Akka Assignment 3
Password Cracking

Sascha Obst, Johannes Hötter
Hasso Plattner Institut
Digital Engineering • Universität Potsdam

22.11.2019

The task

Our task has been to crack passwords composed of a given character universe with a fixed size while getting some hints about the password structure. The password are SHA256 encrypted.

ID	Name	PasswordChars	PasswordLength	Password	Hint1	Hint2	Hint3	Hint4	Hint5	Hint6	Hint7	Hint8	Hint9
1	Sophia	ABCDEFGHIJK	10	c4712866799881	1582824a01	e91aca467f5	52be0093f91	8052d9420a	ca70f765d8c	570d3ada41	f224061bd03	01d8adcfdb7	4b47ac115f6
2	Jackson	ABCDEFGHIJK	10	c178ef3bd2dbf4e	7624e76e72l	834d255d02	b2e939a89b	0f0c2aefcfcf	d22b589632	0066eb98a0	21b5a6f0b9c	49dc000595	0598a2e796
3	Olivia	ABCDEFGHIJK	10	b6dffc272ed2907	69d7ba29ae	2228c851d7	5ea85a209e	e234a05f095	0dd9e26059	f70853fec1c	b0f110e28c9	6cb71f73643	c0053593800
4	Liam	ABCDEFGHIJK	10	109fc23b13e8b4	54dfc60ba18	49042e40c5	282e17962a	75e17b6b7a	90d6920945	b857b99db7	503d344872	c0a6e3136d	9a632dd1734
5	Emma	ABCDEFGHIJK	10	6078335cde9e5e	9df27520532	b83ec9780d	d4e74d9a58	486b829cff9	552ba27c5a	60b64d370b	b7fdd9f77b9	421e96f2d84	380d0b66e33
6	Noah	ABCDEFGHIJK	10	6d42b3f273438d	d4ffe366df9	969a6d0091	31da9b1669	bc351506d4	a1601cb736	62ecbbd806	a1ba7bb71el	2db32b5e66	00d13238a8
7	Ava	ABCDEFGHIJK	10	4121[illegible]55971f	e00595b2ca	90d6247cbef	72c5245735	7ff44950404	628dfdd46cd	c901b55923	b8209fa626	3fe10f1ee05	47452b515e
8	Aiden	ABCDEFGHIJK	10	fbe3[illegible]50f71d7	de2617fb757	06bb6d175e	03ee78244a	87316b71fbf	9aab84d045	87a65ceb83	589c35f4024	ed5778b3ed	245e0af4e3c
9	Isabella	ABCDEFGHIJK	10	5a2[illegible]f6c85	23d6de9da4	7af3c5c070a	824137665f5	c71deb0e1e	49535ddb45	9271a854a0	6b2b2dbf84	896fad1ee1	9cad6a839e0

