Exploration, the basics of cryptographic hash algorithms

Introduction:

For the past two years I've started every day by entering a password in my computer to access online learning tools. After about a month of online learning, I began to ask myself the question of what exactly happened when I entered my password. The solution, as it happens, is based in math and logic. You might make the same mistake I did intuitively assume that your computer would check your password against a copy stored on the disk, however, this has many downsides. By storing a password on the disk, you make it accessible to any program running on your computer, so any site you go to would immediately know your password. Rather, you need a way to store a representation of the data that can be used to verify the original is the same as your password (preferably without letting the world know your password).

This practise of creating a digest that can only be reference one way is often termed hashing. Hashing is incredibly important to the way that we live our lives in the modern age, from logins, credit cards, and even to the web itself. After learning this I started to ask what exactly makes up a hash? Why do they have all of these unique properties? Why does the example below work?

$$\text{"Math is my favourite subject"} \xrightarrow{MD5\ hash\ algorth} 839fc7fa5f0a89335e94ce580ea47b06_{16}$$

What sparked my interest even further is that when I changed my data even a little the hash changed immensely:

$$\text{"Math's is my favourite subject"} \xrightarrow{MD5\ hash\ algorth} bf49b221715e77a83fb08fa36401fea3_{16}$$

Note: the subscript 16 is to represent the base of the counting system, so the hash's above represent an arbitrary integer value rather than that of a specific string. This will be developed further in the section on data normalization.

I used this as a jumping off point for my exploration of hash functions, in the attempt to answer as many of these questions as I can. After all, what exactly makes these hashes secure? What does secure even mean in this context?

Designing our own hash:

When I started trying to analyse what makes a good hash function, I ran into something that stumped me, how does a hash function take in all these different types of data? My first though was the idea that maybe whoever creates these hash functions does so for every data type they need or want. So, an engineer might build a different hash function for text, another for video, and yet another for files. The truth however lies in a practised call data normalization. This data normalization is fundamental to the functioning of hash functions. Data normalization is the conversion of any type of data into a standard form (often binary, though could be any integer representation). For some types of data, like text, this process is well defined with open standards. I've included that sample process below for text data:

1. Data is given in text format

$$\text{“}Binary\ is\ tough\text{”}$$

2. The data is split into an array of single characters

$$['\mathrm{B}','\mathrm{i}','\mathrm{n}','\mathrm{a}','\mathrm{r}','\mathrm{y}',' ','\mathrm{i}','\mathrm{s}',' ','\mathrm{t}','\mathrm{o}','\mathrm{u}','\mathrm{g}','\mathrm{h}']$$

3. Each charter is converted into an 8 bit (1 byte) representation in binary (this is because of the general standard of UTF-8[1] and is binary because it is intended for computers)

$$01000010_2\ 01101001_2\ 01101110_2\ 01100001_2\ 01110010_2\ 01111001_2\ \ldots$$

4a. The list of binary charters is joined to create a new string to be stored as binary

$$010000100110100101101110011000010111001001111001\ \ldots_2$$

4b. In our specific case because we care only about the numerical representations, we would convert each of these bytes to an integer

$$66_{10}\ 105_{10}\ 110_{10}\ 97_{10}\ 114_{10}\ 121_{10}\ 32_{10}\ 105_{10}\ 115_{10}\ 32_{10}\ 116_{10}\ 111_{10}\ 117_{10}\ 103_{10}\ 104_{10}$$

For the sake of explanation all that you need to know is that all types of data can be converted into an array of numbers.[2] This solves the issue of data normalization.

What makes a hash function?

This process of data normalization leads to the obvious question of: 'what do I do with some random numbers?'. In essence, we want to create one final number to represent our original

[1] Please note that the numerical representations are mostly arbitrary.

[2] Notably, this only applies to data that can be discreetly represented, for example if you were to sample some function f(x) you could sample it as much as possible better you could never have every possible set of values, just an appropriate sample.

input. We could even just do simple operations to create this number, say addition as an example.

$$\sum chr(x) = s$$

Note: 'chr' means to find the number corresponding to a letter or data point.

Continuing the previous example would look like this:

$$\sum [66, 105, 110, 97, 114, 121, 32, 105, 115, 32, 116, 111, 117, 103, 104] = 1448$$

There are some obvious flaws in this function. Can you think of ways to create arbitrary inputs to get a numerical output? Obviously. You can just find other sums. Simple mistakes like swapping two characters would still output the same hash value.

