, 40k
3. We ran all 100 target hashes against each table and measured its Success Rate (%).

HANDS-ON: LIVE DEMO

1. Generate a Rainbow Table

```
python3 rainbowgen.py sha1 alphanumeric 1 6 200 10000 my_table.rt
```

2. Crack a Hash

```
python3 rainbowcrack.py e175ea4ce0a553260a14bc5e922a935b40425c1e my_table.rt
```

3. Run the Analysis Experiment

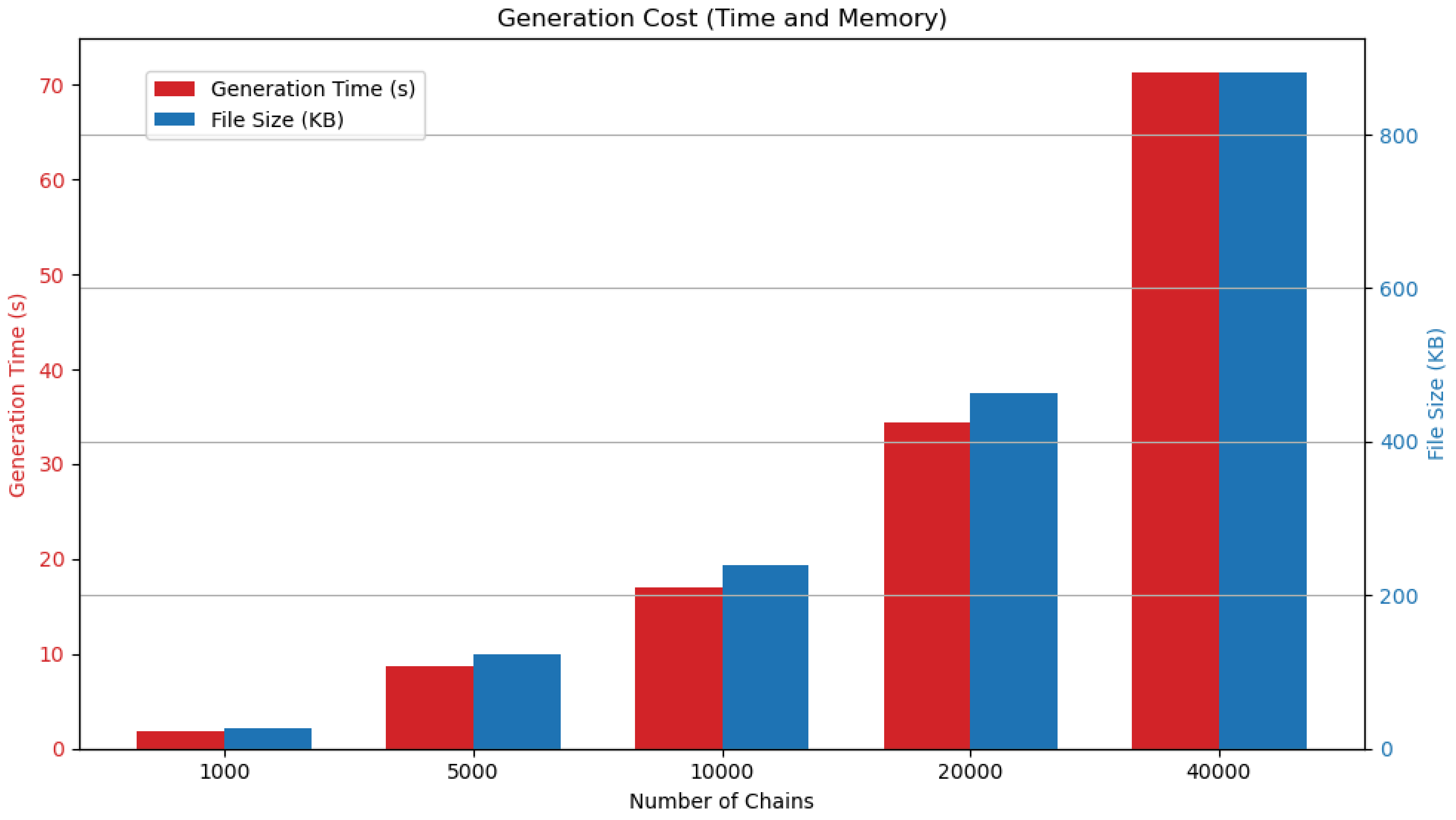
```
python3 analyze_tradeoff.py
```

RESULTS: THE "GENERATION COST" PLOT

First, we measured the "cost" of building the tables.

Finding: The cost is perfectly linear. As we increase the number of chains, the File Size (KB) and Generation Time (s) both increase proportionally.

