COS330 Prac 1
John-Peter Krause
u23533529

Task 1

I used the following hashing algorithms: bcrypt, md5 and sha256. I chose them mainly for simplicity; they are included by default in the bcrypt and hashlib python3 library, however there are still some other factors. Sha256 is very fast to hash, making brute force attack more easy to demostrate, md5 has no salting or key stretching which means rainbow tables exist for most password hashes and hashing is even quicker than sha256, bcrypt is a very good hash, has salting and is very slow by design which makes it good against brute force attacks. I hypothesize that the bcrypt hashes will be the most difficult...

This is how the database looks after running task1.py and the task1.sql query to add those columns.

SQL ▾ 1 / 1 1 - 50 of 50

id	username	password	md5	sha256	bcrypt
1	david.davis50	gIMrC	6ea8939892ec9a106627584ee83a6ef3	9498a35664795512ced8cd78efc5dd3a2a55a69d2bac3fd498c325d2db0f26f6	$2b$12$FB4Dn.xiv5v9ed.gLVcSy.Nu.kqeIE2W4HGxqKemp/V
2	sara.smith47	passwor	f88d9aaa52161c2e4f8c42c0d86aa05d	2a2f822ad5319117db5016955c0d89d98a59e50866789cf8883fc6abdd2a1e17	$2b$12$OuBXuDoIou9VUMC6DuAUeOiij8MK5yViWJ4kADhraM/
3	john.davis94	srdu	ea426ec1bf87a552a7e1237b3cc2cb62	315a83e8b2de01e04a36a3aebd3482d773b900344fa5365262b4d185fa63c5ef	$2b$12$e0/pcRe.Hs4YvFmAVvGE7ubHgqEGkrcTmbDhoNctbZ6
4	alex.davis54	enjIlO	001b5875b4a324cd81b0007f3fa45f95	57beed2eff7d83d58b24049422ce8044859ee662b4e7f383a0fffbcfd3cf89b4	$2b$12$CYbVk5TXPTuk7YkoJwt2EOkn7InXrWe9pi9fy7bH70G
5	john.davis6	IiSG	d715bedfb0c7c21e8cfeade4eca5e0a7	c411d879ff0fe4ef2322c588a71c44bf3dbfc753fa456da85915782cce4ab824	$2b$12$mwn7H2GhO03WKka5/KXHqeWVxp/QESYC1tSAkaaSHew

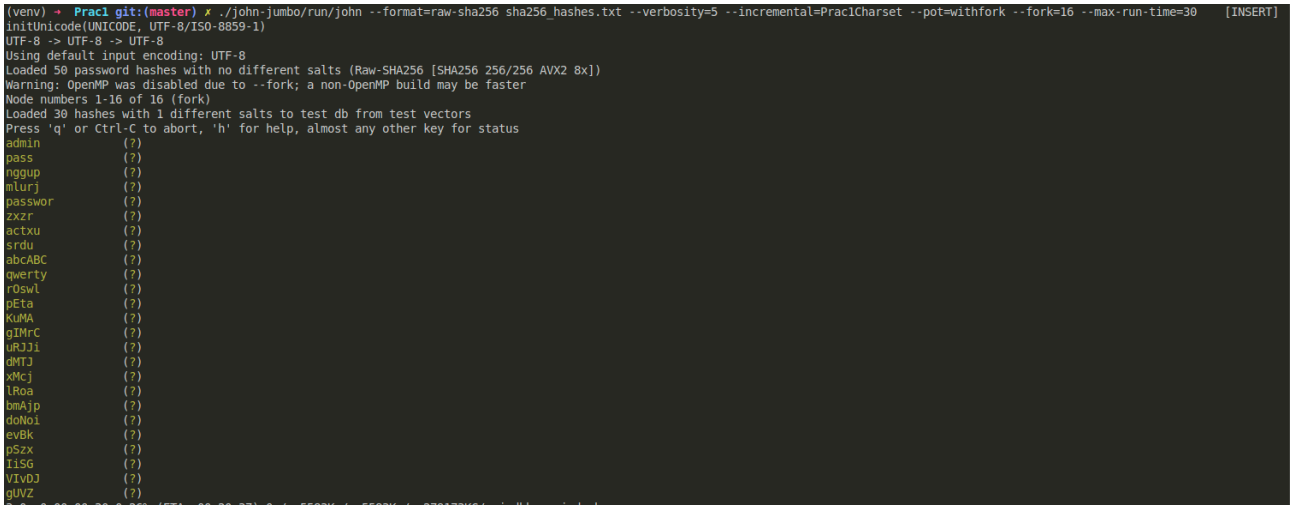
Task 2

I realised that I could significantly improve performance by limiting the search space to password that only included alpha characters(alphabet letters and their uppercase letters). So I created a charset from the following config:

