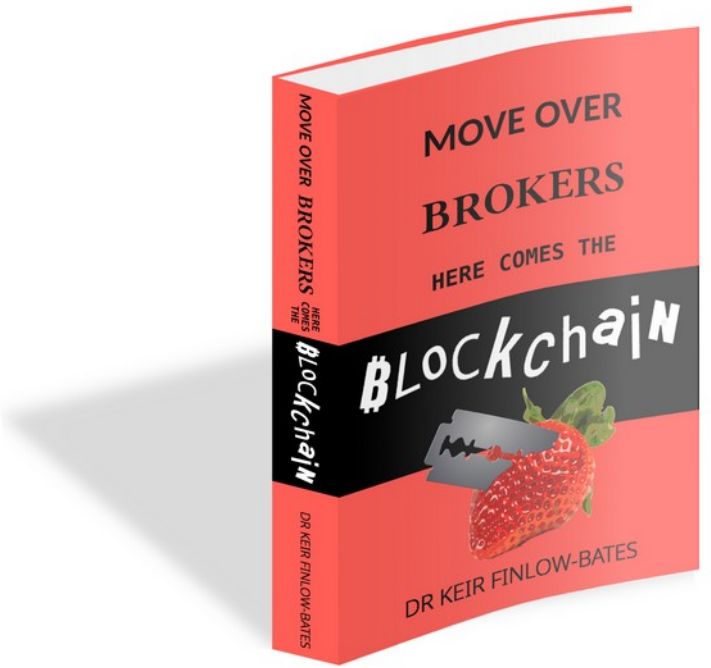
HASH FUNCTIONS

Dr Keir Finlow-Bates

Chapter extract from
Move Over Brokers Here Comes The Blockchain



Hash Functions

You don't do what you want to
But you do the same thing every day
Jello Biafra, 1987

Two chapters ago I said I would explain what a hash-linked list was, and why hash-linked lists, and indeed hashes, are of fundamental importance in the blockchain world. As is usually the case with blockchain, hash-linked lists only offer one part of the solution, which is to provide non-repudiation. The other part is called consensus – either through proof of work or through some other means.

I've just read back over that paragraph, and realized that there are too many terms and concepts packed into too few words. It's the kind of paragraph that makes sense if you already understand the individual concepts and the effect of their combination, but if you don't, then it's gibberish.

So let's back up, forget about computer science and cryptography concepts, and start by looking at something in the familiar social world, namely what non-repudiation is.

That never happened!

When it comes to complicated social interactions, people often go back on their word. A handshake deal turns into an argument after one party fails to deliver to the other, or delivers something that the second party considers insufficient.

That is why society invented contracts – agreements, usually (but not always) drafted in writing, in which all the terms and conditions are laid out, along with costs, deadlines, deliverables, and above all penalties if the contract isn't honored. Contracts are backed by the courts, and hence governments with their ultimate threat of force: either imprisonment or the seizure of assets.

> ***repudiation**: the act of refusing to accept something or someone as true, good, or reasonable.*
>
> *— the Cambridge Dictionary*[26]

So what happens if one party claims that they never signed the contract? Or that the sheaf of papers being waved by the other party has been altered, and the terms currently being presented weren't the terms that were initially agreed on?

This is known as repudiation, and a number of techniques to ensure the opposite, namely non-repudiation, have been used through history to avoid this, the two most obvious being signatures and witnesses.

Signatures do not just include scrawling your name in your own handwriting or style on a piece of paper. In days gone by things called "seals" were used. Devices such as signet rings or chops (symbols carved into jade or soapstone) for making unforgeable marks in wax or with ink, presumably by the individual who had control of the seal. And in these modern days, we have digital signatures, in which the controller of a unique sequence of ones and zeros can use software based on cryptographic algorithms to create

yet another sequence of ones and zeros that only the holder of a digital signing key could have produced.

Witnesses are usually people with “skin in the game”, either notaries who are effectively paid witnesses, or other professionals such as lawyers, doctors, or priests, who have something to lose from lying about their earlier attestation.

