Lecture 23 — Password Cracking, Reduced-Resource Computation

Patrick Lam & Jeff Zarnett

patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

January 4, 2019

scrypt is the algorithm behind DogeCoin.

The reference:

Colin Percival, "Stronger Key Derivation via Sequential Memory-Hard Functions".
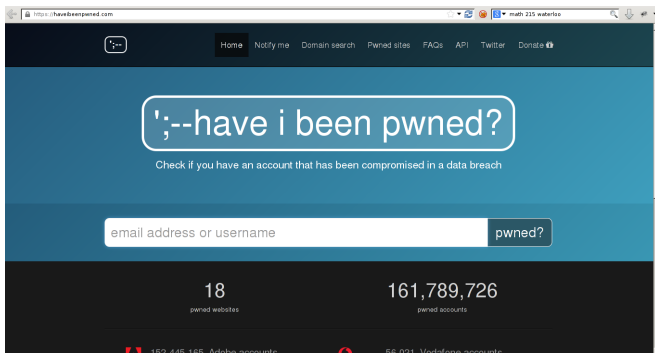Presented at BSDCan'09, May 2009.

`http://www.tarsnap.com/scrypt.html`

- **not** plaintext!
- hashed and salted

One-way function:

- $x \mapsto f(x)$ easy to compute; but
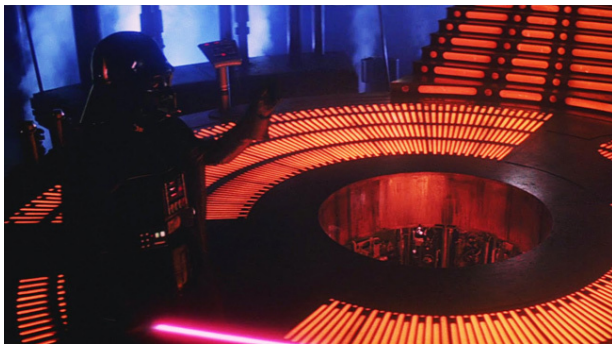- $f(x) \stackrel{?}{\mapsto} x$ hard to reverse.

Examples: SHA1, Scrypt.

Perhaps passwords have already been leaked!

';--have i been pwned?

Check if you have an account that has been compromised in a data breach

`email address or username`   **pwned?**

18
pwned websites

161,789,726
pwned accounts

The first thing is to try really common passwords.

You just might get a hit!

"All too easy."

How can we reverse the hash function?

- Brute force.

GPUs (or custom hardware) are good at that!
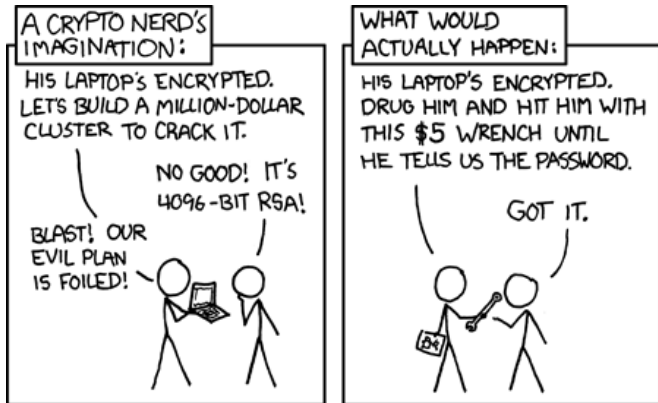
The Arms Race: Making Cracking Difficult

Historically: force repeated iterations of hashing.

Main idea behind scrypt (hence DogeCoin):

- make hashing expensive in time and space.

Implication: require more circuitry to break passwords.
Increases both # of operations and cost of brute-forcing.

Of course, there's always this form of cracking:

(Source: xkcd 538)

Formalizing "expensive in time and space"

Definition

A memory-hard algorithm on a Random Access Machine is an algorithm which uses $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{1-\varepsilon})$.

Such algorithms are expensive to implement in either hardware or software.

Next, add a quantifier:
move from particular algorithms to underlying functions.