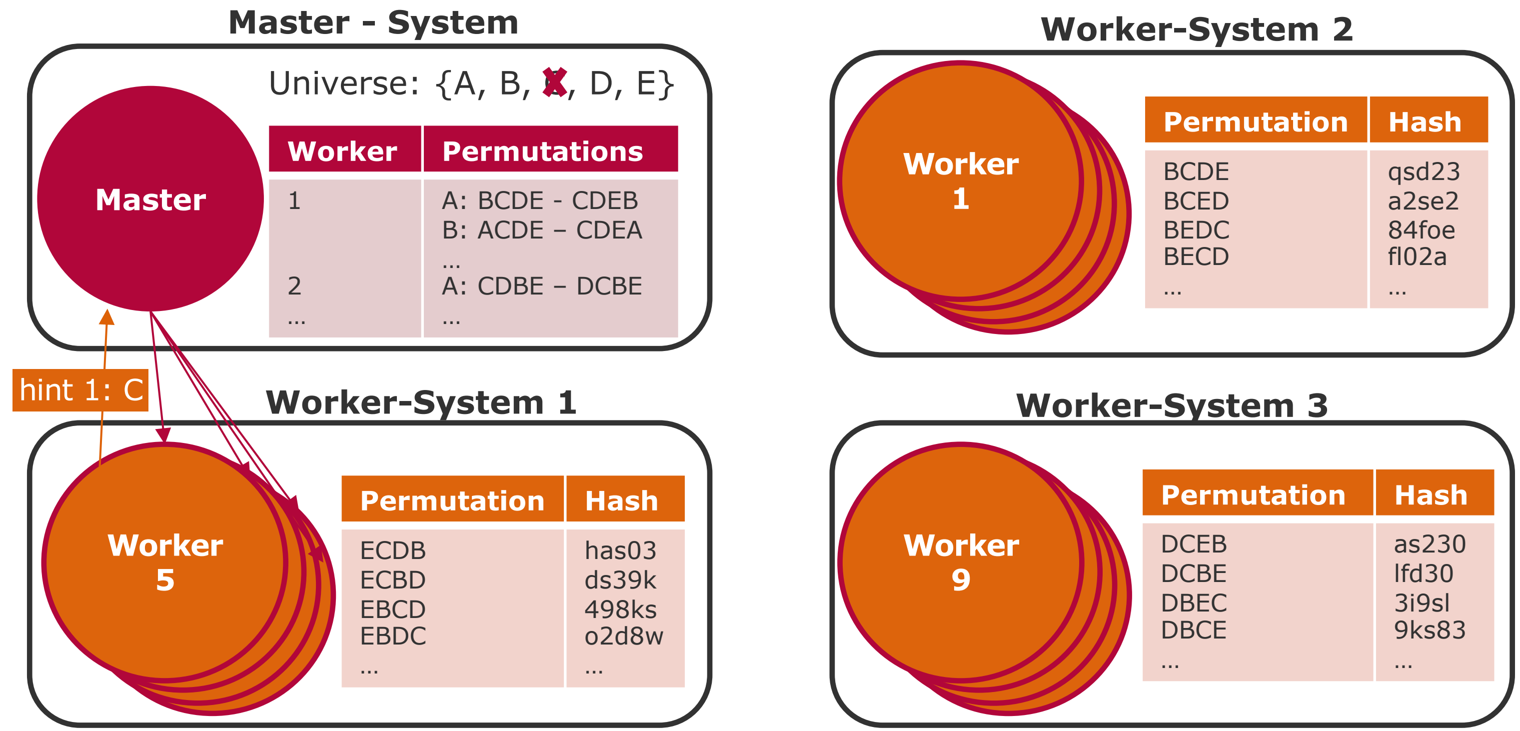
ID	Name	PasswordChars	PasswordLength	Password	Hint1	Hint2	Hint3	Hint4	Hint5	Hint6	Hint7	Hint8	Hint9
1	Sophia	ABCDEFGHIJK	10	GGGFGFFFFG	HJKGDEFBIC	FCJADEKGHI	FAJBDIEKGH	AGCJEHFKIB	BHKICGFADJ	JIFAGKDBCE	GAHDKJBCEF	EBIKHGDAFC	DJHAFGICBE
2	Jackson	ABCDEFGHIJK	10	EFFFEFFEEE	JBKIGEFHDC	AEHJIDGFKC	IDAHFGEKBJ	EHFIJKBGAC	HFJIEDACBK	FGKIDJCEAB	KDHGCAEJFB	FKDAHCIGBE	CDEIJFBHAG
3	Olivia	ABCDEFGHIJK	10	KDDDKDKDKD	DECJKBFIHG	CAKEIFHGJD	JBFEDHIKAG	IDAKGHBFJC	KGBAEICHDJ	DKHFBEJIAC	EABJGFIKDC	JCFBADGHKE	BCKGEDFHAI
4	Liam	ABCDEFGHIJK	10	CCCCGGCCGG	CFHDBJKGEI	FAICGJDHEK	CHBKIGEJAF	AICDKGHJBF	EDAGKBJHIC	JDKIFACEGB	BGKJDAHCFE	KDIEACGBFH	CHFDGABIEJ
5	Emma	ABCDEFGHIJK	10	BDDBDDDDDB	EGICDFKHBJ	HEAJIBDGFK	BAHCKDFIJG	HBEDKAGCIJ	IBHCEFJADK	FAGDEJICKB	GFHEAKCDBJ	HBFACIKGDE	FAEDCJIGHB
6	Noah	ABCDEFGHIJK	10	GHGGHGGHHH	CFKBJGDIEH	CAIGHEJFDK	GJBEKIADFH	AIBCJHGEKF	GDIBCKFHAJ	CGJHDEAIBK	DGKFBEACJH	AGDCFIBEKH	DEGJCFABHI
7	Ava	ABCDEFGHIJK	10	DFFFDFDFDD	FHIKEBGDJC	KHFICAJGED	KIAHDFEJGB	CGFAKIBDHJ	ACEHFKBIDJ	GBADIJEKFC	AFCKGHBDJE	HFDAKBCEGI	IJHGFCEBAD
8	Aiden	ABCDEFGHIJK	10	IIIHHHHIHI	GCIFEHDKBJ	JDHIEGKACF	FJHBEGAKDI	AIBJEHKGFC	CGJAFBIHDK	JECAIDGHBK	IJBDCKEAFH	DEKIAGHFCB	EGAICBJHFD
9	Isabella	ABCDEFGHIJK	10	CCJCJCJCCJ	EHDCGIKBJF	IFJCAEHGKD	AFBHEGKIJC	KGFBIADJHC	JKAGEDHIBC	CBKIDEAHFJ	CJAFKEIBDG	CDKJEHGFAB	ADHFCGJIEB

Our approach - methodology

To solve the given problem, we decided to try the following methodology:

1. (Master) calculate a map once which holds instructions for each worker on which range of permutations it should try to crack hints. For every hint, send these instructions to the corresponding worker.
2. (Worker) calculate all hashes for every permutation in the given range of the instruction map and cache them – this is only applied if there is no cache entry
3. (Worker) run through the given range of permutations and look up the hashed value of the current permutation. If the worker found the solution for the hint, tell the other workers (using a simple class variable) to stop their search. The solution is send to the master.
4. (Master) If all hints are cracked, remove the characters which have been found through the hint messages from the password universe and crack the password using brute force.

