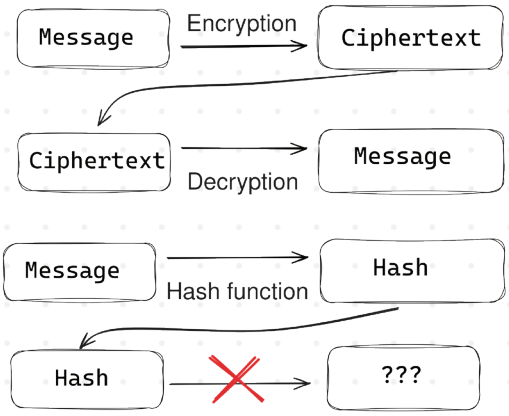
Protocol Foundations 003: Hashing

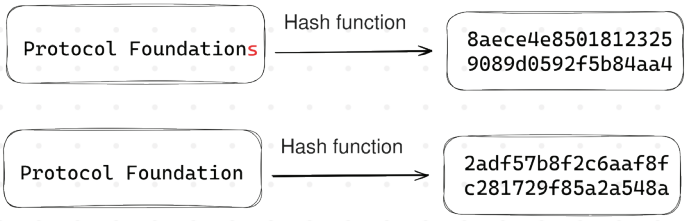
Mario Havel and Tim Beiko

Building on the concepts introduced in previous Protocol Foundations issues, let's now explore another fundamental component powering our digital world: *hashing*.

A *hash* is a one-way function that takes arbitrary-sized data as an input and produces a fixed-sized output, called a *hash* or *digest*. Its most important feature is that a given input will always produce the same output. Additionally, leveraging cryptography, hash functions can be designed such that an output reveals no information about the function's input. In contrast with encryption, which allows for later decryption of the ciphertext, cryptographic hashing is, for most practical purposes, *irreversible*.



More precisely, hash functions are *deterministic*, meaning that their output will always be the same for a given input. In addition to this, distinct inputs generate outputs which are *uncorrelated* to hashes of similar inputs. Changing a single bit in a large input file, for example, will completely change its hash with no predictable pattern. This, however, does not imply outputs are random. Again, while a hash output string may appear randomly generated, and even use pseudorandomness when transforming the input value, it is fully deterministic.



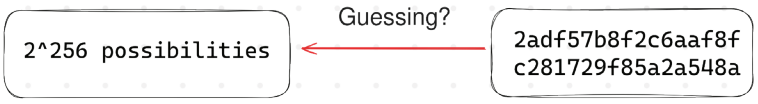
When removing a single character from the input string, the hash function's output, in this case sha256, changes completely.

The main property of a hash function is, therefore, to produce a unique output for each arbitrary input. While a hash function's outputs all have the same format (*e.g.*, 256 bits represented by a 64 character hexadecimal string[1]), their domain should be broad enough, on the order of "atoms in the universe," so that different inputs never collide in the output space. A *hash collision* refers to a hash function producing the same output for two different inputs and is considered a security or implementation failure.

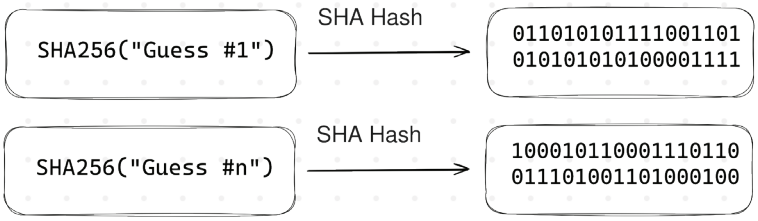
To give a specific example, the popular hashing algorithm sha256 produces a hash of length 256 bits which means there are 2^256 possible output combinations. To find a message whose output is a specific 256 bit hash, one must try and check the hash of random messages over and over. To search through this astronomically large space using the entire world's current computing power would take billions of years.

