Making a Hash of Things

Adam A. Smith and Ursula Whitcher

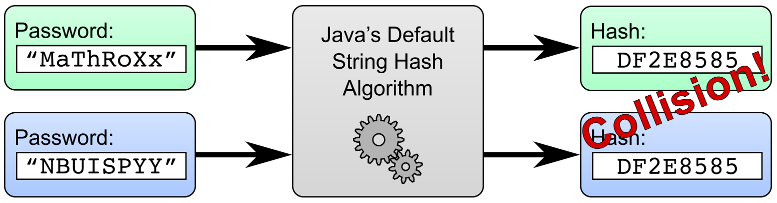

Figure 1. A collision.

Adobe, Ashley Madison, Snapchat, Target, Yahoo!, and Zappos: These are just a few of the well-known companies that have been hacked in the past year or two. If you have an account with any of these services, you probably received a message warning you to change your password. Does this mean a dastardly hacker has your old password? Maybe . . . or maybe not.

Smart websites don't store big lists of passwords; that would be asking for trouble. Instead, they store your user name and a *hashed* version of your password. A hashed password has been transformed by a *hash function*—a function that transforms a string of characters (such as your password) into a string of fixed length.

For example, if your password was `Il0vemath`, a website might store the hexadecimal number `0940a51639174af39ec5d5c1069fb2554453f76017c70426cde3aec984301158`. If your password was the more enthusiastic `Il0vemath!`, the same website would store `e8b690ca2dba3d4d45c-c99fa52e326ae446e4acda0a7f277ac3d8e-a068b0b415`. The original passwords are similar, but the hashed versions don't look anything alike!

What makes a good hash function? A key feature for password storage is *preimage resistance*. That means that it should be hard for a hacker who has stolen a hashed phrase to guess the original input. Ideally, the hacker's only way to find your password would be a giant game of guess and check—plugging phrases into the hash function until he hits on a working password.

If you're familiar with cryptography, preimage resistance may remind you of the quest for *one-way functions*, which are invertible functions that are easy to compute but hard to invert. Such functions are highly useful in encrypting secret messages.

Hash functions are different, however, because they don't have to be invertible: Multiple inputs can yield the same output. The situation in which two or more inputs have the same hash is called a *collision*. We illustrate a collision from a hash function used in the programming language Java in figure 1. (Note that this hash function is not meant to be secure!)

If typical passwords are shorter than the output of a hash function, collisions are unlikely. However, there are applications of hash functions in which it's advantageous to have outputs that are much shorter than the inputs, making collisions inevitable.

For example, hash functions can be used for file download authentication—to verify that a downloaded program has not been corrupted or altered. In this scenario, someone posts an executable file for download. If something went wrong in the transfer, or if someone made a small malicious change to the file, running the program could result in disaster. To prevent this from happening, the original programmer posts a hash of the correct code to the hosting website. Because the hash is short, it can be copied or read in entirety without fear of corruption. People who download the file can compute its hash. If the hash of the downloaded file matches the original programmer's hash, the program should be safe to run.

Some Passwords Are Easy to Guess

How hard is it for a hacker to guess a phrase with the same hash as someone's password? The answer depends a bit on human psychology, of course: If the password is `password`, guessing it will not be difficult. Let's analyze the mathematically simplest scenario, in which

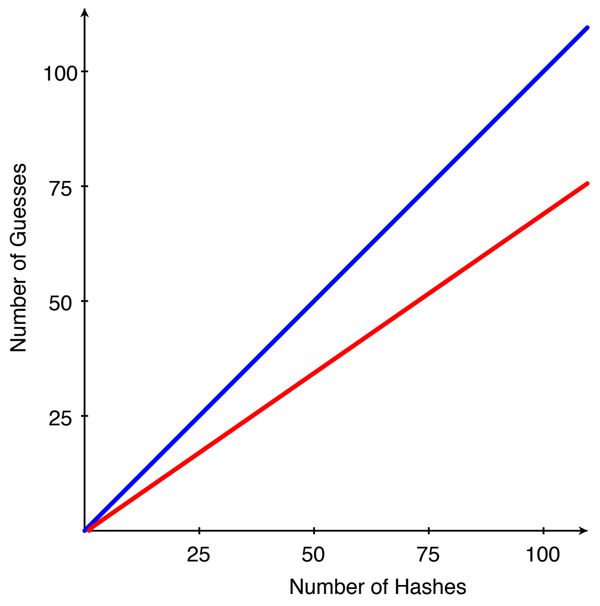

Figure 2. Hash size vs. the mean (blue) and median (red) number of guesses required.

the hacker has no reason to prefer one guess over another.

