Fundamental difference between Hashing and Encryption algorithms

I see a lot of confusion between hashes and encryption algorithms and I would like to hear some more expert advice about:

1. When to use hashes vs encryptions
2. What makes a hash or encryption algorithm different (from a theoretical/mathematical level) i.e. what makes hashes irreversible (without aid of a rainbow tree)

Here are some *similar* SO Questions that didn't go into as much detail as I was looking for:

What is the difference between Obfuscation, Hashing, and Encryption?
Difference between encryption and hashing

security encryption hash cryptography

edited Jun 9 '13 at 19:19

Anna Lear ♦
18.3k ● 8 ● 51 ● 83

asked Feb 9 '11 at 17:30

Kenny Cason
4,366 ● 8 ● 23 ● 56

10 I can foresee this being *the* question to refer people to when they confuse the terms. :) – Adam Paynter Feb 9 '11 at 17:38

2 hashing is one way (cannot be reverted), encryption is two-way (can be decrypted) – bestsss Feb 12 '11 at 22:32

Hashes are also useful for indexing large structures and objects, e.g. files. See hash table. – HABO Jun 12 '13 at 1:04

2 Hashing is like a meat grinder. You can turn a cow to hamburger, but not the reverse. – Neil McGuigan Mar 18 at 18:03

I noticed my question was edited. I had always known the top level differences between the two but was more curious about low level/mathematical differences. :) Either way, lots of good content for SO! Many thanks! – Kenny Cason Mar 19 at 0:09

9 Answers

Well, you could look it up in Wikipedia... But since you want an explanation, I'll do my best here:

Hash Functions

They provide a mapping between an arbitrary length input, and a (usually) fixed length (or smaller length) output. It can be anything from a simple crc32, to a full blown cryptographic hash function such as MD5 or SHA1/2/256/512. The point is that there's a one-way mapping going on. It's always a many:1 mapping (meaning there will always be collisions) since every function produces a smaller output than it's capable of inputting (If you feed every possible 1mb file into MD5, you'll get a ton of collisions).

The reason they are hard (or impossible in practicality) to reverse is because of how they work internally. Most cryptographic hash functions iterate over the input set many times to produce the output. So if we look at each fixed length chunk of input (which is algorithm dependent), the hash function will call that the current state. It will then iterate over the state and change it to a new one and use that as feedback into itself (MD5 does this 64 times for each 512bit chunk of data). It then somehow combines the resultant states from all these iterations back together to form the resultant hash.

Now, if you wanted to decode the hash, you'd first need to figure out how to split the given hash into its iterated states (1 possibility for inputs smaller than the size of a chunk of data, many for larger inputs). Then you'd need to reverse the iteration for each state. Now, to explain why this is VERY hard, imagine trying to deduce `a` and `b` from the following formula: `10 = a + b`. There are 10 positive combinations of `a` and `b` that can work. Now loop over that a bunch of times: `tmp = a + b; a = b; b = tmp`. For 64 iterations, you'd have over 10^64 possibilities to try. And that's just a simple addition where some state is preserved from iteration to iteration. Real hash functions do a lot more than 1 operation (MD5 does about 15 operations on 4 state variables). And since the next iteration depends on the state of the previous and the previous is destroyed in creating the current state, it's all but impossible to determine the input state that led to a given output state (for each iteration no less). Combine that, with the large number of possibilities involved, and decoding even an MD5 will take a near infinite (but not infinite) amount of resources. So many resources that it's actually significantly cheaper to brute-force the hash if you have an idea of the size of the input (for smaller inputs) than it is to even try to decode the hash.

Encryption Functions

They provide a 1:1 mapping between an arbitrary length input and and output. And they are always reversible. The important thing to note is that it's reversible using some method. And it's always 1:1 for a given key. Now, there are multiple input:key pairs that might generate the same output (in fact there usually are, depending on the encryption function). Good encrypted data is indistinguishable from random noise. This is different from a good hash output which is always of a consistent format.

Use Cases

Use a hash function when you want to compare a value but can't store the plain representation (for any number of reasons). Passwords should fit this use-case very well since you don't want to store them plain-text for security reasons (and shouldn't). But what if you wanted to check a filesystem for pirated music files? It would be impractical to store 3 mb per music file. So instead, take the hash of the file, and store that (md5 would store 16 bytes instead of 3mb). That way, you just hash each file and compare to the stored database of hashes (This isn't practical because of re-encoding, changing file headers, etc, but it's an example use-case).