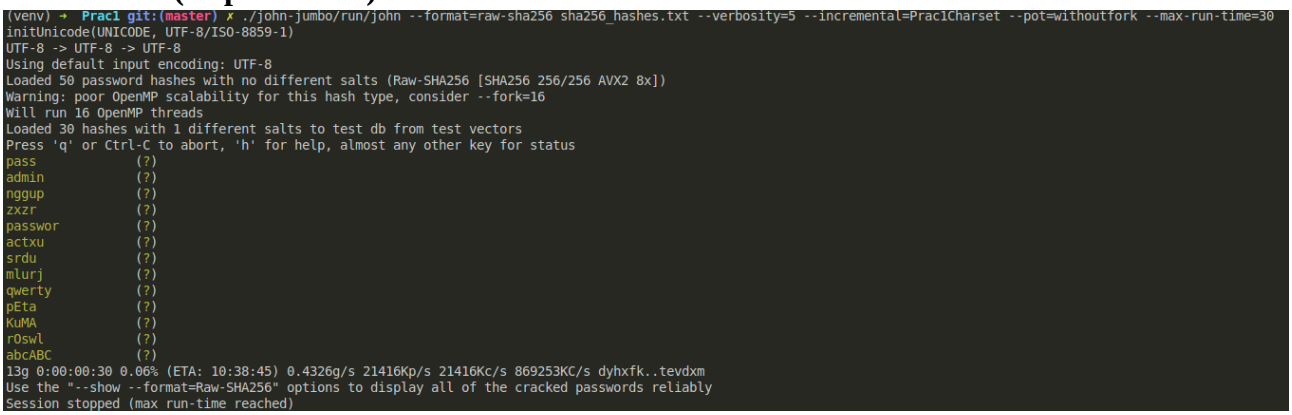
[Incremental:AlphaOnlyMax7]
File = $JOHN/alpha.chr
MinLen = 4
MaxLen = 7
CharCount = 52

and generated with **./john-jumbo/run/john –make-charset=alpha.chr**
I could also improve performance by using all 16 of my cpu cores with –**fork=16**

I tried running with and without fork=16:
WITH(25 passwords):



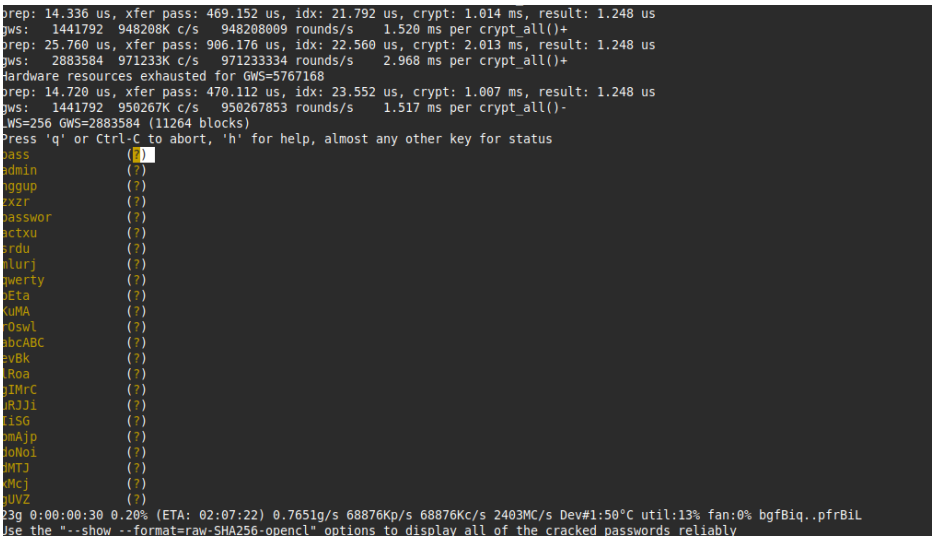
WITHOUT(13 passwords):



Enabling fork is better!

But I have a gpu:

I installed and setup opencl for gpu processing.

