Vanderbilt University Department of Economics Working Papers 19-00013

Encryption, Hashing, PPK, and Blockchain: A Simple Introduction

John P. Conley
Vanderbilt University

Abstract

Blockchain, SSL certificates, HTTPS, cryptocurrencies, public/private key pairs, VPNs, and many other important technologies are all applications of two basic ideas from cryptography: encryption and hash functions. This paper explains these two ideas and shows how they have been deployed in different combinations to create applications that drive the Internet and technologies that will shape our economy in the future.

The author takes sole responsibility for the content of this paper.
Citation: John P. Conley, (2019) "Encryption, Hashing, PPK, and Blockchain: A Simple Introduction", *Vanderbilt University Department of Economics Working Papers*, VUECON-19-00013.
Contact: John P. Conley - j.p.conley@vanderbilt.edu.
Submitted: August 17, 2019. **Published:** August 17, 2019.
URL:http://www.accessecon.com/Pubs/VUECON/VUECON-19-00013.pdf

Encryption, Hashing, PPK, and Blockchain: A Simple Introduction

John P. Conley[1]

Vanderbilt University

August 2019

Abstract

Blockchain, SSL certificates, HTTPS, cryptocurrencies, public/private key pairs, VPNs, and many other important technologies are all applications of two basic ideas from cryptography: encryption and hash functions. This paper explains these two ideas and shows how they have been deployed in different combinations to create applications that drive the Internet and technologies that will shape our economy in the future.

[1] j.p.conley@vanderbilt.edu. The author takes sole responsibility for the content of this paper.

Introduction

Blockchain, SSL certificates, HTTPS, cryptocurrencies, public/private key pairs, VPNs, and many other important technologies are all applications of two basic ideas from cryptography: encryption and hash functions. Although they are mathematically related, they are intended to do very different things.

> **Encryption:** Data is encrypted in order confine meaningful access to authorized users. An encrypted file might be publicly available, but without a key of some sort, the encryption cannot be reversed and the file is unreadable. Examples of encryption algorithms include AES, DES, Blowfish, and RSA. An encrypted file is refereed to as **ciphertext**, while the unencrypted file is refereed to as **plaintext.**
>
> **Hashing:** Data is run through a hashing function to generate a kind of digital fingerprint that is essentially unique to the data file.. The point of hashing is not to hide data, but to allow verification that the data has not been tampered with in any fashion. A hash of a file cannot be "unhashed" back into the original file.

Encryption

Encryption is meant to keep private files private. Data can be encrypted **at rest,** or **in transit**, and both are important. If you encrypt your home directory on your computer, then hackers will find it difficult or impossible to access your private files. However, if you take a file off your disk, decrypted it, and send it to a friend as an attachment via email, then the file can be read if it is intercepted. Similarly, if you download a file over an encrypted connection but then store it on your disk in plaintext, anyone with access to your hard drive can read it.

At the most basic level, encryption transforms a message or data file using an openly available algorithm.[2] The classic **Transposition Cypher** is the simplest example. The idea is that each letter in the alphabet is mapped to a different letter. For example, the alphabet could be transposed one position so that *a* becomes *b*, *b* becomes *c*, and so on. Thus, the plaintext "Local Zoo" becomes the ciphertext "Mpdbm App".

Symmetric Encryption

Modern cyphers are more sophisticated, but at root, use a similar approach. Plaintext is subjected to a series of substitutions and permutations determined by a key. A **key** is a string of bits of specified length. For example, **AES 256 (Advanced Encryption Standard 256)** uses a 256 bit key which means that there are $1.2 x 10^{77}$ different ways that the substitutions and permutations might be executed on plaintext to produce ciphertext. AES-256 is seen as a completely secure encryption system given current computer technology.[3] AES-256 also has the advantage of being

[2] Methods of encrypting data are public information. Protecting encrypted files depends on keeping the key or password secret, not the encryption algorithm itself. You might wonder why a file cannot be decrypted by simply inverting the publicly known encryption function. The reason is that the these algorithms use something called a "trapdoor function" which cannot be inverted.