Use a hash function when you're checking validity of input data. That's what they are designed for. If you have 2 pieces of input, and want to check to see if they are the same, run it through a hash function. The probability of a collision is astronomical for small input sizes (assuming a good hash function). That's why it's recommended for passwords. For passwords up to 32 characters, md5 has 4 times the output space. Sha1 has 6 times the output space (about). Sha512 has about 16 times the output space. You don't really care what the password *was*, you care if it's the same as the one that was stored. That's why you should use hashes for passwords.

Use encryption whenever you need to get the input data back out. Notice the word **need**. If you're storing credit card numbers, you need to get them back out at some point, but don't want to store them plain text. So instead, store the encrypted version and keep the key as safe as possible.

Hash functions are also great for signing data. For example, if you're using HMAC, you sign a piece of data by taking a hash of the data concatenated with a known but not transmitted value (a secret value). So you send the plain-text and the hmac hash. Then, the receiver simply hashes the submitted data with the known value and checks to see if it matches the transmitted hmac. If it's the same, you know it wasn't tampered with by a party without the secret value. This is commonly used in secure cookie systems by HTTP frameworks, as well as in message transmission of data over HTTP where you want some validity to the data.

A note on hashes for passwords:

A key feature of cryptographic hash functions is that they should be very fast to create, and **very** difficult/slow to reverse (so much so that it's practically impossible). This poses a problem with passwords. If you store `sha512(password)`, you're not doing a thing to guard against rainbow tables or brute force attacks. Remember, the hash function was designed for speed. So it's trivial for an attacker to just run a dictionary through the hash function and test each result.

Adding a salt helps matters since it adds a bit of unknown data to the hash. So instead of finding anything that matches `md5(foo)`, they need to find something that when added to the known salt produces `md5(foo.salt)` (which is very much harder to do). But it still doesn't solve the speed problem since if they know the salt it's just a matter of running the dictionary through.

So, there are ways of dealing with this. One popular method is called key strengthening (or key stretching). Basically, you iterate over a hash many times (thousands usually). This does two things. First, it slows down the runtime of the hashing algorithm significantly. Second, if implemented right (passing the input and salt back in on each iteration) actually increases the entropy (available space) for the output, reducing the chances of collisions. A trivial implementation is:

```
var hash = password + salt;
for (var i = 0; i < 5000; i++) {
    hash = sha512(hash + password + salt);
}
```

There are other, more standard implementations such as PBKDF2, BCrypt. But this technique is used by quite a few security related systems (such as PGP, WPA, Apache and OpenSSL).

The bottom line, `hash(password)` is not good enough. `hash(password + salt)` is better, but still not good enough... Use a stretched hash mechanism to produce your password hashes...

Another note on trivial stretching

Do not under any circumstances feed the output of one hash directly back into the hash function:

```
hash = sha512(password + salt);
for (i = 0; i < 1000; i++) {
    hash = sha512(hash); // <-- Do NOT do this!
}
```

The reason for this has to do with collisions. Remember that all hash functions have collisions

because the possible output space (the number of possible outputs) is smaller than then input space. To see why, let's look at what happens. To preface this, let's make the assumption that there's a 0.001% chance of collision from `sha1()` (it's **much** lower in reality, but for demonstration purposes).

```
hash1 = sha1(password + salt);
```

Now, hash1 has a probability of collision of 0.001%. But when we do the next `hash2 = sha1(hash1);`, **all collisions of `hash1` automatically become collisions of `hash2`**. So now, we have hash1's rate at 0.001%, and the 2nd sha1 call adds to that. So now, `hash2` has a probability of collision of 0.002%. That's twice as many chances! Each iteration will add another `0.001%` chance of collision to the result. So, with 1000 iterations, the chance of collision jumped from a trivial 0.001% to 1%. Now, the degradation is linear, and the real probabilities are **far** smaller, but the effect is the same (an estimation of the chance of a single collision with `md5` is about `1/(2^128)` or `1/3e38`. While that seems small, thanks to the birthday attack it's not really as small as it seems).