A *sequential memory-hard function* is one where:

- the fastest sequential algorithm is memory-hard; and
- it is impossible for a parallel algorithm to asymptotically achieve lower cost.

Exhibit. ReMix is a concrete example of a sequential memory hard function.

The paper concludes with an example of a more realistic (cache-aware) model and a function in that context, BlockMix.
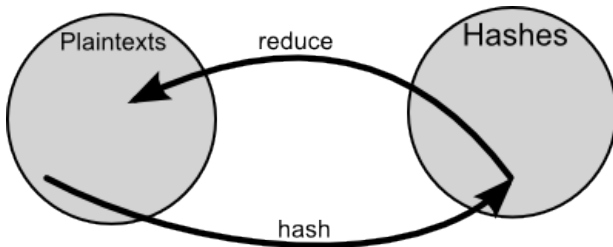
If designed well, hash functions aren't easy to brute force.
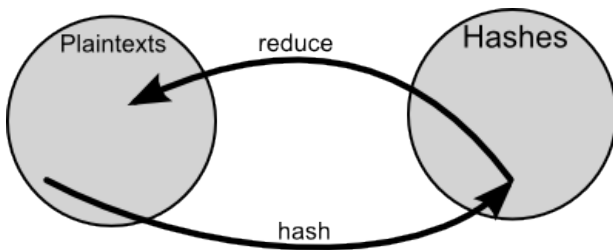
What if we remembered some previous work?

It isn't practical, or possible, to store the hashes for every possible plaintext.

The rainbow table is a compromise between speed and space.

The "reduction" function maps hashes to plaintext:

Example: 123456 $\rightarrow$ d41d8cd98f00b204e9800998ecf8427e $\rightarrow$ 418980

We should do this to develop some number of chains.

This is the sort of task you could do with a GPU, because they can do a reduction relatively efficiently.

Once we have those developed for a specific input set and hash function, they can be re-used forever.

You do not even need to make them yourself anymore (if you don't want) because you can download them on the internet... they are not hard to find.

They are large, yes, but in the 25 - 900 GB range.

I mean, Fallout 76 had a day one patch of 52 GB, and it was a disaster of a game.

1. Look for the hash in the list of final hashes; if there, we are done
2. If it's not there, reduce the hash into another plaintext and hash the new plaintext
3. Go back to step 1
4. If the hash matches a final hash, the chain with the match contains the original hash
5. Having identified the correct chain, we can start at the beginning of the chain with the starting plaintext and hash, check to see if we are successful (if so, we are done); if not, reduce and try the next plaintext.

Like generation, checking the tables can also be done efficiently by the GPU.

Some numbers from
`http://www.cryptohaze.com/gpurainbowcracker.php.html`:

- Table generation on a GTX295 core for MD5 proceeds at around 430M links/sec.
- Cracking a password 'K#n&r4Z': real: 1m51.962s, user: 1m4.740s. sys: 0m15.320s

Yikes.

A common physics problem that programmers are asked to simulate is the N-Body problem.
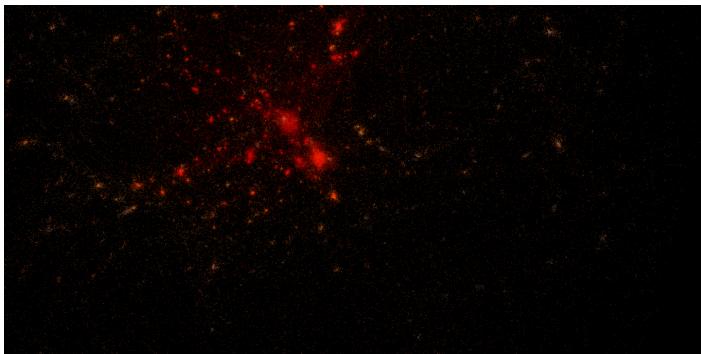

Image Credit: Michael L. Umbricht

Let's assume it's OpenCL converted and is optimized.

Can we use `float` instead of `double`?

What if we want more?

Points that are far away contribute only very small forces.

So you can estimate them (crudely).