There is, however, a third way. You can put the document out in public, which means that enough people will have a copy for future comparisons, in which case you are using the world as a witness. But what if your document is very large? Or even worse, what if you want to prove at a later date that you had the document in your possession, but it contains confidential information that you can’t simply publish on the World Wide Web?

Cryptographic hash functions to the rescue!

Shorter is better

To begin with, you need to understand what a hash function is: an algorithm that takes as its input a data file of any size (so it will work on any kind of data, whether it’s a short string of letters and numbers, or a large video file), and outputs a fixed-length number. Furthermore, if you put the same data in at a later date, you get the same output back. Hash functions don’t change their mind over time, and there is no true randomness built into them.

The initial use of hash functions was for indexing and then searching for data. An analogy is car number plates. It is much easier to report a car to the police, and for them to subsequently find it if you can give them the number on the plate. Describing the make, model, color, and distinguishing marks such as fluffy dice hanging from the rear-view mirror doesn’t really cut it.

In Finland, number plates are typically two or three letters, a dash, and one, two, or three digits.

However, number plates are simply serially assigned, so there is no algorithm used to generate them. The authorities could have used a simple formula based on the vehicle identification number (VIN),

instead. The VIN itself can't be used, because it is 17 characters long, which would make it too small to read if put on a number plate.

A simple example would be to use the first three letters of the VIN and sum each pair of digits of the serial number (modulo 10) to get the last three digits of the plate. So for example, a vehicle with a VIN of SCEDT26T8BD005261 would be fitted with a much shorter number plate that reads SCE-077, as shown in figure 1.

Here you can immediately see one of the problems with hash functions – different inputs can map to the same output. The first three characters of a VIN number represent the "world manufacturer identifier", and the last six digits are the vehicle serial number, so two identical models of the car made in the same factory nine units apart will often end up with the same number plate.

In the computer science world, this is described as a collision, and a good hash function is supposed to be collision-resistant. Two different inputs to the hash function should have an extremely low probability of producing the same output. Our number plate algorithm obviously doesn't satisfy that requirement, so it's not a good hashing function.

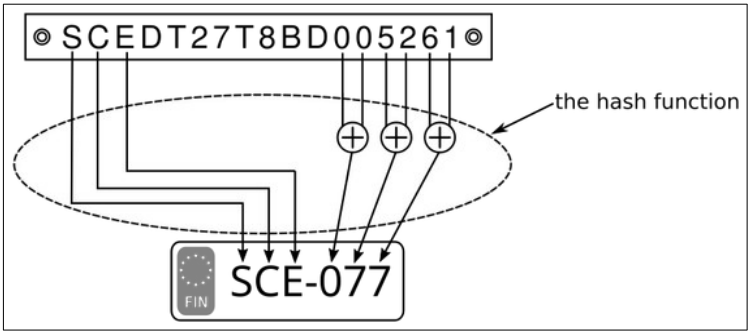


Figure 1: An example of a hash function

The fingerprint of a file

Sometimes a hash of a file is described as a “fingerprint”, in that it’s often a lot smaller, but it is almost uniquely linked to the file. Almost.

There is a very simple proof that a hash function cannot guarantee uniqueness. Consider a hash function that outputs an eight-digit number for any input. That means that the outputs can range anywhere from 0 to 99,999,999, so there are a total of 100 million possible outputs. However, there have been over 100 million books published[27]. This means that if you fed each book in turn into the hash function and were amazingly lucky in that when processing the first 100 million books there were no two books with the same hash output, it is guaranteed that once you feed in the 100,000,001st book, it has to have the same output as one of the previous books.

However, the cryptographic hash functions that are used to produce fingerprints for files have much bigger outputs. For example, the RIPEMD-160 hash function outputs a 160-bit number, which means there are about 1.5×10^{48} possible outputs. Roughly speaking, that’s comparable to the number of atoms that constitute the Earth[28].

Another very popular blockchain cryptographic hash function is called SHA-256, and it has as its output a 256-bit number, of which there are about 1.15×10^{77}. For comparison, the current estimate as to how many particles there are in the entire visible universe is about 10^{80}, so that’s one distinct output for every thousand particles we know of (or rather, can estimate the existence of[29]). In any case, they’re both numbers that are so big that they make the USA’s budget deficit of $3 trillion look like pocket change, even if you count it in cents[30].

