

Evaluating Password Cracking Techniques: A Study on Security Vulnerabilities and Protection Mechanisms

F415581

This manuscript was compiled on October 25, 2024

Abstract

Passwords are still the most widely used authentication mechanism due to their simplicity and flexibility. Approximately 83% of organizations globally use password-based authentication for access to IT resources (Hatice Ozsahan 2023) and their integrity as an authentication mechanism is contingent on their confidentiality. This report details research done into cryptographic hash functions (CHFs) which are used for securely storing passwords and the role played by password strength in protecting against offline brute force password attacks. We compare the time required to crack passwords with varying levels of complexity and 2 hashing algorithms (SHA256 and MD5). We find that the factors that most influenced brute-force resistance was the inclusions of salts and increasing password the character set and length of the password.

Keywords: *Cryptographic hash function, password cracking, brute-force, complexity, John the Ripper*

Rho LaTeX Class © This document is licensed under Creative Commons CC BY 4.0.

1. Introduction

Passwords, initially used as passphrases to confirm identity before the advent of computers (Clark 2007), remain a reliable authentication method due to their simplicity and variability in strength. However, passwords are only secure if unauthorized individuals cannot obtain them. Today, most services, like Netflix and LinkedIn, store credentials with encryption and hashing to protect database leaks. Login attempt limitations shift attackers toward offline attacks, where they target stolen hashed passwords. With 83% of organizations still relying on username-password authentication (Hatice Ozsahan 2023), understanding its strengths and weaknesses is crucial. This report discusses experiments using John the Ripper (JtR) and examines cryptographic hashing, hash salting, and password strength, offering a foundational understanding of password security

2. Literature Review

2.1. Importance of password security

When investigating the choice of passwords and password re-use, Hanamsagar et al. investigated what lead to the use of weak passwords. They found the underestimation of the importance of the use of strong passwords was not correlated with strength of passwords. Here are some of factors they found to be linked to why users created weak passwords:

1. Memorability - Users wanted easier to remember passwords
2. Number of password vs number of accounts - Lead to password reuse/password versioning (memorability & convenience)
3. Not realising the importance of password length - Short and weaker passwords
4. Reluctance towards having to reset passwords frequently - Hence memorability
5. Lax organizational password policies
6. Not realising how fast password cracking algorithms guess passwords even with slow hashing algorithms

Though the responsibility is on the user to create a strong unique password, strong password policies and preventing leaks should be prioritised. Database leaks a significant reason passwords are vulnerable to offline password attacks.

Data breaches which result in the theft of user information (not just passwords) are estimated to cost \$4.8 million dollars on average per leak in 2024 (IBM 2024) yet it is estimated most companies spend only 7% to 20% of their IT budget on cyber security (Martisiute 2023).

Passwords are still used by 83% of companies as their main authentication method for the access of IT resources so it is important that companies find ways of minimizing the frequency of data breaches.

If memorability is being prioritised then instead of using something obvious or more easily guessable, an alternative is to use mnemonics to that allow you to remember the password but keep it just complicated enough to make it very hard to guess ((CISA) 2024) Considering the vulnerabilities passwords have and the many alternatives available, why do we still use passwords? Passwords do have weaknesses that can be exploited, but all alternatives their costs. Discontinuing passwords would require resource investment and also poses different risks. Authentication mechanisms like biometrics are not instance isolated but can impact all aspects of life for any compromised individual should their biometric data be stolen. (Herley, Oorschot, and Patrick 2009).

2.2. John the Ripper

John the Ripper is a widely used password cracking algorithm among both security professionals and cyber criminals Marchetti and Bodily 2022. Marchetti et al. note that it is considered a poor brute-force attacker, but I argue that the effectiveness of brute-force attacks inherently depends on the time required to compute every possible combination and therefore there is technically no such thing as a "good brute-force attacker". In their report they show how effective JtR is at dictionary attacks and show that changing hashing algorithms had no effect (but that was for dictionary attacks). In following sections we will investigate if the same is true but for brute force attacks.