The property that we have just discovered is something called pre-image resistance. It's the idea that with only the output of the function we could create an input with the same output. Functionally this means somebody could 'crack' your password without ever actually knowing it.

There exists a simple solution. One so simple that it has been right in front of us the whole time. Just use a random number each time you hash. There are two version of this, the first is what we call a 'random oracle' meaning that the output of any hash is completely randomly generated at that time and is held forever; such that, any subsequent calls to this oracle have the same result. This is, in fact, the ideal hash function because it meets any and all of the requirements of a hash function you will see described. Sadly, because of the technical constraints required a hash function (particularly that everyone should be able to know how it works and that it works), it isn't possible.

The second implementation is too simple generate a random number each time you call your hash function. This presents one major issue; you wouldn't always get the same output from the same input. We call the core aspect of hashes which is currently being broken 'Determinism'. Determinism is the idea that we should always get the same output from the same input, no matter the machine its running on (aka substrate neutrality).

Putting it together:

At this point you might be beginning to see that hashing might not be as simple an affair as we once thought. So, I'll give up the game and show you what I came up with when presented with the exact same challenge and why it meets these criteria.

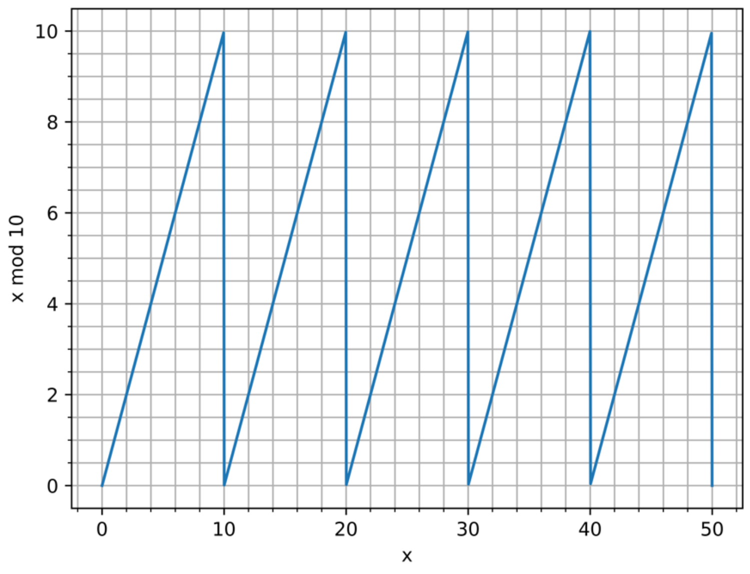
$$h(x) = \left(\sum_{n=0}^{len(x)} chr(x)(some\ large\ prime)^\wedge n\right) mod\ (base\ system^\wedge length)$$

So, what exactly makes this function any better than other? There are three main ‘features’ in this case.

The first (and likely most important) is: $chr(x)(some\ large\ prime)^\wedge n$. The reason for this, is because if you want to solve for $chr(x)$ you have to fully factorize what is sure to be a huge number. This calculation in and of itself could take weeks on a modern computer to fully factorize. Though in and of itself it doesn’t present strong security because if you know the prime factor then you could simply divide out and the remainder would be your $chr(x)$ value. Particularly, by raising to the n^{th} power you make reverse engineering the addition nearly impossible because you don’t have a common factor to take out.

This brings us to our second major feature, the repeated addition. The summation of the previous step ensures a couple of things. Primarily, it means that the only way to find the results of our previous multiplication would be to guess and check. IN doing so, you waste a lot more computational power.

Finally, the modulus operator further obfuscates the previous results. For those of you who might not know, the modulus operator works sort of like a clock in which every time you go over a certain value it ‘resets’ back to zero). This cyclical change is the equivalent of you only using the last n number of digits of a number. In practise, this means that it is almost impossible to find what the original number that you took the mod of, was. See the graph below for what exactly a modulus function looks like. In this case it doubly serves so that we can have a fixed-length output (the part computer scientists care about).