[3] The only way to decrypt a file is to know the right key. Breaking encryption therefore requires that the key be discovered by brute-force guessing. Suppose that a hacker had enough computational power to make 10^{12} (one quadrillion) guesses per second. This means that the hacker could make about 3×10^{19} guesses per year. To have a 10% chance

computationally efficient, meaning that relatively few computer clock cycles are needed to decrypt each byte of the ciphertext if the key is known.

Although AES is a secure system, it depends on the sender and receiver sharing knowledge of a common key (sometimes called a "**shared secret**"). This is called **private key encryption** and is form of **symmetric key cryptography**. How can users agree on a key while keeping it secret from everyone else? Obviously, sending the key in an unencrypted form exposes it to interception. On the other hand, the sender cannot send the key to the receiver in an encrypted form unless the receiver has the key needed to decrypt encrypted key. We have a chicken and egg problem.

There are several ways to solve this. For example, users might meet in person and agree upon a key. An even more sophisticated approach would be for users to meet in person and agree upon a book and an algorithm such as: "Use the ASCII binary representation of the first 32 characters on the n^{th} line of page x where n is today's date, and x is a page number to be sent in an open email". Unless the book the two users choose is known to others, the page number is useless information. This system allows the sender and receiver to securely generate a new key each time they communicate.

Asymmetric Encryption and Public Private Keys (PPK)

Both of these solutions depend upon users having at least one completely secure, unencrypted, exchange of information. This is impractical if users do not know in advance that they may wish to communicate securely. For example, I may wish to send a secure email to someone I have never met, or give a credit card number to a merchant I have never used before.

Where it is impractical for users to meet securely and agree upon a shared secret, **public key encryption,** which is a form of **asymmetric key cryptography,** can be used instead. The drawback is that decrypting text without a shared key takes more computational effort. As a result, public key encryption is often used only to begin a secure communication session in order to send a symmetric encryption key to be used by both sides for the remainder of session.

The real magic of public key encryption is that the public and private keys are special mathematically relationship. Not only is the private key the one and only way to decrypt a message encrypted with complementary public key, but the *public key is the one and only way to decrypt message encrypted by the complementary private key.* This symmetry will turn out to be essential to blockchain, SSL certificates, and many other applications.

At the highest level, public key encryption works like this:

1. The receiver generates two large numbers. One is called a **Public Key**, and is made openly available. The other is call a **Private Key** and is kept secret by the receiver.
2. The sender uses the receiver's public key to encrypt his message.

of guessing which of the 1.2×10^{77} possible keys was correct, the hacker would have to test 1.2×10^{76} keys. This would take approximately 4×10^{56} years. As far as we know, no one has this much computational power. In any event, the universe is only about 13.7×10^{9} years old. Thus, while it is not impossible to break AES encryption in theory, it would take more computing power than is likely to exist in the foreseeable future and require many orders of magnitude longer than the universe has existed to do do. This is why the National Institute of Standards and Technology (NIST) says that it is computationally impractical to break AES-256 encryption, although this may change as new quantum computing based decryption techniques are developed.

3. The receiver uses his private key to decrypt the message.

To summarize:

- Data can be encrypted at rest, in transit, or both.
- A symmetric key can encrypt messages or documents such that anyone who has the key can decrypt them at low computational cost. Without the key, decryption is computationally impractical.
- Public/private key pairs are mathematically entangled numbers which can mutually decrypt ciphertext created by the other. It takes more computational effort to decrypt ciphertext created this way which is why public key encryption is generally used only to agree on a shared symmetric key for communications.

Applications of Encryption

We have already discussed how symmetric and asymmetric encryption are used to send secure messages between users. The same basic technique is behind VPNs, HTTPS connections, encrypted WiFi communications, and SSL and TLS connections. One other application deserves mention.