3. Model Overview

Title of Model:

- Password Cracking Techniques Using John the ripper

Purpose:

- To analyse the effectiveness and efficiency of brute-force password attacks implemented by John the Ripper, focusing on MD5 and SHA256 and the impact of password complexity on these attempts.

3.1. Theoretical Background

Cryptographic Hash Functions (CHFs) and the security and integrity of hash functions are discussed in the below subsections

3.1.1. Cryptographic hash functions

The National Institute of Standards and Technology (NIST) describe Cryptographic Hash Functions (CHFs), as functions that map bit strings of arbitrary lengths to a bit string of fixed length (NIST 2023). All approved hash functions are expected to have the following properties.

1. Must be a one-way function. Meaning that you should not be able to compute the key using the outputted hash. For example;
 - Let H be the Hash function
 - Let x be the input
 - Let Output of $H(x)$ be the hash value y
 - Let G be some arbitrary function that reverses the hashing process
 - For H to be an approved hash the bellow must be true

$$\nexists G : G(y) = x$$

2. All approved hash functions must be collision-resistant. Meaning each key that is hashed by an approved hash function must produce a hash that is unique and that cannot be obtained by hashing any other key. For example;

- Let there be some arbitrary hash function H
- Let there be some arbitrary inputs

$$n_1, n_2, n_3 \dots n_{\text{inf}}$$

For H to be an approved hash function the following must be true

$$\nexists n_p, n_m : H(n_p) = H(n_m)$$

Property 1 allows for entities to only have to store hashed passwords instead of plain text in order to negate the impact possibility of a data leak. Although collision-resistance is required, in theory, it is impossible to create a hash function that is completely collision-resistant due to the infinite set of possible passwords and the finite possible set of outputs to a given hash function (since CHF's output fixed length strings).

Collision resistance reduces the **likelihood** of more than one password input existing for a given hash and which allows for the more secure storage of passwords as hashes.

3.1.2. Security of Hash Functions

The security of hash functions comes down to a few aspects:

- Collision resistance - Reduces vulnerability to Collision attacks
- Time taken to compute hash - Increases time taken for brute force attacks
- Presence of salts - Makes it almost impossible to brute force

Eliminating collisions is usually the main aim as the later 2 aspects have their own disadvantages in industry wide use namely;

1. Hash computation time - When used for many users, may slow down authentication time and sign in process
2. Hash Salting - When dealing with a high volume of users, it is computationally expensive to have a unique salt for all users and having one salt for all users defeats the purpose for having salts in the first place (Authgear 2024)

These factors are partly why SHA256 is still used even though there are more secure alternatives available. It is quicker to compute than more secure CHF's (Mia 2023) but its security can be bolstered with salting if necessary.

3.1.3. Collision Resistance

Collision resistance has been a long standing problem with hash functions. In 2009 NIST released details regarding MD5's lack of collision resistance (NIST 2009). This allowed attackers to perform spoofing attacks on a specific digital certificate that relied on MD5's integrity. As recently as 2017 SHA-1 was discovered to be not collision-resistant (NIST 2017) which emphasised the need for newer and stronger hashing algorithms. In the end it is not feasible to create a hashing algorithm that is 100% collision resistant, But it is possible to reduce the chance of collisions occurring, especially when it comes to passwords.

For more information on collisions take a look at :Vadhan 2013.

To reduce the chance of a collision salts can be used. Salts in cryptographic hashing, are random bits or strings added to the input of the hash function (Arias 2021), in order to make it more difficult to find the corresponding password of the produced hash. With a salt, even if you hash the correct password, unless you know there is an added salt and compute it as well, you cannot verify that you have the right password. This effectively raises the complexity of the password without having to inconvenience the user. Salts make impractical to brute force passwords.