Although there may be a lot of 256-bit numbers, any individual number can be represented very simply using a 16 by 16 grid, as in figure 2. When represented as such, it does start to look a bit like a fingerprint, or perhaps four chess boards glued together in a bigger square.

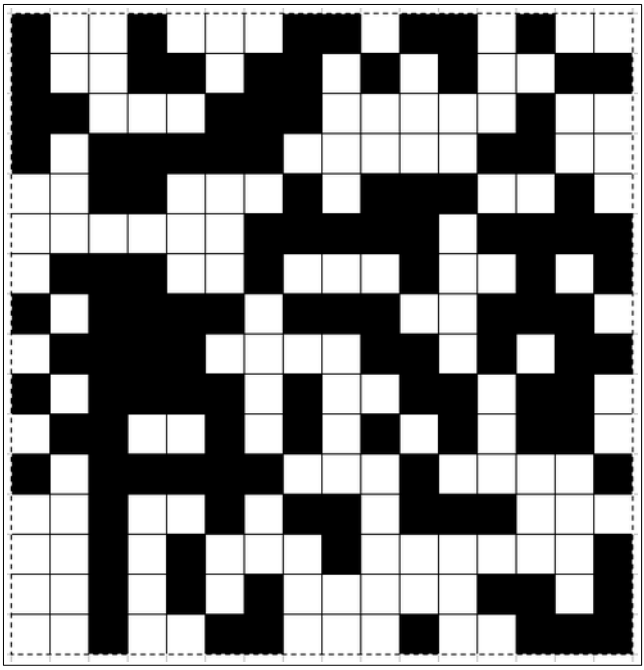


Figure 2: Grid representation of a SHA-256 hash output

Cryptographic hash functions

In the previous section, I subtly switched from talking about "hash functions" to "cryptographic hash functions". To qualify as the latter you need to have a few extra properties. First, let's recap what we have so far:

1. A hash function takes as an input some data of any size (even empty data), and invariably produces an output of a fixed length,
2. Outputs are deterministic, which means that the same input gives the same hash output time after time, even if you perform the hashing again days, weeks, or years later.

3. Collisions (identical hash outputs for two different inputs) are rare.

Cryptographic hash functions also require that:

4. When given a hash output, it is practically impossible to produce an input that when hashed returns that output. Cryptographic hash functions are meant to be irreversible. That's why they are also sometimes known as one-way functions or trapdoor functions.
5. It should be practically impossible to find two different inputs that give the same output (this is kind of a combination of items 3 and 4 on this list, but there is a subtle difference that should become clearer later).
6. Two very similar inputs should, generally speaking, produce very different hash outputs. For a cryptographic hash function to work that way, it has to act in what seems like a random fashion, without actually being random. Instead, cryptographic hash functions should be pseudo-random. See item 2 on this list.

Some papers and textbooks also list a further property, namely that it should be fast and easy to compute the hash output from an input[31]. However, that is not always the case. Fast and easy is a relative term, and for some uses, especially in blockchain, we want the computation of a hash to take a bit of effort. There will be more on that later as well.

Magical elves under toadstools

The ideal cryptographic hash function would be the algorithmic equivalent of a magic elf sitting under a toadstool, with a notebook, a pen, and a coin. We will call him ELF-256.

ELF-256 sits there patiently waiting for someone to pass him a message. On receiving the message, the elf springs into action infinitely quickly (he's a really fast elf):

He checks in his notebook to see if the message is already in there.

- If it is not, he writes it down in his book, and tosses the coin two hundred and fifty-six times, writing down the result after the message for each coin toss: a 1 for heads, and a 0 for tails.

  Then he checks to see if that particular sequence of 1s and 0s has already previously been generated for an earlier message. If it has, he crosses it out and repeats the coin tosses again, and if necessary, again and again until he is sure a new unique number has been generated for the new message.

- If, on the other hand, the message is already in the book, he looks up the number he wrote down after it the previous time he saw it.

  In either case, he then pauses for a fixed duration of time – perhaps a microsecond – and then reads back the list of ones and zeros that are jotted down in his book to the person who handed him the original message.