Instead, by re-appending the salt and password each time, you're re-introducing data back into the hash function. So any collisions of any particular round are no longer collisions of the next round. So:

```
hash = sha512(password + salt);
for (i = 0; i < 1000; i++) {
    hash = sha512(hash + password + salt);
}
```

Has the same chance of collision as the native `sha512` function. Which is what you want. Use that instead..

edited Dec 20 '11 at 12:08

Iceland_jack
936 ● 1 ● 14 ● 29

answered Feb 9 '11 at 17:36

ircmaxell
95.6k ● 19 ● 181 ● 249

@ircmaxell : I think the main reason for not decoding the hash output is that you don't know the size of the input. Whatever be the input you get the same size of hte output. So you don't know what the size of the original messasge was. – Ashwin Apr 17 '12 at 2:25

@ircmaxell : I did not quite understand the reversing of iteration. MD5 provides 15 functions like you said. The functions can be inverted to find the inout for each iteration right? – Ashwin Apr 17 '12 at 2:29

7 Too bad the programmers at LinkedIn didn't read this before they stored passwords as unsalted SHA1 hashes... money.cnn.com/2012/06/06/technology/linkedin-password-hack/… – Eric J. Jun 8 '12 at 15:56

5 Why does this answer give so little emphasis on encryption? – Pacerier Jul 18 '12 at 0:18

1 Great answer. My only nitpick is that the degradation of trivial stretching can't be linear or it would eventually pass 100%. I think in your example with .001% the second step should be .001 + (1 - 0.001) * .001, or 0.001999. – AlexDev Jun 11 at 17:19

A hash function could be considered the same as baking a loaf of bread. You start out with inputs (flour, water, yeast, etc...) and after applying the hash function (mixing + baking), you end up with an output: a loaf of bread.

Going the other way is extraordinarily difficult - you can't really separate the bread back into flour, water, yeast - some of that was lost during the baking process, and you can never tell exactly how much water or flour or yeast was used for a particular loaf, because that information was destroyed by the hashing function (aka the oven).

Many different variants of inputs will theoretically produce identical loaves (e.g. 2 cups of water and 1 tsbp of yeast produce exactly the same loaf as 2.1 cups of water and 0.9tsbp of yeast), but given one of those loaves, you can't tell exactly what combo of inputs produced it.

Encryption, on the other hand, could be viewed as a safe deposit box. Whatever you put in there comes back out, as long as you possess the key with which it was locked up in the first place. It's a symmetric operation. Given a key and some input, you get a certain output. Given that output, and the same key, you'll get back the original input. It's a 1:1 mapping.

answered Feb 9 '11 at 17:40

Marc B
240k ● 18 ● 165 ● 293

37 I prefer the "hamburger back to cow" analogy for reversing a hash function ;) – caf Feb 10 '11 at 0:45

Use hashes when you don't want to be able to get back the original input, use encryption when you do.

Hashes take some input and turn it into some bits (usually thought of as a number, like a 32 bit integer, 64 bit integer, etc). The same input will always produce the same hash, but you PRINCIPALLY lose information in the process so you can't reliably reproduce the original input (there are a few caveats to that however).

Encryption principally preserves all of the information you put into the encryption function, just makes it hard (ideally impossible) for anyone to reverse back to the original input without possessing a specific key.

Simple Example of Hashing

Here's a trivial example to help you understand why hashing can't (in the general case) get back the original input. Say I'm creating a 1-bit hash. My hash function takes a bit string as input and sets the hash to 1 if there are an even number of bits set in the input string, else 0 if there were an odd number.

Example:

```
Input    Hash
0010     0
0011     1
0110     1
1000     0
```

Note that there are many input values that result in a hash of 0, and many that result in a hash of 1. If you know the hash is 0, you can't know for sure what the original input was.

By the way, this 1-bit hash isn't exactly contrived... have a look at parity bit.

Simple Example of Encryption

You might encrypt text by using a simple letter substitution, say if the input is A, you write B. If the input is B, you write C. All the way to the end of the alphabet, where if the input is Z, you write A again.

```
Input   Encrypted
CAT     DBU
ZOO     APP
```

Just like the simple hash example, this type of encryption has been used historically.

edited Feb 9 '11 at 17:40

answered Feb 9 '11 at 17:33

Eric J.

93k ● 29 ● 187 ● 352

My one liner... generally Interviewer wanted the below answer.

Hashing is one way . You can not get convert your data/ string from a hash code.