Suppose you and your professor have just returned from a summer math conference where you presented your research. Your school's travel reimbursement form is an old Excel worksheet (.xls format), and the overly paranoid administrator who created the form filled and password-protected certain fields. The problem is, some of these entries contain last year's information, and you want to fix them.

The worksheet doesn't store the password. Instead, it stores a hashed version of the password, with 2^{15} possible hashes. How long will it take to find a phrase with the right hash?

Let's assume that each of our guesses is equally likely to result in any of the 2^{15} possible hashes. That means our problem is equivalent to rolling a die with 2^{15} sides and computing how long it will take to roll a one. After k guesses, our probability of *not* rolling a one is $\left(\frac{2^{15}-1}{2^{15}}\right)^k$, so the probability that we will roll a one at least once is $1-\left(\frac{2^{15}-1}{2^{15}}\right)^k$. Solving for the number of attempts that will give us a 50 percent chance of rolling a one, we obtain

$$k = \frac{\ln(2)}{\ln(2^{15}) - \ln(2^{15}-1)} \approx 22{,}713.$$

This is the median number of rolls needed.

The expected number of attempts required—the mean—is a little larger. It is precisely the number of sides on the die: $2^{15} = 32{,}768$. (To see why this is the case, look up the *geometric distribution.*)

For someone used to pencil-and-paper computations, the prospect of making 32,768 guesses may sound daunting. But this is trivial for a computer: A modern desktop can do it in less than a second. If that's not fast enough for you, you could use multiple processors to make many guesses simultaneously, speeding up the process even more. You can easily find an input that will let you update the travel form!

How can we make a safer hash function? If we repeat our die-rolling analysis on a hash function with m possible hashes, we find that the median number of guesses is

$$\frac{\ln(2)}{\ln(m) - \ln(m-1)}.$$

Graphing this function, we find that for large m, it is asymptotic to a straight line with slope $\ln(2) \approx 0.693$ (see figure 2). (It is an interesting calculus exercise to justify this.) The expected number of guesses is m. Thus, the median and the mean number of guesses, and thus the security of the hash function, are directly proportional to the number of possible hashes.

A Better Hash Function

A currently popular hash function is named SHA-256. Here, SHA stands for Secure Hash Algorithm. The 256 tells us that the hashes consist of 256 *bits*, or binary digits, so there are 2^{256} (more than 10^{77}) possibilities. We used SHA-256 to generate the hashes of `I10vemath` and `I10vemath!` at the beginning of this article.

How does SHA-256 work? The starting point is the computer's binary representation of the input. The algorithm divides the input into *bytes*: Each byte is eight bits.

The first step of the algorithm is to pad the input with a 1 bit and many 0s before the original input, so that the total number of bytes is a multiple of 64. It is then split into n discrete 64-byte blocks, each of which is operated on separately.

Each 64-byte block is split into 16 32-bit values. Meanwhile, another 48 32-bit values are generated from complicated permutations and transformations of the data (rotating and shifting the binary digits, changing ones to zeros and vice versa, and so on). This results in 64 32-bit values for each of the original 64-byte blocks.

Next there are 64 rounds of further operations on each block. Each round uses one of the values generated before, together with some predefined constants, to update eight 32-bit variables called a_i through h_i, where i is the block's number. Again, we rotate and flip digits, change ones to zeros, and so on.

When these rounds are complete, we use the a_i to compute a new 32-bit value,

$$H_a = (C_a + a_1 + \cdots + a_n) \mod 2^{32}.$$

Here C_a is a constant, and the notation "mod 2^{32}" means that H_a is the remainder when we divide the value by 2^{32}. We compute H_b through H_h similarly.

Finally, we concatenate the eight 32-bit variables $H_a, \ldots, H_h$ into a single 256-bit value to output.

The SHA-256 hash function has a couple of interesting features. For one, the length of the input is concatenated to the end of the message during the original padding. This is to make it harder to generate two messages with the same hash. This isn't a huge issue in password storage. However, it provides extra security in other hash function applications, such as file download authentication, where we want to prevent malevolent additions to our files.

Another important design factor is the choice of the predefined constants. We don't want the binary digits of these constants to have obvious patterns, because this might make it easier to predict the output of the hash from its input. One way to avoid patterns would be for the algorithm's creators to choose each digit randomly. However, a user of a hash function may not trust that choices were random. Edward Snowden accused the National Security Agency of inserting weaknesses into a popular pseudo-random number generator to make its own code-breaking tasks easier. (For more about controversies involving government-defined constants and hash functions, see [1].)