Oh, and finally, ELF-256 jealously guards his notebook with magic, so no one else can see what is written on the pages.

Why is ELF-256 a perfect cryptographic hash function? We can backtrack through our list of properties and see exactly why:

1. The elf always reads out exactly 256 digits, regardless of the length of the message passed to him.
2. If the same message is passed to the elf sometime later, he will read out the same number that he generated the first time he saw the message.

3. The elf never uses the same output number twice, so there can never be any collisions. Well – there can be, but only once he has generated 2^{256} outputs, which as I previously demonstrated is an almost unfathomably large number.

4. Imagine you are trying to find a message the elf responds to with a sequence of 256 ones. The chance of the elf tossing that many heads in a row is 1 in 2^{256}. Given that the elf takes one microsecond to respond, the odds are that it will take about two thousand vigintillion years*. For comparison, the universe has only been in existence for about 14 billion years. In short: it's not going to happen.

5. The elf does not reuse output numbers, so two different messages can never have the same output.

6. The outputs are generated randomly, so there is no relation between the original messages and the outputs. As a result, two very similar messages are going to get two very different responses from the elf.

Cryptographers who work on designing cryptographic hash functions are trying to come up with functions that behave as well as ELF-256, but competing with magical elves is a difficult task.

Oh the wonderful things that a hash can do

If you have paid enough attention over the last few pages, you should have a decent enough understanding of how cryptographic hash functions work and be able to appreciate how amazingly useful these things are in all sorts of areas of computing. We have already looked at the fact that you can take a confidential document, put it into a cryptographic hash function like SHA-256, and then make the two hundred and fifty-six bit output number public knowledge. At a later date, you can produce the original document, and anyone can

* A vigintillion is a 1 followed by 63 zeros.

hash it with SHA-256 to check that it is indeed the real deal, and has not been tampered with. If you published the hash output in the notices section of a newspaper, you even have a time-stamped record of when you had the document in your possession.

And the best and most interesting uses, in my opinion, are when you start hashing the output of previous hashes. What that means is that you feed an input into a cryptographic hash function, and then you take the output you get and put it back into the hash function again. And sometimes again and again. In the next four subsections, we will have a closer look at some of those applications.

One time hash pads

Up until a couple of years ago, Finnish banks used “code cards” to allow their customers to securely log in to their online bank accounts. The card would have a list of a hundred different four-digit numbers, and each time you used one, you would cross it off the list, and the next time you logged in you would use the following unmarked number. When you ran out of numbers, the bank would send you another card in the post. Now that they’ve finally switched to authenticator apps, I’ve got a bunch of them sitting in the drawer of my desk that I should probably throw away, but I’m waiting until winter to burn them in the fireplace. You never know if they might still work, after all.

Such cards are known as one-time password pads, and they are usually constructed randomly. However, there is another way to make them using the repeated application of cryptographic hash functions.

In figure 3, I show what happens when the message ‘the quick red fox jumped over the lazy brown dog’ is passed into the SHA-256 hash function, and the output is recorded in the next row of the table. Each row that follows contains a hash of the previous row.

I’ve entered the numbers in hexadecimal to make sure they fit in the table. As modern humans, we tend to use decimal notation for numbers (our digits are 0 to 9, presumably because we have ten

fingers), but note that not too long ago the Romans used letters like I, V, X, C, and M instead. So there's nothing special about the way we choose to write numbers today. Hexadecimal is a way of writing numbers if you were fortunate enough to be born with sixteen fingers (the letters a, b, c, d, e, and f are used to represent the extra digits), and as it happens, computers like to pretend that they have sixteen fingers. But don't worry about it too much. Just think of it as writing numbers in a slightly different way.

Back to the story at hand: each time the output is fed into SHA-256 again, to get another number, and so on for seven more goes. You can keep hashing outputs as long as you like, to make a longer and longer one-time hash pad, depending on how many times you think it will be used.

You can try it for yourself, as there are plenty of online SHA-256 hash generators[32]. One thing to note: your first input is text, but the output is a hexadecimal number, so when you copy/paste it into the input box, you need to select the input type to be hex.