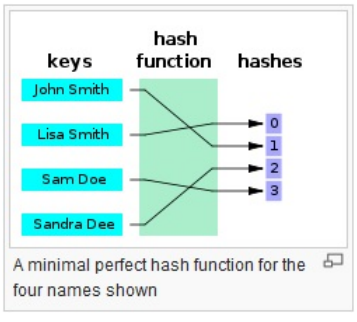
Encryption is 2 way - you can decrypt again the encrypted string if you have the key with you.

answered Jan 29 '13 at 5:10

Shiv Mohan

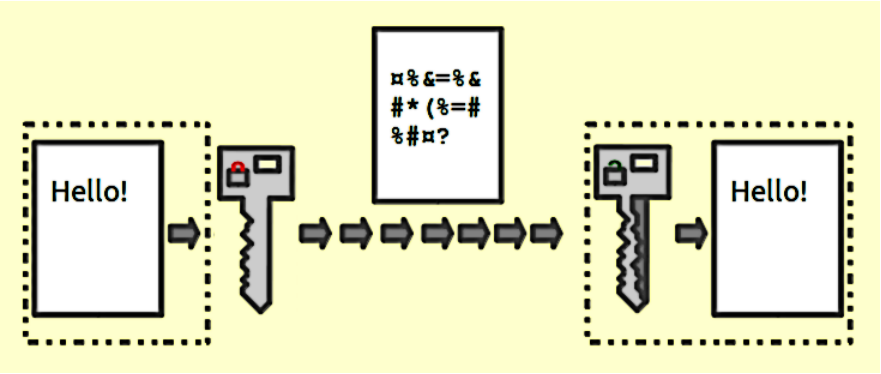
147 ● 2 ● 9

Hash function turns a variable-sized amount of text into a fixed-sized output



A minimal perfect hash function for the four names shown

Encryption function turns a text into an unreadable ciphertextwith the use of an encryption key.



Source: Wikipedia

edited May 7 '14 at 15:28

cryptic ツ
11.1k ● 7 ● 32 ● 56

answered Apr 10 '14 at 9:14

Julian
529 ● 6 ● 26

---

1. Use hashes when you only need to go one way. For example, for passwords in a system, you use hashing because you will only ever verify that the value a user entered, after hashing, matches the value in your repository. With encryption, you can go two ways.
2. hashing algorithms and encryption algorithms are just mathematical algorithms. So in that respect they are not different -- its all just mathematical formulas. Semantics wise, though, there is the very big distinction between hashing (one-way) and encryption(two-way). Why are hashes irreversible? Because they are designed to be that way, because sometimes you want a one-way operation.

answered Feb 9 '11 at 17:37

hvgotcodes
64.2k ● 14 ● 110 ● 168

---

when it comes to security for transmitting data i.e Two way communication you use encryption.All encryption requires a key

when it comes to authorization you use hashing.There is no key in hashing

Hashing takes any amount of data (binary or text) and creates a constant-length hash representing a checksum for the data. For example, the hash might be 16 bytes. Different hashing algorithms produce different size hashes. You obviously cannot re-create the original data from the hash, but you can hash the data again to see if the same hash value is generated. One-way Unix-based passwords work this way. The password is stored as a hash value, and to log onto a system, the password you type is hashed, and the hash value is compared against the hash of the real password. If they match, then you must've typed the correct password

why is hashing irreversible :

Hashing isn't reversible because the input-to-hash mapping is not 1-to-1. Having two inputs map to the same hash value is usually referred to as a "hash collision". For security purposes, one of the properties of a "good" hash function is that collisions are rare in practical use.

edited Feb 9 '11 at 17:45

answered Feb 9 '11 at 17:37

ayush
7,492 ● 6 ● 32 ● 60

"Hashing isn't reversible because the input-to-hash mapping is not 1-to-1", Thanks, I think that is a very important factor when it comes to differentiating hashes from encryptions! :) – Kenny Cason Feb 9 '11 at 18:48

Encryption and hash algorithms work in similar ways. In each case, there is a need to create confusion and diffusion amongst the bits. Boiled down, *confusion* is creating a complex relationship between the key and the ciphertext, and *diffusion* is spreading the information of each bit around.

Many hash functions actually use encryption algorithms (or primitives of encryption algorithms. For example, the SHA-3 candidate Skein uses Threefish as the underlying method to process each block. The difference is that instead of keeping each block of ciphertext, they are destructively, deterministically merged together to a fixed length

answered Feb 9 '11 at 17:43


Justin Morgan
1,911 ● 1 ● 8 ● 18

In Simple way

If You **Hash** any **plain text** again you **can't** get that plain text from that data (which is created by hashing)

If You **Encrypt** any **plain text** with key again you **can** get that plain text with **Decryption** of the data (which is created by Encryption) with the same key in symmetric algorithms.

Source: link

Hope you got it

edited Jun 14 at 10:31

answered Apr 7 '12 at 8:48


MR Srinivas
2,130 ● 1 ● 11 ● 26

Not necessarily the same key. – Respect My Authoritah Jun 10 at 16:45

@RespectMyAuthoritah: I mean to say it in symmetric cryptography. – MR Srinivas Jun 14 at 10:32