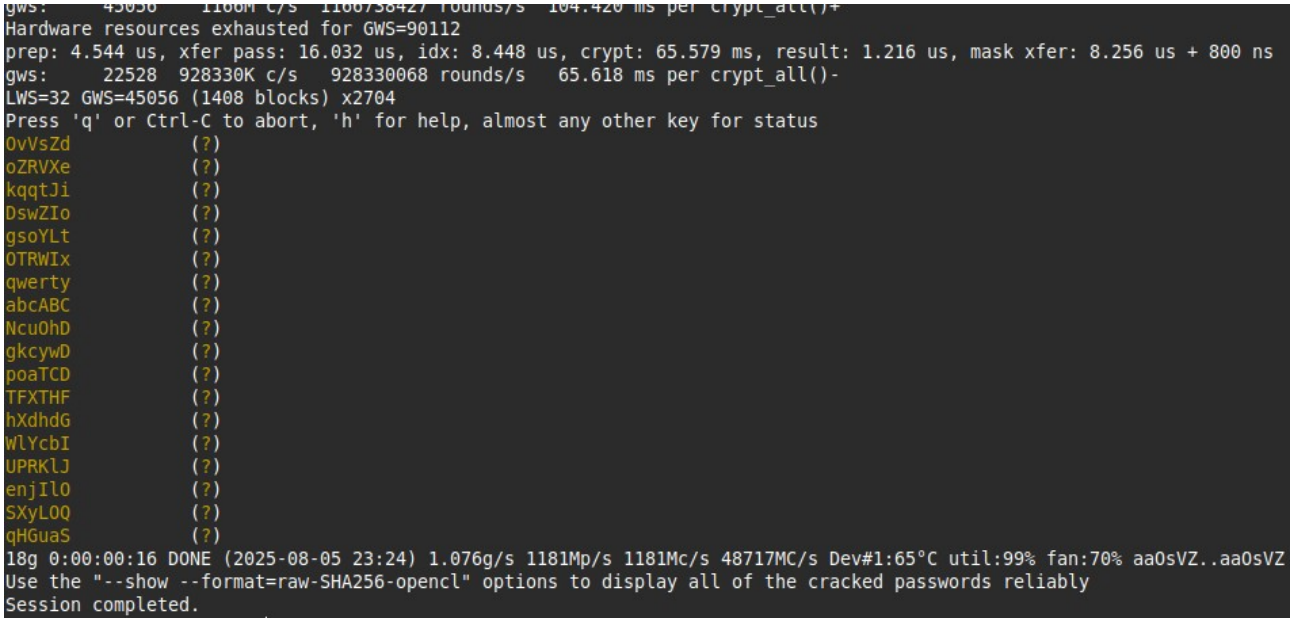


It was seemingly worse, but it was being throttled by io(reading from the custom charset file). Instead I decided to use the mask setting to improve performance by reducing bottleneck from IO. The drawback from this is, that I have to run several commands for each length password from 4 up to 7(I saw this when looking at the passwords in the sqlite db). Thus I will only show one screenshot of a certain length password per hashing algorithm, else the report would get too convoluted.

SHA256

The results were basically instant, except for length 6 which took a few seconds and length 7 which took about 9 minutes. So about 10 minutes in total for all 50 passwords.

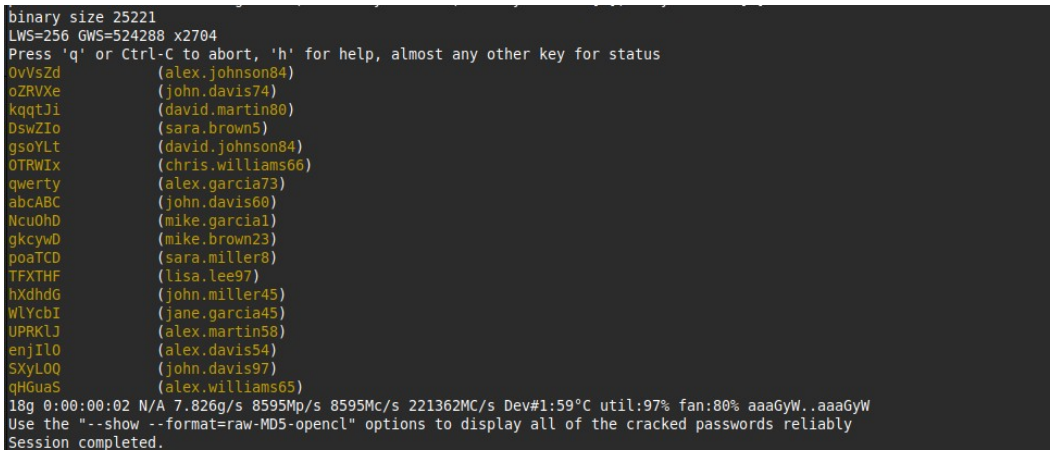
Example of command I ran with mask that matches against alphanumeric password with 6 characters
./john-jumbo/run/john sha256_hashes.txt --verbosity=5 --mask='[a-zA-Z][a-zA-Z][a-zA-Z][a-zA-Z][a-zA-Z][a-zA-Z]' --pot=withgputestandmask –format=raw-SHA256-opencl



Took 16 seconds to find all 18 passwords with length:6

MD5

Md5 was faster at about 2 minutes for all 50 passwords:



BCRYPT

For bcrypt I decided to use my CPU instead of my GPU, because bcrypt is designed to limit the GPU advantage since it is memory hard and not massively parallelizable. My 16-core CPU is likely more efficient.

I ran bcrypt for over 30 minutes for a 4 character password and didn't find a single password, this makes sense as Bcrypt is designed to be slow and resistant to bruteforce attacks. John gave an estimate that it would take about a months time, so for all the passwords it would probably take years...

Task 3

The strongest algorithm was bcrypt, but that already uses a bunch of very secure techniques – so I couldn't really improve on that. I decided to use the second best SHA256, I added salting and encrypted the salts with AES-256-GCM. The pepper is a server-only secret(it's defined in the python3 script, but in production it would be loaded from a file with secure permissions). Salting and pepper combined protect against offline dictionary attacks and rainbow tables.
I'm used a nonce as well to encrypt salt to protect from replay attacks.

To log in a user simply calls the login_user function with username and password, it returns True or False depending on if a user exists with that username and if the password matches the hash.

Testing with bob with password: password123. Validates correctly!