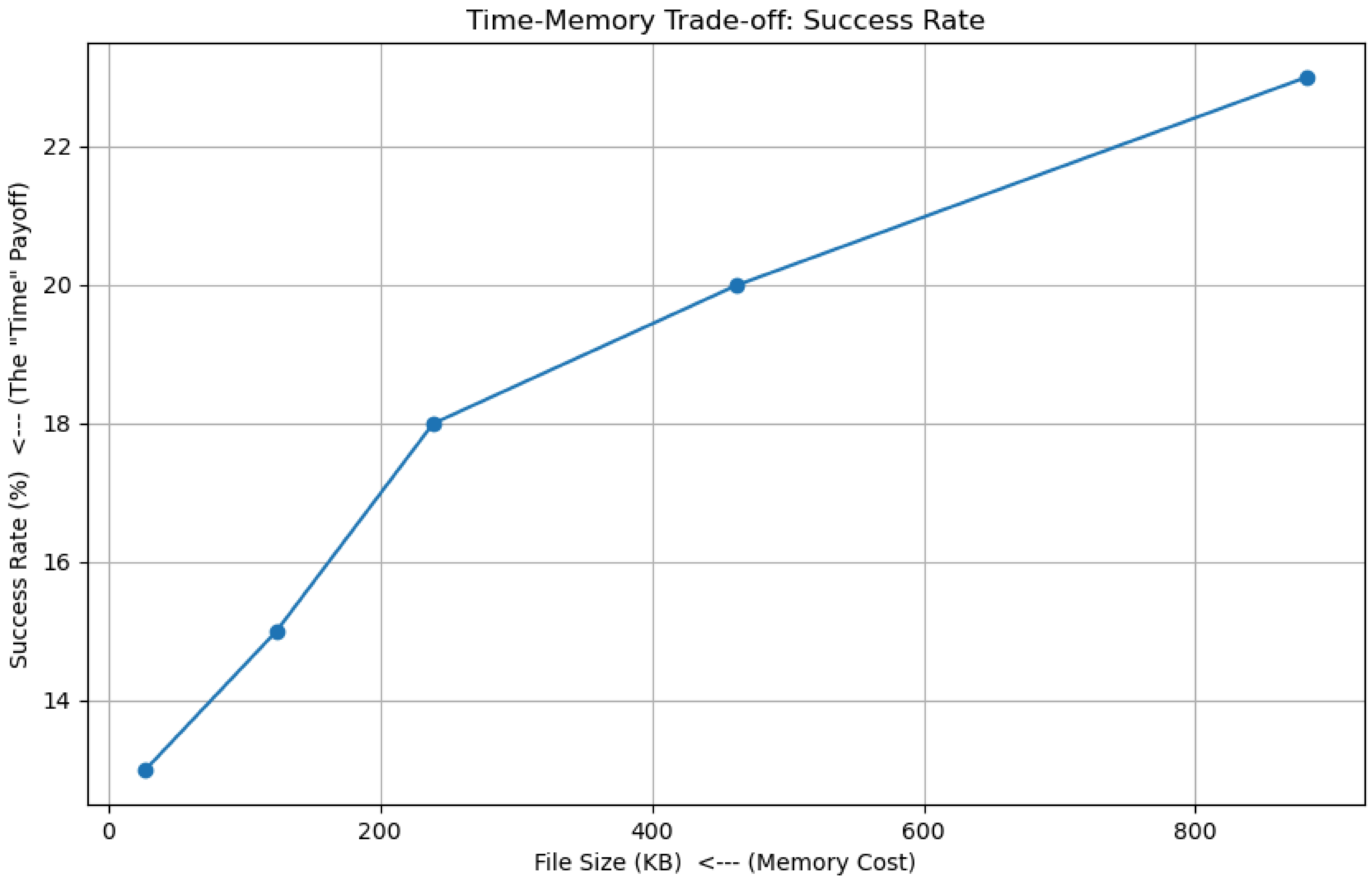
This proves our `reduce_function` is working and we are not getting collisions.



RESULTS: THE "PAYOFF"

Chains	File Size (KB)	Generation Time (s)	Success Rate (%)
1000	27.5186	1.7888	16.0
5000	122.6064	8.8787	18.0
10000	236.784	16.3868	24.0
20000	459.3545	35.7878	25.0
40000	883.4658	71.2410	30.0

RESULTS: THE TRADE-OFF PLOT



CONCLUSION

- We successfully built a working Rainbow Table generator and cracker in Python.
- We proved that the quality of the reduce_function is the most critical part of preventing chain collisions.
- We successfully demonstrated the Time-Memory Trade-off:
  - Time Cost: We paid a one-time, linear "pre-computation" cost.
  - Memory Cost: We traded file size...
  - ...for a "Payoff": A much higher probability of successfully cracking a hash.
- This attack is powerful, but it is defeated by "salting" passwords, which makes pre-computed tables useless.

Thank You