Instead of random or pseudo-random numbers, SHA-256 uses part of the binary expansion of the square roots of the first eight prime numbers. The algorithm's designers believed that these numbers lacked obvious structure that could be exploited to predict hashes but were simple enough that average programmers could trust that the constants hadn't been manipulated. Because the constants have been carefully chosen to avoid opportunities for manipulation, they are sometimes referred to as "nothing-up-my-sleeve" numbers.

Is SHA-256 as secure as its name implies? Our previous calculations suggest that for a hacker trying to guess a particular password completely at random, half of the time it would take at least $\ln(2) \cdot 2^{256} \approx 8 \times 10^{76}$ guesses. Let's assume that we can make 1,000 guesses per second, which is a reasonable estimate for a modern computer. Then all 8×10^{76} guesses would take about 2.5×10^{66} years. For comparison, the entire universe is only about 1.4×10^{10} years old!

But the hacker's case isn't nearly as dire as this calculation suggests. To begin with, unless hackers are motivated by a burning personal vendetta, they don't need to guess a specific person's password. Website breaches can net thousands or millions of hashed passwords, and guessing any of them is profitable.

Second, human beings don't pick passwords at random: They pick passwords they can remember. Computers can try millions of combinations of common words and numbers very quickly, and hackers

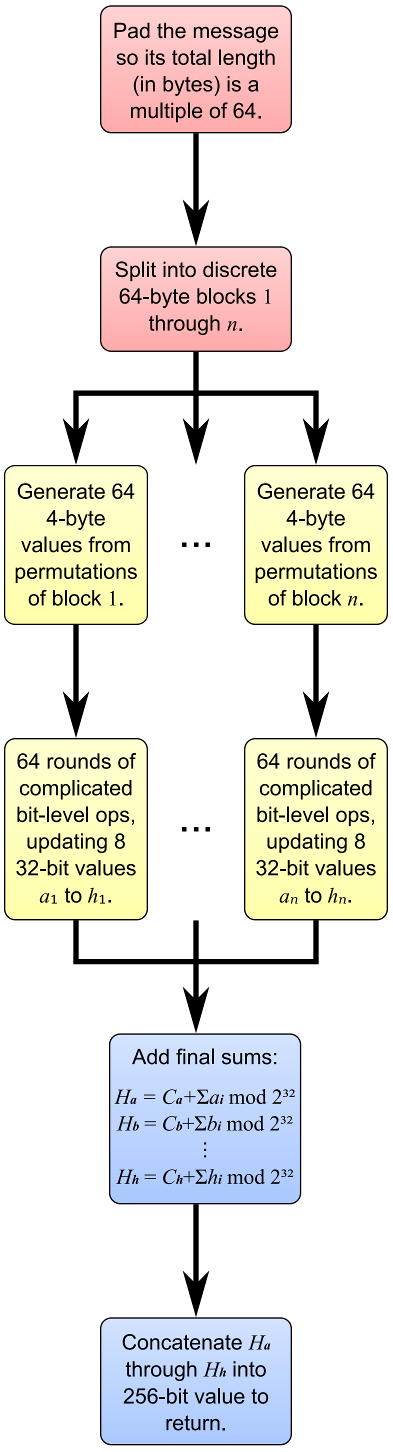
Pad the message so its total length (in bytes) is a multiple of 64.
Split into discrete 64-byte blocks 1 through n.
Generate 64 4-byte values from permutations of block 1.
...
Generate 64 4-byte values from permutations of block n.
64 rounds of complicated bit-level ops, updating 8 32-bit values a_1 to h_1.
...
64 rounds of complicated bit-level ops, updating 8 32-bit values a_n to h_n.
Add final sums:
$H_a = C_a + \Sigma a_i \bmod 2^{32}$
$H_b = C_b + \Sigma b_i \bmod 2^{32}$
⋮
$H_h = C_h + \Sigma h_i \bmod 2^{32}$
Concatenate H_a through H_h into 256-bit value to return.

Figure 3. The SHA-256 algorithm.