The idea is to divide the points into a number of "bins" which are cubes representing a locale of some sort.

Then, compute the centre of mass for each bin.

When calculating the forces: centre of mass for faraway bins; individual particles for nearby particles.

A more concrete explanation with an example: suppose the space is divided into $[0, 1000]^3$, so we can take bins which are cubes of length 100.

This gives 1000 bins.

To increase the accuracy, what should we do?

To increase the speed, what should we do?

Compute all of the masses in parallel: create one thread per bin, and add a point's position if it belongs to the bin:

```
int xbin, ybin, zbin; // initialize with bin coordinates
int b;
for (i = 0; i < POINTS; i++) {
    if (pts[i] in bin coordinates) {
        cm[b].x += pts[i].x; // y, z too
        cm[b].w += 1.0f;
    }
}
cm[b].x /= cm[b].w; // etc
```

Note that this parallelizes with the number of bins.

For the next step, the program needs to keep track of the points in each bin.

In a second phase, iterate over all bins again, this time putting coordinates into the proper element of `binPts`.

The payoff from all these calculations is to save time while calculating forces.

In this example, we'll compute exact forces for the points in the same bin and the directly-adjacent bins in each direction

That makes 27 bins in all, with 6 bins sharing a square, 12 bins sharing an edge, and 8 bins sharing a point with the centre bin).

If there is no adjacent bin (i.e., this is an edge), just act as if there are no points in the place where the nonexistent bin would be.

Necessarily, writing the program like this is going to mean more than one kernel.

This does mean there is overhead for each kernel, meaning the total amount of overhead goes up.

Is it worth it?

With 500*64 points:

- OpenCL, no approximations (1 kernel): 0.182s
- OpenCL, with approximations (3 kernels): 0.168s

With 5000*64 points:

- OpenCL, no approximations (1 kernel): 6.131s
- OpenCL, with approximations (3 kernels): 3.506s

Consider Monte Carlo integration.
It illustrates a general tradeoff: accuracy vs performance.
Martin Rinard generalized the accuracy vs performance tradeoff with:

- early phase termination [OOPSLA07]
- loop perforation [CSAIL TR 2009]

We've seen barriers before.
No thread may proceed past a barrier until all of the threads reach the barrier.

This may slow down the program: maybe one of the threads is horribly slow.

Solution: kill the slowest thread.

"Oh no, that's going to change the meaning of the program!"

OK, so we don't want to be completely crazy.

Instead:

- develop a statistical model of the program behaviour.
- only kill tasks that don't introduce unacceptable distortions.

When we run the program:
    get the output, plus a confidence interval.

Monte Carlo simulators:
Raytracers:

■ already picking points randomly.

In both cases: spawn a lot of threads.

Could wait for all threads to complete;
or just compensate for missing data points,
assuming they look like points you did compute.

In scientific computations:

- using points that were measured (subject to error);
- computing using machine numbers (also with error).

Computers are only providing simulations, not ground truth.

Actual question: is the simulation is good enough?

Like early-phase termination, but for sequential programs:
   throw away data that's not actually useful.

```
for (i = 0; i < n; ++i) sum += numbers[i];
```

$$\Downarrow$$

```
for (i = 0; i < n; i += 2) sum += numbers[i];
sum *= 2;
```

This gives a speedup of ~ 2 if numbers[] is nice.

Works for video encoding: can't observe difference.

Applications of Reduced Resource Computation

Loop perforation works for:

- evaluating forces on water molecules (summing numbers);
- Monte-Carlo simulation of swaption pricing;
- video encoding.

More on the video encoding example:
Changing loop increments from 4 to 8 gives:

- speedup of 1.67;
- signal-to-noise ratio decrease of 0.87%;
- bitrate increase of 18.47%;
- visually indistinguishable results.

```
min = DBL_MAX;
index = 0;
for (i = 0; i < m; i++) {
  sum = 0;
  for (j = 0; j < n; j++) sum += numbers[i][j];
  if (min < sum) {
    min = sum;
    index = i;
  }
}
```

The optimization changes the loop increments and then compensates.