3.2. Methodology

Assumptions

- Attackers Know what hashing algorithms the passwords are stored as
- Attackers have 6-12 hours to crack passwords
- Attackers can only use a brute force attack

Expected Outcomes

- **All** level 1 Passwords will be solved
- **All** level 2 Passwords will be solved
- **No** level 6 Passwords will be solved

Limitations

- Time constraints mean experiment cannot mimic real world setting and give attackers more time and access to more tools

3.3. Experimental Setup

Password varied by:

1. Their complexity
2. Their Hashing algorithm

3.3.1. Scripts and code

Example script:

Code run in Kali Linux terminal to start the brute force attack using 6 processors for SHA256:

```
1 --incremental --fork=6 --format="Raw-SHA256" ~/Desktop/Hashed_Passwords/Level2/
  level2_sha256_output.txt
```

3.3.2. Experimental Setup

List of 10 passwords for each level

The Level of complexity of passwords is as follows;

- Level 1: Simple (Lowercase letters, complete word, 8 characters)
- Level 2: Simple (Uppercase letters, complete word, 8–9 characters)
- Level 3: Complex (Mixed case, numbers, special characters, no word requirement, 9–11 characters)

There 2 hashing algorithms that are used;

1. MD5
2. SHA 256

Relevant host Machine Specifications:

- Processor: AMD Ryzen 7 6800H, 3201 Mhz, 8 Core(s), 16 Logical Processor(s)
- Installed Physical Memory (RAM): 16.0 GB
- System Type x64-based PC

Virtual Machine:

- Operating system: Ubuntu 64-bit (Kali linux)
- Base Memory: 6915 MB
- Acceleration: Nested Paging, Nested VT-x/AMD-V, KVM Paravirtualization
- Storage: 30.68 GB
- Number of virtual CPUs available to Virtual Machine (VM): 6

4. Results

Table 1 presents the number of passwords cracked within 12 hours using different algorithms and varying password complexity levels. The algorithms tested were SHA256 and MD5, across three complexity levels (Level 1, 2, and 3). Varying the algorithm between SHA256 and MD5 did not significantly impact the time taken to crack the passwords. However, altering the character set greatly affected the time. The inclusion of uppercase letters increased the time required to approximately 12 hours, and introducing mixed case, symbols, and numbers made cracking impossible within the 12-hour window. The results mirror what was discovered by Bodily et al. Computing the final password of MD5 and SHA256 took 7 because it has 1 capital letter at the beginning of the word "Jellyman". The Hashes and passwords will be available on github along with any code used.

Algorithm and Level	Passwords Cracked in 12 hrs
SHA256 Level 1	10
SHA256 Level 2	1
SHA256 Level 3	0
MD5 Level 1	10
MD5 Level 2	1
MD5 Level 3	0

Table 1. Passwords cracked within 12 hours across SHA256 and MD5 and levels of complexity.

5. Conclusion

This report investigated and discussed the strengths and limitations of password-based authentication, focusing on the effectiveness of brute-force password attacks using John the Ripper. Through experimentation with MD5 and SHA256 hashing algorithms and varying levels of password complexity, the results showed that the hashing algorithm did not significantly affect the time required to crack passwords. However, password complexity played a crucial role. Simple lowercase passwords were easily cracked, while the inclusion of uppercase letters extended the time to around 12 hours. When mixing case, symbols, and numbers, it became impossible to crack the passwords within the 12-hour limit. This reinforces the importance of strong password policies and highlights that enhancing password complexity is a more effective strategy than simply changing hashing algorithms for improving password security. There is plenty of space for future work to be done on finding just how long it would take John the Ripper to compute passwords using brute force methods on hashes that contain salts and the strongest hashing algorithm ARGON2 (Gupta 2024) with a lengthy time limit to mimic real world conditions

6. Acknowledgements

The template for this report was obtained from LaTeX and the original author of it was Guillermo Jiménez and Eduardo Gracidas. ChatGPT was also used to streamline programming and finding the relevant shell scripting commands. It helped a great deal speed up the coding and shell scripting process and increased coding efficiency for creating the different hashes, find the relevant linux commands and find starting points for relevant literature regarding CHFs. We have also have had to rely on blogs for some of the information as companies will not inform everyone what hashing algorithms they use (among other information) for password storage as this reduces the security of their systems and password storage. All code, Hashes and passwords used in this investigation will be released on github however it has not been included for the sake of anonymity as per university guidelines.