Our approach - overview



arcs to systems 2 and 3 are not drawn for simplicity's sake

Calculating the permutation ranges

We wanted to calculate a map which tells each worker on which range of permutations it should operate on (e.g. worker 1 from „BCDE" to „CDEB" for key „A").

Worker	Permutations
1	A: BCDE - CDEB B: ACDE – CDEA …
2	A: CDBE – DCBE
…	…

To do so, we calculated the number of possible permutations for a given length as n = (|password| - 1)!

We then gave each worker a start and end index by defining how many permutations each must calculate in order to get every combination. From these indices and our custom function to find the next permutation given a string, we calculated the ranges. The number of ranges a worker has is (|password| - 2)! x characters_in_universe.

Creating a lookup structure for hashed values

We tested, how long computing a hash takes: 1000 operations cost roughly 50 milliseconds. Therefore, we decided to compute those values once, caching them for lookups afterwards. 1000 lookups cost roughly 2 milliseconds, leading to a potential speed-up factor of 25.

„BCDE“ → SHA256* → „qsd23“

Permutation	Hash
BCDE	qsd23
BCED	a2se2
BEDC	84foe
BECD	fl02a
…	…

* Not the actual SHA256 value for BCDE

Cracking the hints

Once the map is build and the cache has been set up (which happens during the processing of the first message), the hints can easily be cracked. Each worker propagates through its instruction map, i.e. start at one range definition (such as BCDE – CEBD) and looks up every hash value for the current permutation. If the hashed value equals the hint, the missing word can be extracted as the solution for the hint. This is especially easy, as the keys for each range are the characters which are actually missing, (i.e. BCDE – CEBD is accessed in our map through the key ‚A‘).

Once the hint is cracked, the worker tells all the other actors to stop searching (by setting a class variable which is checked at every permutation) and sends the solution to the master. The master can use this solution to prune the search space for the actual password. After this, the master sends the next hint or starts with the password cracking (given that all hints are solved).

Solving the puzzle

Once all hints are solved, the brute force search for the actual password is started*. As the universe of possible characters has been reduced before by the workers, this task has become much easier than at the beginning.

We generate each possible combination (with n = |remaining_chars_in_universe|**|length_of_password|), and hash these combinations to check whether they equal the encrypted password. Once we find the right candidate, we‵re done with the given password and can continue to the next one until we‵re completely finished.

* We also thought about beginning with the brute force password cracking as soon as the first message is sent, but we decided to implement this in a later version.

Our key learnings (1)

When we designed our solution, we were really confident that our solution would have the potential to compete against the other solutions, – our design was completely scalable independent of the number of hints and it cached complex calculations to save runtime. However, in reality we were confronted with the following problems:

- One worker dominated the other n-1 ones, as it was heavily faster. Thus, counterintuively our approach is faster if the number of workers is set to 1, as in this case this worker propagates through all ranges in a steady pace – with more workers, the ranges would be propagated in very different speeds, making it more likely possible to find the solution in a really slowly propagated range
- Actors crashed during the first time-consuming calculations (hashing and storing the hashes in the cache), as they apparently didn‘t respond to heartbeat messages

Our key learnings (2)

When we designed our solution, we were really confident that our solution would have the potential to compete against the other solutions, – our design was completely scalable independent of the number of hints and it cached complex calculations to save runtime. However, in reality we were confronted with the following problems:

- We heavily messed up our own implementation of the permutation calculation as we developed it as an inplace method, leading to many hours of debugging
- We built too complex data structures, which may could have been simplified: Map<List<Map<Character, Character[][]>>> as our main dictionary to keep calculation ranges for the workers

Our key learnings (3)

When we designed our solution, we were really confident that our solution would have the potential to compete against the other solutions, – our design was completely scalable independent of the number of hints and it cached complex calculations to save runtime. However, in reality we were confronted with the following problems:

- We should have built the easy solution first!! Adding optimizations incrementally would have been the better approach.
- The D-Space is a really good place to get creative and figure out cool solution designs (we actually came up with 2 to 3 possible designs)
- …and Akka can be really fun ;-)

Some statistics… ;-)

Some other measures to quantify our learnings for this task:

- Hours spent: ~ 20 (from Saturday to Friday)
- „That shoud definitely be the last problem/bug"-count: over 20
- Kicked out of D-space/main building as the building was closed: 3 times
- Whiteboards filled with solution approaches and calculations: over 10
- Chocolates eaten out of frustration: too many
- Jubilation whenever a hint/password was cracked: a lot ☺