For example, if I were to tell you:

$$x \ mod \ 9 = 1$$

The only way for you to find x would be trials since there are infinite correct answers

$$x = 9g + 1; g \epsilon \mathbb{N}$$

Collectively these features mean that though it is possible to create multiple inputs for the same output or reverse engineer it. There is so many different branches you would have to brute force that it could take the lifetime of the universe. Yet any modern computer could complete this function in nanoseconds. Meaning that for all intensive preposes its one-way.

Notably we achieve another remarkable thing in this function by using both exponentiation and modular athematic. We get the 'avalanche' effect, wherein even small changes in the input result in large changes to the output. If you have ever tried to solve exponential equations, you should have a strong intuition for this.

The significance of hashing:

Now that we have this hash function, the possibilities are endless. Hash functions are all around us, and without them many of the systems that we know today wouldn't work. I've already talked at length about the applications of hashing in password storage, but hashing's biggest use is in lookups.[3] As an example if you search your computer, it would take ages for your computer to match what you enter by simply checking against every single file you have. Rather they use hashes to simply jump to the position in which your file is located. This means that the time complexity of search's can be brought down significantly (in technical notation this is called O(1) algorithm as opposed to searching all which would be O(n)).

Hashing is also commonly used to verify the integrity of data. It's easy for us humans (most of the time) to look at something on a computer and come to a conclusion; it's the same, or its not. However, for a computer, this task can be much harder. So, by hashing two versions of a file a computer can quickly tell if they are exactly the same.

In effect, every time you do anything with your computer there is almost definitely a hash function (or twenty) going on in the background to make sure that your experience is fast, secure, and functional.

[3] Notably hashing alone isn't considered good practise for password storage, this is due to vulnerability's like 'rainbow table'. The solutions involve something called 'salt and peppering' data, though it isn't the subject of this paper.

I hope then that the significance of hash functions isn't lost on you. The use of them s

Limitations of hash's:

There are two main limitations of a hash function like this, the first pertains only to this specific type of hash function (using modular arithmetic and prime multiplication), the second is related to all hash functions.

The first limitation is that it is theoretically possible that quantum computers will bring with them the ability to speed up the factorization of numbers. Say you have a charter with a know product. You would need to brute force the factorization of that prime number to find the number that you are multiplying by and corresponds to your data point. This would look like the equation below:

$$(p^n)(x) = s$$

Which after we substitute in know values would look like:

$$(941^n)(x) = 15335392564739329 \dots 1153927437751162626$$

To 'crack' what x is, we would need to factorize this huge number to find its only other factor (in this case; 66) for every value of n.[4] Though this type of computation is possible, in practise it is rarely pulled off due to the massive scale of the necessary computation. Especially considering that in practise you would only know the last few digits of the sum (represented by s) as part of a larger sum.

There however is a much more fundamental flaw in a hash that cannot be engineered away. Limited outputs with unlimited inputs. This necessarily means that there will always be a way to simply test to find an original value which will result in the desired statement because the below statement is always true:

$$|\mathbb{N}| > |\{x; 0 < x < b^l \text{ } and \text{ } x \in \mathbb{N}\}|; \ (b, l) \in \mathbb{N}^2$$

Which implies that there must be at least 1 output for which there exists more than one output. Though in a properly distributed hash function we should assume all outputs to have the same properties meaning that for all outputs there exists more than one input. Counterintuitively this means that our hash function isn't actually a 'function' as they are traditionally defined.

[4] Much like anybody who wishes to crack our hashing function, I'll leave calculating n to you.

Let’s stop for a second to explain what this statement means since it uses some notation many math 30 students might be unfamiliar with. Particularly the cardinality of sets. What the statement above says is that the set of all possible natural numbers ($\mathbb{N}$, whole numbers greater than 0) is larger than the set integer numbers greater than 0 and less than some arbitrary number b^l where b and l are both natural numbers. We know this is true because the set of all natural numbers is countably infinite as opposed to the set representing our output which must be finite.

Cryptographically, this is called the collision domain of our function is limited in any individual implementations this is unavoidable. Even if it was, this would result in such a large range of outpoints that it defeats the purpose of the hash function itself and would be more akin to a compression function.

Reflection:

Looking back, it never occurred to me that math was so applicable to our daily lives and how some niches like modular arithmetic or summation could be so integral to the security of our everyday lives.

Personally, what I found most interesting is how you can back up simple yet strong intuitions about some of the most fundamental operations in math to create something so surprisingly difficult to reverse engineer. The example that immediately pops into head is the cyclical nature of the modulus operator and the learnings I have about the remainder theorem and division.