■ References

- (CISA), Cybersecurity Infrastructure Security Agency (2024). *Understanding Brute Force Attacks*. URL: <https://www.cisa.gov/uscert/ncas/tips/ST04-002>.
- Arias, Dan (2021). *Adding Salt to Hashing: A Better Way to Store Passwords*. Auth0 Blog. URL: <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>.
- Authgear (2024). *Password Hashing: How to Pick the Right Hashing Function*. URL: <https://www.authgear.com/post/password-hashing-how-to-pick-the-right-hashing-function#:~:text=In%20a%20collision%20attack%2C%20the,are%20less%20vulnerable%20to%20collision..>
- Clark, Anna (2007). *Passwords and their significance*.
- Gupta, Deepak (2024). *Comparative Analysis of Password Hashing Algorithms: Argon2, bcrypt, scrypt, and PBKDF2*. Security Bloggers Network. URL: <https://securityboulevard.com/2024/07/comparative-analysis-of-password-hashing-algorithms-argon2-bcrypt-scrypt-and-pbkdf2/>.
- Hatice Ozsahan, David Worthington (2023). *2024 Multi-Factor Authentication (MFA) Statistics and Trends to Know*. URL: <https://jumpcloud.com/blog/multi-factor-authentication-statistics>.
- Herley, Cormac, P.C. van Oorschot, and Andrew S. Patrick (2009). “Passwords: If we’re so smart, why are we still using them?” In: *Proceedings of the 2009 Financial Cryptography and Data Security Conference*. Springer, pp. 230–237.
- IBM (2024). *Cost of a Data Breach Report 2024*. URL: <https://www.ibm.com/reports/data-breach>.
- Marchetti, Kaden and Paul Bodily (2022). “John the Ripper: An Examination and Analysis of the Popular Hash Cracking Algorithm”. In: *Proceedings of the Department of Computer Science, Idaho State University*. Available at Idaho State University. Pocatello, ID, USA.
- Martisiute, Laura (2023). *How Much Should a Business Spend on Cybersecurity (Updated for 2023)*. Accessed: 24-October-2024. URL: <https://www.senseon.io/blog/how-much-should-a-business-spend-on-cybersecurity#:~:text=As%20a%20general%20rule%20for,breach%2C%20and%20its%20overall%20budget..>