You may be wondering, what is the point of all of this? How does it work? To start with, you need to get the last entry in the table to the entity that you want to authenticate yourself to (and there is more on authentication in the chapter titled "Error: Reference source not found" on page Error: Reference source not found). And then, whenever you want to prove to that entity that they are talking to you rather than someone else, you give them the entry in the table just before the one they already have.

the quick red fox jumped over the lazy brown dog
dd77f952e29e4a64c2bf5e27993fd2ebf1b0f378237abd299239bb4454d028b8
355a1ef19cb2e02ca528f60a8d9dfe533cb37e0b52ace3fabf960bd710904fe5
69c3610d92b04e41ef0937b27b06bfbea7e55f7716dd534e354f99041e8fff72
324e14abb32ee21573b5c1bdd7d4dff1d485c0ab030bd2d4c9780d5858c08dda
1c554d67a79a27b12a43d7e3e2b212f468312541cefda684c7678ea819745b24
5470bbf5478dc15bb25cd095bffb09bed25abb20063fc7e914d7e28ab80ac40f
665593ea8363949abcd3208a8470cd35be42f30809fd2ef8254244c585161136

Figure 3: A hash function-generated one-time password pad

So what happens then? Well, they take the password you presented them with, and simply hash it with the SHA-256 function. If the result is the previous password that they already have, then they know it is you. Do you remember cryptographic hash function property number 4? It's pretty much impossible to create an input that, when hashed, gives a specific output. As a result, it's more than reasonable for the bank to believe that the person who gave them the precursor number to the output they previously had is the same person. And that's the point of identifying yourself with a one-time password pad: to prove you are the same person they were previously talking to.

Hash linked lists

Imagine you are the proprietor of a popular newspaper. Every day your presses print out a new issue with the company masthead, the date, and the news of the day.

And then one day you discover that an eccentric billionaire has decided to take one of your earlier editions from two months ago, change one of the stories so it says something completely opposite to what the genuine issue actually said (perhaps it is even libelous), and has printed millions of copies. And not only that, but he then paid an army of highly trained ninjas to break into all the houses of

your loyal subscribers in order to replace their cherished copies with the false ones.

That may sound like a far-fetched scenario, but in the online world it is actually not that hard to achieve – printing digital copies costs virtually nothing, and hacking allows a single highly skilled computer ninja to break into many digital accounts. How can you protect yourself against this scenario?

Cryptographic hash functions to the rescue! These functions allow us to produce something called a “hash linked list”. If each edition of the newspaper contains a paragraph or notice that includes a hash of contents of the paper published the day before, then the eccentric billionaire has a much more difficult task on his hands. Why is that the case?

Because your newspaper history is now a hash-linked list. If a paper from sixty days ago is altered, then its hash is going to be different. That means that the paper from the next day needs to have the notification containing that hash altered too, which alters the hash of that paper. And so on, and so on, through all the papers right up to today.

Suddenly our eccentric billionaire doesn’t just have to make a change to the article that he disliked from two months ago, but he has to change every single newspaper from then on. That’s a lot more work, and furthermore, with another application of cryptographic hash functions, namely “proof of work”, we can make it even harder for him, to the point where he might as well not bother.

Proof of work

The fact of the matter is that in the scenario presented in the previous section we made it a little bit harder for the eccentric billionaire to subvert our newspaper, but we didn’t make it impossible. Instead of simply replacing one issue of the paper, our hash-linked list means that he has to replace sixty issues (or more, if

the story that he hated was further back in the past). That's a bigger pain in the neck, but it can still be overcome.

This is where "proof of work" properly enters the story. The concept was first invented by Dwork and Moni in 1993 as a means for combating spam emails[33]. Jakobsson and Juels[14] then coined the term "proof of work" to describe the idea and expanded on it, and Hal Finney adapted it to enable the creation of tokens backed by proof of work[34], the precursor to Bitcoin.

Satoshi Nakamoto's insight was to apply that concept to hash-linked lists in order to make it harder and harder over time to rewrite the past, and to put the whole thing on a peer-to-peer network, thereby allowing anyone to view the data in the list, verify that it is correct, and even assist in ensuring so much work is done when adding more data that no one person can go back and re-write it.