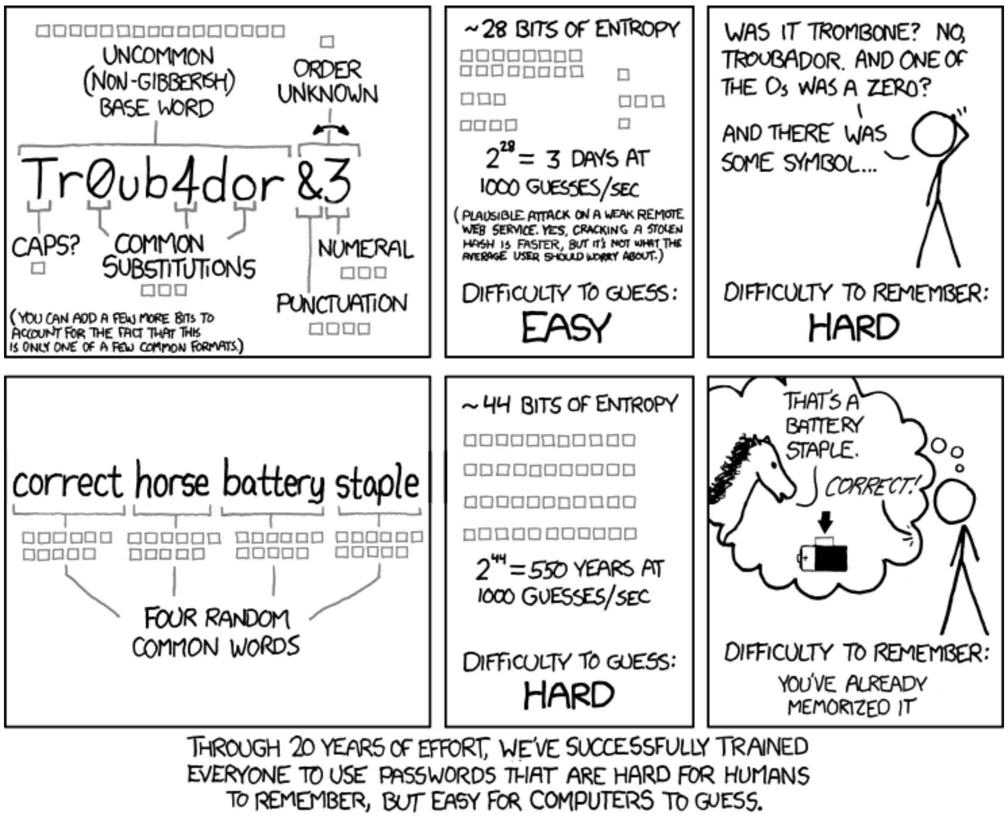
UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor &3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Figure 4. xkcd #936.

can use as many computers, with as many processors, as they can buy or suborn. If the website collects users with a common interest, such as sports fans or employees of a particular company, many of them will pick passwords that relate to that interest; that's another possible "in" for the hackers.

Finally, hackers can benefit from experience: One of the quickest ways to guess a password is to use a list of real passwords from another hack. For more details on real-world hacking methods, check out [2].

One defense against hacking is to train users better. People can learn to choose better passwords. (Please don't use `Il0vemath!` as your MAA website login!) For example, xkcd writer Randall Munroe has suggested building a story around randomly generated words (see figure 4). You could also use a password-keeping application that creates and stores long, unpredictable passwords, eliminating the frailty of human memory.

Other defenses are technical. Choosing hash functions that take a long time to evaluate or output big hashes can slow hackers down. A practice called *salting* combines a user's password with other data, such as her or his name or a randomly generated key, before hashing. Salting helps to ensure that even if two users pick the same password, the hashes are different.

The net effect is an arms race: Cryptographers work to invent new and better hash functions, while hackers try to break them. In the short term, you may want to change your passwords regularly. In the long term, inventing and testing new hash functions offers many interesting puzzles for the mathematically minded. ■

Further Reading

[1] Jonathan Berliner, NIST releases next generation SHA-3 hash function for public comment . . . after year of turmoil, *http://bit.ly/1HqIirV.*

[2] Dan Goodin, Anatomy of a hack: how crackers ransack passwords like "qeadzcwrsfxv1331," *Ars Technica* (May 27, 2013), *http://bit.ly/1Fxvult.*

[3] Joshua Holden, A good hash function is hard to find, and vice versa, *Cryptologia* **37** no. 2 (2013).

Adam A. Smith is a computer scientist at the University of Puget Sound. He enjoys seeing how long it takes students to realize that he's not an expert in their field.
Email: adamasmith@pugetsound.edu

Ursula Whitcher is a mathematician at the University of Wisconsin–Eau Claire. She enjoys teaching number theory and cryptography, finding people who are wrong on the Internet, and informing her cat, Jerome, that he is a kitty.
Email: whitchua@uwec.edu

http://dx.doi.org/10.4169/mathhorizons.23.2.5