- Mia (2023). *What is the Best Hashing Algorithm?* URL: <https://itoolkit.co/blog/2023/08/what-is-the-best-hashing-algorithm/>.
- NIST (2009). *CVE-2004-2761 Detail - MD5 Not Collision Resistant*. URL: <https://nvd.nist.gov/vuln/detail/cve-2004-2761>.
- (2017). *Research Results on SHA-1 Collisions*. URL: <https://csrc.nist.gov/news/2017/research-results-on-sha-1-collisions>.
- (2023). *Cryptographic Hash Function*. URL: https://csrc.nist.gov/glossary/term/cryptographic_hash_function.
- Vadhan, Prof. Salil (2013). *Collision-Resistant Hash Functions*. Harvard lecture notes. URL: <https://people.seas.harvard.edu/~salil/cs127/fall13/lec18.pdf>.

```
START Copy code

FUNCTION hash_password(hash_type, password, salt = "somesalt"):
    IF hash_type IS "md5":
        RETURN md5 hash of password
    ELSE IF hash_type IS "sha1":
        RETURN sha1 hash of password
    ELSE IF hash_type IS "sha256":
        RETURN sha256 hash of password
    ELSE IF hash_type IS "sha512":
        RETURN sha512 hash of password
    ELSE IF hash_type IS "bcrypt":
        GENERATE salt and RETURN bcrypt hash of password
    ELSE IF hash_type IS "pbkdf2":
        CREATE derived key using pbkdf2_hmac with sha256 and RETURN its hex value
    ELSE:
        RAISE error for unsupported hash type

FUNCTION main():
    PARSE command-line arguments for:
        - hash_type
        - password_file
        - optional salt for PBKDF2

    TRY:
        OPEN password file and READ passwords

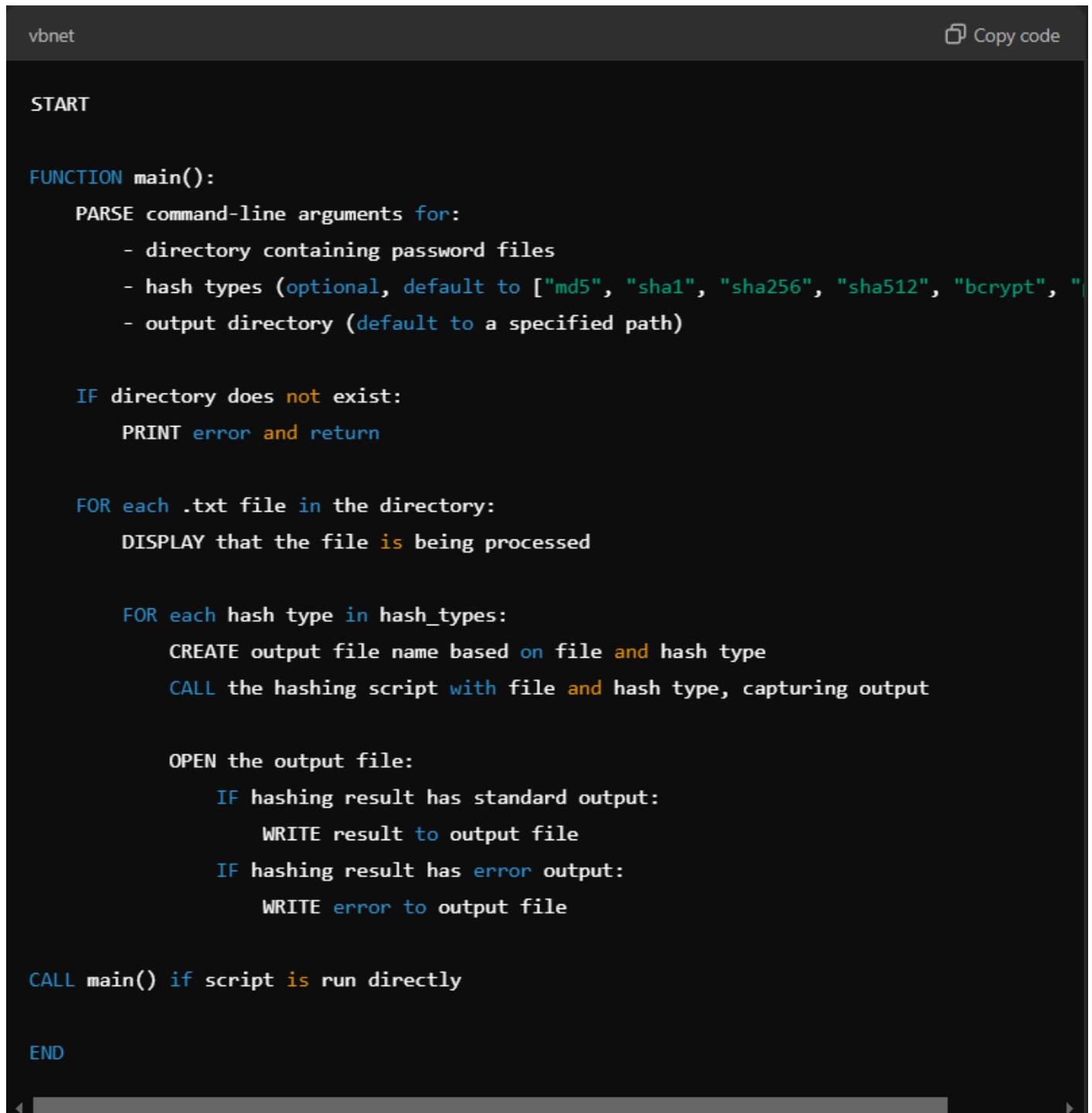
        FOR each password in passwords:
            REMOVE whitespace from password
            CALL hash_password with hash_type, password, and salt (if needed)
            PRINT hashed password

    IF file not found:
        PRINT error and EXIT with status 1

CALL main() if script is run directly

END
```