Cloud Storage

Cloud backup services like OneDrive and Dropbox, and enterprise applications built on Azure or AWS, hold sensitive client data and use encryption to keep it secure. A common approach is to use a shared key to encrypt data that is transmitted between a client's computer and the cloud service. This encrypts data in transit, but not at rest. If either the client or cloud service is hacked, the data is revealed. A better approach is to use a symmetric private key to encrypt data end-to-end. That is, encrypt the data on both the client's computer and the cloud provider's server, and only transmit data in encrypted form. This encrypts data in transit and at rest on both the local and cloud disks. The problem with this approach is that if the cloud service knows the shared encryption key, it can read your data. Even if you trust the cloud provider, a hacker may find a way to steal the encryption keys or a government agency may compel the provider to turn over your unencrypted data. Thus, an even better approach is **client-side encryption** which keeps the key in hands of client instead of making it a shared secret with the cloud provider. This key is used to encrypt data on the client's local disk and only encrypted files are sent to the cloud server. Since the cloud provider does not know the private key, it is unable to read the client's files. If the cloud provider is hacked, only encrypted files without keys are compromised. Even if the government ordered the cloud provider to turn over client data, it could only give up encrypted files. Since the cloud provider does not have the key, it is technologically impossible for it to turn over cleartext client data.

Hashing

Hashing is a method used to verify the integrity of a message or file. The **hash function** itself is public knowledge and does not need any kind of key to work. When a message or file in run through a hash function it produces a fixed-length output string. The Secure Hash Algorithm-256 (SHA-256) is one widely used approach and has the following properties:

- Running any file, regardless of length, through SHA-256 returns a 256 bit binary string.
- Very similar files produce quite different hashes in an unpredictable way. In fact, SHA-256 is designed so that hashes of different files are, in effect, randomly and uniformly distributed[4] over the set of all possible 256 binary strings of which there are 2^{256} or about 10^{77}.
- The hash of any file is unique. The same input always produces the same output. For this reason, the hash is sometimes referred to as a file's "fingerprint".
- Although hashing a file always gives the same result, two files may have the same hash. This is called a "collision". In practice, however, this extremely unlikely to occur.[5]
- Hash functions are noninvertible. It is impossible to recover a file from its hash.

Applications of Hash Functions

Verifying Documents

Suppose you sign a partnership agreement and later on have a dispute. You and your partner both present copies of the contract to a judge, but they say different things. How can the judge determine which is genuine? Suppose the judge had access to a hash of the contract as it was on the day it was signed. He could then compare it to hashes of the contracts presented to him by you and your partner. The genuine contract will produce the same hash, and any altered contract will produce something else. You may ask how the judge might have access to a hash of the contract he knows was taken on the day it was signed. Read on about digital signatures and blockchain below.

Verifying Logins

When a user tries to log into a system, the system compares the password the user provides to the one stored in its files. The problem with this is that the passwords stored by the system might be stolen by hackers. Thus, it is good security practice to store only hashes of user credentials. Users still type in passwords, but the system immediately takes a hash and compares it to the hashed password stored in its files. If the password is correct, the hashed password will match a stored hash. If the hashed password file is stolen from the system, it is of no use. The password cannot be determined from the hash, and the system requires a correct plaintext password to grant access, not the hash itself.

Securely Resetting Passwords

Sometimes when you forget a password, you can request that it be sent from the website to your verified email address. This is not the most secure practice since email often transits the Internet in

[4] Okay, not randomly; hash functions are deterministic. However, if you took a file, modified it one bit at a time, and then looked at the distribution of the resulting hashes, they would be approximately uniformly distributed between `0...0` and `1...1`, where these are each 256 bit long binary strings.

