

Lecture 23 — Password Cracking, Reduced-Resource Computation

Patrick Lam & Jeff Zarnett

2018-12-15

GPU Application: Password Cracking

GPUs are good—too good, even—at password cracking. We'll discuss a paper that proposes a technique to make it harder to crack passwords. This technique is *script*, the algorithm behind DogeCoin [Per09]. See also <http://www.tarsnap.com/script.html>

First, let's talk about acceptable practices for password storage. It is *not* acceptable engineering practice to store passwords in plaintext. The inevitable security breach will end with your company sending a “sorry” disclosure email to its clients, and you will be responsible for the ensuing bad publicity.

Acceptable practices: **not** plaintext; hashed and salted. (We're not going to talk about what “salted” means here. Look it up if you need it. Hint: ECE 155.)

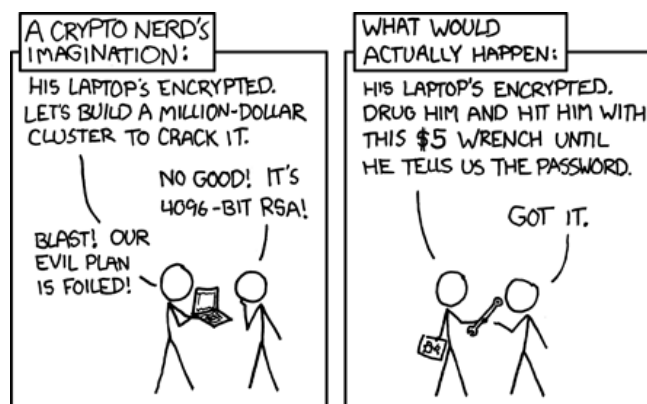
Cryptographic hashing. Instead of storing the plaintext password, you store a hash of the password, under a cryptographic hash function. One important property of a cryptographic hash function is that it must be (believed to be a) one-way function; that is: $x \mapsto f(x)$, the forward direction, must be easy to compute, but $f(x) \mapsto x$, the inverse mapping, must be hard to compute. Examples of such functions include SHA1 and *script*.

Breaking the hash. Even if there is no known short computation for the inverse function, it's always possible to brute-force the password computation by trying all possible passwords. Think about how GPUs work. Each potential password is a point in the computation space, and we compute the hash over all of them simultaneously. That's a lot of speedup. Custom hardware is also good, and those of you who know something about hardware could think about implementing it. Oh, hello, Bitcoin miners.

Arms race: making cracking difficult. The idea has always been to make it more difficult to compute the hash function, so that it's OK to try one password, but it's intractable to try one billion passwords. Even way back, UNIX passwords forced repeated applications of the hash function to accomplish these.

The main idea behind *script* is to make hashing expensive in both time and space, increasing both the number of operations and the cost of brute-forcing. Hence crackers will need to use more circuitry to break passwords.

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



(Source: xkcd 538)

Formalization. Let's make the notion of "expensive" a bit more formal. The idea is to force the use of the "most memory possible" for a given number of operations. More memory implies more circuitry required to implement.

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

Memory-hard algorithms are expensive to implement in either hardware or software.

Now, we want to move from particular algorithms to the underlying functions (that is, we would like to quantify over all possible algorithms). Intuitively, a *sequential memory-hard function* is one where (1) the fastest sequential algorithm is memory-hard; and (2) it is impossible for a parallel algorithm to asymptotically achieve lower cost.

Existence proof. Of course anyone can define anything. It's much better if the thing being defined actually exists. The script paper then goes on to exhibit ReMix, which is a concrete example of a sequential memory hard function.

Finally, the paper concludes with an example of a more realistic (cache-aware) model and a hard function in that context, BlockMix.

Reduced-resource computation

In an n-body simulation, you can implement an optimization that trades off accuracy for performance. In that specific case, domain knowledge would enable you to skip out on some unnecessary computation: points that are far away contribute only very small forces.

In [RHMS10], Martin Rinard summarizes two of his novel ideas for automatic or semiautomatic optimizations which trade accuracy for performance: early phase termination [Rin07] and loop perforation [HMS⁺09]. Both of these ideas are applicable to code we've learned about in this class.

Early phase termination

We've talked about barriers quite a bit. Recall that the idea is that no thread may proceed past a barrier until all of the threads reach the barrier. Waiting for other threads causes delays. Killing slow threads obviously speeds up the program. Well, that's easy.

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

Let's consider some arguments about when it may be acceptable to just kill (discard) tasks. Since we're not completely crazy, we can develop a statistical model of the program behaviour, and make sure that the tasks we kill don't introduce unacceptable distortions. Then when we run the program, we get an output and a confidence interval.

Two Examples. When might this work? Monte Carlo simulations are a good candidate; you're already picking points randomly. Raytracers can work as well. Both of these examples could spawn a lot of threads and wait for all threads to complete. In either case, you can compensate for missing data points, assuming that they look like the ones that you did compute.

Also recall that, in scientific computations, you're entering points that were measured (with some error) and that you're computing using machine numbers (also with some error). Computers are only providing simulations, not the ground truth; the question is whether the simulation is good enough.

Loop perforation

You can also apply the same idea to sequential programs. Instead of discarding tasks, the idea here is to discard loop iterations. Here's a simple example: instead of the loop,

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

simply write,

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

and multiply the end result by a factor of 2. This only works if the inputs are appropriately distributed, but it does give a factor 2 speedup.

Example domains. In [RHMS10], we can read that loop perforation works for evaluating forces on water molecules (in particular, summing numbers); Monte-Carlo simulation for swaption pricing; and video encoding. In that example, changing loop increments from 4 to 8 gives a speedup of 1.67, a signal to noise ratio decrease of 0.87%, and a bitrate increase of 18.47%, producing visually indistinguishable results. The computation looks like this:

```
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.

References

- [HMS⁺09] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT CSAIL, Cambridge, MA, September 2009.
- [Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009. Online; accessed 6-January-2016. URL: http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf.
- [RHMS10] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Proceedings of Onward! 2010*, pages 806–821, Reno/Tahoe, NV, USA, October 2010. ACM. URL: <http://doi.acm.org/10.1145/1932682.1869525>.
- [Rin07] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of OOPSLA 2007*, pages 369–386, Montreal, Quebec, Canada, October 2007.