Figure 1. Python program for hashing a single specified file

A screenshot of a code editor with a dark background. The editor has a tab labeled 'vbnet' at the top left and a 'Copy code' button at the top right. The code is written in a light blue/cyan color. It starts with 'START', followed by a 'FUNCTION main():' block. Inside this function, it 'PARSE command-line arguments for:' with a list of options: '- directory containing password files', '- hash types (optional, default to ["md5", "sha1", "sha256", "sha512", "bcrypt", "])', and '- output directory (default to a specified path)'. Then it checks 'IF directory does not exist:' and 'PRINT error and return'. It then loops 'FOR each .txt file in the directory:' and 'DISPLAY that the file is being processed'. Inside this loop, it loops 'FOR each hash type in hash_types:', 'CREATE output file name based on file and hash type', and 'CALL the hashing script with file and hash type, capturing output'. After the loops, it 'OPEN the output file:', then checks 'IF hashing result has standard output:' and 'WRITE result to output file', and 'IF hashing result has error output:' and 'WRITE error to output file'. Finally, it 'CALL main() if script is run directly' and ends with 'END'.

```
vbnetCopy code

START

FUNCTION main():
    PARSE command-line arguments for:
        - directory containing password files
        - hash types (optional, default to ["md5", "sha1", "sha256", "sha512", "bcrypt", "])
        - output directory (default to a specified path)

    IF directory does not exist:
        PRINT error and return

    FOR each .txt file in the directory:
        DISPLAY that the file is being processed

        FOR each hash type in hash_types:
            CREATE output file name based on file and hash type
            CALL the hashing script with file and hash type, capturing output

        OPEN the output file:
            IF hashing result has standard output:
                WRITE result to output file
            IF hashing result has error output:
                WRITE error to output file

    CALL main() if script is run directly

END
```