This valuable feature opens up many practical use cases, *e.g.*, storing sensitive data such as passwords. Website services don't (or at least *shouldn't*) store the passwords of registered users directly. Instead,

1. 256 character strings can represent 2^256 possible numbers using binary representation (*i.e.*, 0's and 1's). Since 2^256 is equal to 16^64, the same numbers can be represented in 64 character strings using hexadecimal representation (*i.e.*, 0–9 and A–F).



they typically store hashes of them. When logging in, a user's password is hashed and compared to the stored hash in the service's database. This enables password verification, as only the valid password will produce the correct hash, without giving anyone with access to the database, be they employees or hackers, access to the plain text versions.



Fixed-sized outputs are another valuable practical property of hash functions. Creating uniform, fixed-sized outputs for different kinds of inputs enables more efficient data storage and verification. For example, when making important files downloadable, websites often provide users with a hash of its content for verification. Users can then independently run a hash function on the downloaded file to make sure its hash matches the one provided by the website, proving the file is unaltered. Similarly, services which allow users to upload files will hash their content and use that to detect duplicate uploads.

One last, highly impactful, use case for hashes is the proof-of-work mechanism. In short, the idea is to create a "challenge," consisting of a constrained output space of valid hashes. The solver of the challenge needs to produce a hash which fits into the constrained output space. Because hash outputs are uncorrelated, the only way to solve the challenge is by running the hash function repeatedly with random inputs. The more the domain of the outputs is constrained, the more runs of the hash function—and hence computing power, or "work"—are required to find an input whose output satisfies the constraints.

To use a toy example, let's imagine a deck of cards from which the solver must pick a card. A constraint of "hearts or spades" means that half of the solvers' random picks would, on average, satisfy the constraint. If the constraint is narrowed to "hearts only," then ¼ of picks would on average satisfy it. Proof-of-work mimics this process but instead uses cryptographic hash functions and adds numerical constraints to their output space. If we represent a hash function's numerical outputs in binary format, using only 1's and 0's, we can then require a certain number of digits to match a predefined pattern. For example, requiring the first digit of an output to be 0 would constrain the domain such that half of the hash function's inputs would produce a valid output, like "hearts and spades" for our deck of cards. Requiring the second digit to also be 0 would halve that again such that ¼ of inputs produce a valid output, and so on.

Proof-of-work was originally proposed as a solution against email spam. By requiring email senders to "mine" a hash with easy-to-fulfill constraints and including it alongside their message, individual users could send emails as they currently do but large-scale spam campaigns would be too costly to take on.

Today, the main use cases for proof-of-work is securing blockchains without the need for a trusted third party. Blockchains can allocate the right to create the next block in the chain, as well as the rewards and transaction fees associated with that block, using public proof-of-work challenges. These networks broadcast the constraints (a.k.a. *difficulty*) publicly and then wait for a solver (a.k.a. *miner*) to propose a solution. While it can be quite computationally costly for miners to find a valid solution to the challenge, the computational work to verify a solution's validity is trivial. Once a valid input is found and

shared by a miner, all other nodes in the blockchain network run it through the network's standard hash function to confirm whether the output meets the required constraints. To go back to our toy example, this is similar to the difference between "randomly drawing the Ace of Spades out of a deck of cards" and "showing the card you've drawn."

This mechanism enables cryptocurrency networks to throttle the rate of new blocks on their network, by increasing or lowering the proof-of-work difficulty, in a way that requires no central intervention to respond to changes in mining demand.

Hopefully this clarifies a common misconception about cryptocurrency mining: thinking it involves "solving difficult computational problems." In fact, miners are solving simple problems (evaluating a hash function) over and over, billions to trillions of times, in order to find a solution in a constrained output space. Everyone else then easily verifies that hash's validity.

MARIO HAVEL & TIM BEIKO are part of the Ethereum Foundation's Protocol Support team which helps facilitate Ethereum network upgrades as well as other protocol-related initiatives, including Summer of Protocols.
MARIO | github.com/taxmeifyoucan
TIM | warpcast.com/tim

ProtocolKit

ProtocolKit
summerofprotocols.com
hello@summerofprotocols.com

ISBN-13: 978-1-962872-53-9
Printed in the United States of America
Printing history: March 2024

1 3 5 7 9 10 8 6 4 2

ISBN 978-1-962872-53-9
90000>
9 781962 872539

ProtocolKit

RETROSPECTUS

NEWSLETTER