In other words, with a historical record stored in a hash-linked list that is secured by proof of work, the further back in the past the alteration you want to make is, the harder it is to perform the rewrite. You have to redo all the work from the point that you want to change, all the way through to the future.

So how does proof of work actually work in practice? Look back at figure 3 - all those entries below the passphrase "the quick red fox jumped over the lazy dog" are numbers. The fifth number starts with 1, and it's therefore the smallest number on the list. That is what proof of work is aiming at – finding a hash output that is below a target level; one that is small.

Hang on, if I hash the phrase, and I get a big number, then that's it. How am I going to get a better cryptographic hash output? The answer is that you have permission to add some "junk" at the end of the phrase. In cryptography, this is called a nonce, which is short for "number used only once".

Instead of feeding the output of the hash function back into the SHA-256 hash function, as we did in the previous section, let's feed our original sentence in again and again with different endings, as shown in figure 4, until we get an output that starts with 0:

This input's hash:	Starts with:
the quick red fox jumped over the lazy brown dog 0	3eec2267...
the quick red fox jumped over the lazy brown dog 1	30a16983...
the quick red fox jumped over the lazy brown dog 2	Be2cf376...
the quick red fox jumped over the lazy brown dog 3	12e08101...
the quick red fox jumped over the lazy brown dog 4	729107f6...
the quick red fox jumped over the lazy brown dog 5	297e5ae2...
the quick red fox jumped over the lazy brown dog 6	9e24b096...
the quick red fox jumped over the lazy brown dog 7	3f938d83...
the quick red fox jumped over the lazy brown dog 8	0ca85b7d...

Figure 4: SHA-256 hashing of a phrase with an added nonce

That didn't take long – only nine tries. Because the SHA-256 cryptographic hash function acts as a random number generator, on any individual attempt the odds are 1 in 16 that we will get an initial digit of 0 (remember, in hexadecimal, there are sixteen possible digits). That means that we have a 50% chance of finding a suitable nonce to go with the sentence within eleven goes. Of course, we could be unlucky and have to try hundreds of times, or we could be lucky and find a suitable nonce in one go.

What if we are looking for an output that starts with two zeros? The odds for an individual attempt are 1/16 × 1/16, or 1/256. To have better than even odds of finding the suitable nonce we are going to need to try 178 different nonces, so it's probably going to take more work. And if we want to find an output with eight leading zeros, well for any individual attempt we have a chance of less than one in four billion that we will luck out and find it. That means to have a better than 50% chance of finding a suitable nonce, we are going to have to try hashing over three billion times.

In other words, the smaller the hash output is required to be, the more work needs to be done to find it. If your computer can perform a SHA-256 hash in 1 microsecond, and I refuse to read an email you send me if it doesn't hash to a number with eight leading zeros, on average it is going to take you about 50 minutes to find a suitable nonce to add to your message that satisfies that requirement. And it only takes me 1 microsecond to check that you have done the work.

And that is how proof of work could be used to prevent spam. Similarly, returning to our newspaper example, an individual issue can be made harder to forge by requiring that a nonce is added to the last page, such that the full contents of the newspaper, when hashed with SHA-256, produce an output with eight leading zeros. Now our eccentric billionaire would have to do fifty hours extra hashing work if he wanted to change an issue that came out sixty days ago. But the downside is that the newspaper office is delayed by about an hour each day when issuing the next paper because they have to do the extra work too. Fortunately, they get to do it day by day, rather than in one big go.

Merkle trees

We've looked at repeatedly hashing a passphrase to generate a one-time password pad, hash linking lists of blocks of data by including a hash of the previous block in the next block, and making people (or rather their computers) prove that they have done some work by challenging them to add some extra data to their messages, which causes them to hash to a low number output. What else can be done with cryptographic hash functions? I will start with an analogy:

Imagine a fictional country, where all the residents are totally obsessed with the law, and the laws that are passed by the parliament or that arise from case history out of court decisions are gathered up and bound in special books.

The funny thing is that the contents of those books then become the absolute law of the country, which means that if somebody can tamper with one of them and add a rule that says that "people named Gary are allowed to conduct bank heists with impunity", then anyone called Gary is allowed to rob banks without fear of prosecution. Criminals could tamper with the laws in order to get away with their crimes!