Figure 2. Python program for all files in a given directory


```

john --incremental --fork=3 --format="Raw-MD5" ~/Desktop/Hashed_Passwords/Level1/level1_md5_output.txt
Using default input encoding: UTF-8
Loaded 9 password hashes with no different salts (Raw-MD5 [MD5 256/256 AVX2 8x3])
Node numbers 1-3 of 3 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
chicken      (?)
password     (?)
sunshine     (?)
computer     (?)
3 2g 0:00:7 0.2853g/s 23860Kp/s 23860Kc/s 167132Kc/s anik2182..anik2473
2 0g 0:00:00:07 0g/s 24268Kp/s 24268Kc/s 218413Kc/s gort1o8..gorroba
1 2g 0:00:00:07 0.2849g/s 23407Kp/s 23407Kc/s 163879Kc/s bhebh1a..bheatr2b
2 0g 0:00:00:15 0g/s 27113Kp/s 27113Kc/s 244017Kc/s d33j0y..d33ss5
3 2g 0:00:00:15 0.1332g/s 26871Kp/s 26871Kc/s 188149Kc/s m16e429..m1stpuf
1 2g 0:00:00:15 0.1331g/s 27653Kp/s 27653Kc/s 193589Kc/s jewd2tc..jewhiuj
2 0g 0:00:00:18 0g/s 28028Kp/s 28028Kc/s 253655Kc/s t2l5e1..t2lcoe
3 2g 0:00:00:18 0.1105g/s 27805Kp/s 27805Kc/s 194683Kc/s cbw8hg2..cbwar87
1 2g 0:00:00:18 0.1104g/s 28470Kp/s 28470Kc/s 199307Kc/s tabr4104..tabr475j
3 2g 0:00:00:20 0.09995g/s 28330Kp/s 28330Kc/s 198353Kc/s twhitall..twhited7
2 0g 0:00:00:20 0g/s 28374Kp/s 28374Kc/s 255623Kc/s cuttynald..cuttiship
1 2g 0:00:00:20 0.09990g/s 28871Kp/s 28871Kc/s 202110Kc/s 1gRRH9..1gRJME
2 0g 0:00:00:22 0g/s 28603Kp/s 28603Kc/s 257661Kc/s r4fndz..r4fm1m
3 2g 0:00:00:22 0.09086g/s 28797Kp/s 28797Kc/s 201707Kc/s figwty!..figwup.
1 2g 0:00:00:22 0.09082g/s 29257Kp/s 29257Kc/s 204994Kc/s lh0130l..lh01080
keyboard     (?)
3 3g 0:00:00:23 0.1303g/s 28999Kp/s 28999Kc/s 202497Kc/s khinnny9..khinnnd08
2 0g 0:00:00:23 0g/s 28786Kp/s 28786Kc/s 259306Kc/s kkbid5$..kkb77*6
1 2g 0:00:00:23 0.08688g/s 29309Kp/s 29309Kc/s 205350Kc/s lpxrlg..lpxnx-
3 3g 0:00:00:24 0.1249g/s 29012Kp/s 29012Kc/s 201387Kc/s kuxp1=..kuxz_k
1 2g 0:00:00:24 0.08326g/s 29053Kp/s 29053Kc/s 203553Kc/s 0fgjjz..0fsrec
2 0g 0:00:00:24 0g/s 28952Kp/s 28952Kc/s 260791Kc/s l1i33347..l1i3308*
3 3g 0:00:00:25 0.1195g/s 28953Kp/s 28953Kc/s 200568Kc/s prtartome2..prtaren10
2 0g 0:00:00:25 0g/s 29137Kp/s 29137Kc/s 263389Kc/s euli123..eullite
1 2g 0:00:00:25 0.07964g/s 28785Kp/s 28785Kc/s 202391Kc/s sitro10!..sitruzo0
2 0g 0:00:00:34 0g/s 30086Kp/s 30086Kc/s 271177Kc/s gt4m.$A..gt4m8ts
3 3g 0:00:00:34 0.08813g/s 30012Kp/s 30012Kc/s 199516Kc/s cjhg43aj..cjhg4242
1 2g 0:00:00:34 0.05873g/s 28578Kp/s 28578Kc/s 200350Kc/s lj0hm6..ljm65v
football     (?)
notebook     (?)
3 3g 0:00:06:30 0.007673g/s 30915Kp/s 30915Kc/s 132813Kc/s 889779c..88977ly
2 0g 0:00:06:30 0g/s 31922Kp/s 31922Kc/s 162716Kc/s nh294pii..nh296sha
1 4g 0:00:06:30 0.01023g/s 31800Kp/s 31800Kc/s 119651Kc/s antandranno..antandria21
2 0g 0:00:06:33 0g/s 31852Kp/s 31852Kc/s 161997Kc/s ileprick2..ileprio29
3 3g 0:00:06:33 0.007632g/s 30858Kp/s 30858Kc/s 132338Kc/s 89csep..89csep9
1 4g 0:00:06:33 0.01017g/s 31694Kp/s 31694Kc/s 119011Kc/s assourligee..assourracoh
2 0g 0:00:07:23 0g/s 32013Kp/s 32013Kc/s 153333Kc/s hathum5%..hath26pk
3 3g 0:00:07:23 0.006771g/s 31209Kp/s 31209Kc/s 128902Kc/s ntatmolm..ntatmiou
1 4g 0:00:07:23 0.009028g/s 31910Kp/s 31910Kc/s 113166Kc/s 00jpahmj..00jprock
3 3g 0:00:07:25 0.006740g/s 31226Kp/s 31226Kc/s 128795Kc/s cuew65l..cuew6ci
1 4g 0:00:07:25 0.008987g/s 31922Kp/s 31922Kc/s 112969Kc/s fllsp164..fllspmcb
2 0g 0:00:07:25 0g/s 32021Kp/s 32021Kc/s 152947Kc/s nn1ckm61..nn1cabdu
elephant     (?)
elevened     (?)
2 2g 0:00:09:40 DONE (2024-10-23 09:31) 0.003448g/s 31584Kp/s 31584Kc/s 131377Kc/s elevem8*..eleven99
3 3g 0:00:10:00 DONE (2024-10-23 09:32) 0.004999g/s 31060Kp/s 31060Kc/s 113221Kc/s 0ek7ltz..0ek7pc$
1 4g 0:00:10:00 DONE (2024-10-23 09:32) 0.006666g/s 31848Kp/s 31848Kc/s 100130Kc/s cplioa145..cpliuined

```

Figure 3. Exaple Console output when running John for 10 hours on level 1 for MD5