[5] Collisions must occur in theory. A typical MP3 file is tens of millions of bits long. Obviously, it is impossible to do a one-to-one mapping between the set of every possible 10M bit sting and every possible 256 bit string. However, If I took a hash of any file and then started looking for another file with the same hash, I could test $10^{77}/2$ files and still have only a 50% chance of finding one. Thus, it is extremely unlikely that you will every actually see a collision.

plaintext and can be stored on any intermediate server that passes the message along. However, if a website follows the practice of only storing hashes of passwords, it literally cannot send a user his password since it does not know it itself. This is why you sometimes get a message asking you to click on a one-time link that quickly expires and and choose a new password. This password is immediately hashed and stored. This approach relies on three things to make it secure: that the email address has previously been validated, the use of two part authentication in many cases, and the fact that a hacker would have to intercept and use the link immediately while having access to your email account.

Privacy Preserving Data Matching

Suppose we all joined a social network and wanted to know if any of the people in our address books had also joined. None of us, however, wanted to reveal our own email address or the contents of our address books to the network. We could instead submit a hashed version of our address books. The network would be able to see that one person's email address was in another person's address book by finding identical hashes. However, the network would not be able to read the email addresses themselves and so user privacy would be preserved.

Efficient Database Search

Suppose we had a database that contained names, email addresses, email message texts, documents, audio recording, case histories, and other large and dissimilar items. Suppose we wanted to find the record that contained a specific document or email message. Running a standard search looking an identical document in the database would be computational intensive since it would require comparing data elements that are many kilobytes or megabytes long against the document we wanted to find. We could instead create a database that contained hashed versions of the all the data elements and search for a match with a hash of the document we were looking for. This would require comparing data elements that are only 256 bits long. The bigger the average size of the original data elements, the greater the computational savings from searching for matching hashes.

Applications that Combine Encryption and Hashing

Really interesting things become possible when encryption and hashing are used together.

Digital Signatures

Signing paper documents is the traditional way of indicating agreement or acknowledging receipt. Signatures can be forged, so banks and notary publics require that people show identification documents such as passports or driver’s licenses. These documents often include photographs, signatures, physical descriptions, and even fingerprints. In effect, these documents are an **attestation** by a government agency or whoever issued the documents that it believes that the person named on the ID document is the same one in the photograph, uses a signature that looks like the one on the document, has a certain fingerprint, and so on.

There are several weaknesses to this approach. First, we have to trust in the truthfulness and due diligence of the document issuer if we are to believe its attestation. This is why banks ask for official government IDs instead of your work ID or AAA card. Second, the document could be forged. Third, the ID could be stolen. These last two problems might be solved if we could request a copy or image of the document from the issuing agency. This would make it immediately apparent if the document being presented is forged, altered, stolen, or revoked.

In the digital world, public private key (PPK) pairs combined with hashes can be used to sign documents, messages, transactions, and any other type of digital object. Signing and verifying signatures works as follows:

1. The signer produces a hash of the document.
2. The signer encrypts the hash with his *private key.*
3. The signer attaches this encrypted hash to the unencrypted (cleartext) document
4. To verify the signature, the reader decrypts the hash using the signer's *public* key. The decrypted hash could only have been encrypted in this exact way by the holder of the complementary private key (that is, the signer). Thus, the reader knows that this is the correct hash of the document as it was signed by the private key holder.
5. Finally, the reader produces his own hash of the unencrypted document. If the hashes match, then he knows the document is exactly what was signed and has not been changed in any way.

For this to work, the reader must have access to the public key that can decrypt the encrypted hash, and believe that this public key belongs to a specific person or entity. The reader also has to believe that the owner of the public key had control of the corresponding private key when the document was signed. If a hacker managed to get his hands on the private key, he could use it to sign documents just as easily as the true owner. Thus, if we think the key has not been stolen or compromised, and we are confident that we know the real world person or entity who owns and controls the key, then we can be equally confident that that person or entity signed the document verified by matching hashes.

SSL Certificates

How can we verify that a private key has not been stolen or compromised? How can we be confident that we know the true real world owner of the public key used to decrypt the hash in a digital signature? Just like physical signatures, we need some kind of digital ID document to give us confidence that the owner of the public key is really who he claims, and that the corresponding private key is under his exclusive control.