The residents have come up with a very clever scheme to protect these books from such tampering without it requiring an

immense amount of effort. The first thing is that they have an amazing machine that you can feed the text of a book into and it produces something similar to an ISBN number as an output that you can stick on the book, but the ISBN number that is produced is unique to each different form of text. It’s a cryptographic hash function machine. If you take a book and you change one letter and then put it into this machine, you get a completely different ISBN code out.

The second thing about it is that you can't engineer a book to give a specific ISBN code - they're almost random in that respect. Again, evidence that the machine is a cryptographic hash function.

What the inhabitants of this country could do is to put all of the books in a library, and then gather all the texts from all the books, feed them into the machine, and make one single code for the whole library. That would ensure that in the future if tampering was suspected, they could repeat the whole arduous process, and if the second code matched the first one, they'd know that all the books were intact and no one had messed about with them.

But this is terribly inefficient – it means that every time you suspect a single letter or word in one book has been changed, you then have to go and collect all the books in the whole library from all the different rooms in order to ensure everything is okay.

Instead, they have taken the following approach:

- They group books into pairs. Each book is run through the machine and gets its own code, which is stuck on the back of the book, and then the two books are put on a shelf.
- The pair of codes off the back of the two books are concatenated (that means written one after the other) and that new concatenated code is fed into the machine to get yet another code. The shelf is labeled with that code.
- Each bookcase has two shelves. They repeat the process with the codes on those shelves: they take a copy of the codes of the front of the shelves, concatenate them, and put

them in the machine to get a new code, which they stick on the bookcase.

- They have two bookcases in each room. Again, the same process is conducted with the codes on the bookcases, in order to get a label for the room's door.
- And there are two rooms on each floor of the law library so in order to label the floor, they use the codes from the two doors and then finally they put a label on the front of the law library that is the codes from the two floors concatenated and run through the machine.

I’m sure you understand what the process is by now. but you may be wondering, “Why are they doing this?”

If there's a court case, and it requires the laws from book number three, the court officials don't have to go and gather up all the other books to check that just the relevant book has not been changed. They take book number three and check that its code, or hash, matches. That means running one book through the machine, and verifying that the output matches the sticker on the back of the book. Now somebody could have gone in and changed book number 3, then run it through the machine and put a new code on it. But what the court officials can now do is look at the number on the other book next to it, check that's okay, and that together they hash to the shelf label.

Of course, somebody could have changed the shelf label as well, but then the court officials take the shelf label below and they concatenate and check again, and then they check that the two bookcase labels give the room number and that the two room numbers on the floor give the floor number and finally that the two floor numbers give the number on the front of the law library. It still sounds like a lot of work, so let’s put it in a list:

1. Check that book codes hash to shelf code,
2. Check shelf codes hash to bookcase code,

3. Check bookcase codes hash to room code,
4. Check room codes hash to floor codes,
5. Check floor codes hash to library code, and
6. Check library code hasn't changed.

That's a total of six basic operations instead of collecting all 32 books (did you work out how many books a library of this type can hold?) in order to ensure that the law in one book is still correct.

There is one final piece, namely ensuring that the code on the front of the library is correct, but as it is displayed out in public on the street (and perhaps they have security cameras pointed at it too), the chances of someone called Gary making all those changes and then getting up on a ladder in broad daylight without being seen are slim indeed.

What I have described is called a Merkle tree (because it was invented and patented by Ralph Merkle[35]), and it should be pretty obvious that in blockchain systems such as Bitcoin the books correspond to transactions. The code on the front of the library, which in cryptographic terms is called the *root* of the Merkle tree, is stored in the block header to ensure it cannot be tampered with, making the library the equivalent of a blockchain block.

These days it is a very common practice to use Merkle trees to allow people to verify a snippet or chunk of data has remained intact and unaltered, but without having to download lots and lots of other irrelevant transactions. You only have to check a single branch of the Merkle tree, rather than the whole tree. As a result, Merkle trees are used in such places as the peer-to-peer file-sharing system BitTorrent, the anonymous communication network Tor, the distributed version control system for software development called Git, and of course: blockchains.