Another thing that I found interesting was the direct connection to information theory from my other subjects like Philosophy, Computer Science, and Physics. Particularly with how we can represent an object in a discrete manner (see data normalization). For example, how exactly to we shred some of that information or entropy about our original data. Which I can now clearly see happening in the summation and modulus aspects of hashing as well as obviously in data normalization.

Having completed this exploration to a depth that I didn’t original plan. Especially having made small (to very large) excursions into fields like discrete math, set theory, philosophy[5], cryptography, formal systems of proof, and data science. Most of which was immensely interesting but sadly didn’t directly connect to hashing directly. I feel like this has helped me further my knowledge not only about math or cryptography, but the world around me in a more indirect sense. In the future I hope to take up a carrier in cryptography and get to study the most relied on technology of our age.

[5] Did he say philosophy?! Yes, as it turns out some ideas like ‘substrate neutrality’ and ‘underlying mindlessness’ have strong roots in the field.

Conclusion:

In conclusion, we have learned that fundamentally the hash functions that we so dearly rely on (whether we know it or not) are not nearly as simple as you might at first think. Who knew that the hunt for a 'one-way' function could prove so fruitful?

The key understandings that we should have are that, good hash functions are easy only one way, and that the way in which they do this is by combining addition, exponentiation, and modular arithmetic into one 'uncrackable' bundle.

Particularly though understanding that though the idea of a 'random oracle' is a great one in practise it is impossible. Any attempt of ours, however close, will have pitfalls. Like computers becoming faster, or even somebody not giving proper time to designing a hash function.

Moving forward I can see that the hash functions of today will soon be obsolete and new ones will take their place. In essence all that we have done is make the odds of 'breaking' a hash 1 in a hundred trillion billion. Even that, may not prove to be enough of a challenge.

Bibliography:

Patel, Mayur. *The Goulburn Hashing Function*. http://www.sinfocol.org/archivos/2009/11/Goulburn06.pdf.

"Cornell CS 3110 Lecture 21 Hash Functions." *Lecture 21: Hash Functions*, https://www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html.

"Secure Hash Standard (SHS) - NIST." *NIST - National Institute for Standards and Technology* , NIST, Aug. 2015, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

Steinebach, Martin, et al. *Robust Hash Algorithms for Text*, https://link.springer.com/content/pdf/10.1007%2F978-3-642-40779-6_11.pdf.

Menezes, Alfred J., et al. *Handbook of Applied Cryptology*. CRC, 1997.

"Hash Function." *Wikipedia*, Wikimedia Foundation, 5 Jan. 2022, https://en.wikipedia.org/wiki/Hash_function.

"Hash Table." *Wikipedia*, Wikimedia Foundation, 18 Jan. 2022, https://en.wikipedia.org/wiki/Hash_table.

theschmitzertheschmitzer, et al. "Why Should Hash Functions Use a Prime Number Modulus?" *Stack Overflow*, 1 July 1957, https://stackoverflow.com/questions/1145217/why-should-hash-functions-use-a-prime-number-modulus#1147232.

"21. Cryptography: Hash Functions - Youtube." *Youtube*, MIT OCW, 4 Mar. 2016, https://www.youtube.com/watch?v=KqqOXndnvic.

Pound, Mike. "SHA: Secure Hashing Algorithm - Computerphile - Youtube." Edited by Sean Riley, *Youtube*, Computerphile, 11 Apr. 2017, https://www.youtube.com/watch?v=DMtFhACPnTY.

Denk, Timo. "Hash Function Tool." *Online Tools*, https://tools.timodenk.com/?p=hash-function.

Xie, Tao, et al. "Fast Collision Attack on MD5 ." *Cryptology EPrint Archive*, 2013, https://eprint.iacr.org/2013/170.pdf.

Holden, Joshua. "6.1-6.9, 8.4." *The Mathematics of Secrets: Cryptography from Caesar Ciphers to Digital Encryption*, Princeton University Press, Princeton, NJ, 2019.

Hammack, Richard H. *Book of Proof*, Virginia Commonwealth University, Virginia, 2018, pp. 3–127.

Dennett, Daniel C. *Intuition Pumps and Other Tools for Thinking*. W.W. Norton & Company, 2014.