We do this using the **PKI (Public Key Infrastructure)**. This is system of hardware, software, policies, protocols and participants which can issue and revoke **SSL Certificates (Secure Sockets Layer Certificates)**. An SSL certificate is a small file that can be attached to an email, requested from a web or email server, or from a server set up specifically to hold SSL certificates. SSL certificates are issued by a **CA (Certificate Authority),** such as VeriSign, Comodo, Globalsign, Google or Microsoft and consist of five basic elements

- The real world identity of the certificate holder.
- A serial number and expiration date for the certificate.
- The name of the CA issuing the certificate
- The public key of certificate holder.

- A digital signature from the certificate-issuing authority. This includes a hashed version of the *certificate holder's public key* encrypted using the *certificate issuer's private key.*

SSL certificates are used as follows:

1. A user obtains the holder's SSL certificate.
2. The user decrypts the hash of the holder's public key using the CA's public key (which is included in the CA’s signature on the holder’s SSL certificate).
3. If the decrypted hash matches the user's own hash of the holder's public key as included in the certificate, then he can be sure the CA has attested that this, in fact, is the holder's public key and identity.

Unfortunately, this just backs the identity verification problem up one level. If we are sure we know the CA's true public key, then we are equally sure that the holder's public key is the one in the certificate. But how can we be sure that we have the correct public key for the CA? The PKI based on what is called the **web of trust**. The CA who signed the certificate will also have one or more SSL certificates from other CAs specifying its public key. The issuers of those certificates will also have SSL certificates issued by other CAs, and so on. Provided we can find at least one CA in this chain we believe and trust, then the rest of the structure is verified. There is no central authority in the web of trust by design. If you cannot find CA in the chain for whom you believe you have a correct public key and whose honesty you trust, SSL certificates are of no help.

The most common use of SSL certificates is to verify that the websites you visit are genuine. When I type in the URL of my bank for example, how can I be sure that I end up at the right place? I might have clicked through a link on an email that sends me to the wrong place or uses a DNS that has been hacked and sends me to a fraudulent IP address. The SSL certificate allows me to be sure that I am talking to the right URL before I send my login credentials or any confidential information.

Blockchain

One of the most exciting uses of these technologies is blockchain. Blockchains come in many forms and variations, but at root, a blockchain is a time-ordered, immutable, verifiable, distributed ledger. Let’s break this down into parts:

Ledger: Traditionally, paper ledger books have been used to keep records of account balances and transactions of customers, ownership of property, marriages, births, deaths, and so on. Bitcoin’s blockchain is similar in that it contains a list of transactions in which some number of bitcoins is debited to one account (technically, a PPK address) and credited to another. Newer blockchains built on the Ethereum codebase can store documents. smart contracts, and other digital objects in flexible data structures, but these are just more complicated and useful types of ledgers.

Distributed: Blockchain ledgers are distributed in two senses. First, it is usually the case that information to be recorded in the ledger is submitted and processed by many different agents.

For example, any owner of bitcoins can send a request to transfer some of his holdings to another account using Bitcoin’s decentralized peer-to-peer network. Bitcoin's network has about 10,000 nodes that are supposed to independently verify these transactions by making sure that the transactions have correct cryptographic signatures and correspond to an accounts that have enough bitcoins to their credit to cover the requested transfers. Other blockchains are built around a trusted set of stakeholders such as a consortium of banks, who are allowed to submit information to be recorded. Second, up-to-date copies of the blockchain ledgers are usually maintained in several places. Copies are sometimes distributed only to stakeholders and are hidden from public view. Public blockchains like Bitcoin maintain ledgers that can be examined, copied, and stored by anyone.

Verifiable: A “block” in a blockchain is a collection of transactions or other data that accumulates over a set time interval to recorded as a unit. In the case of Bitcoin, blocks are written about every ten minutes and contain 1000-2000 individual transactions. These blocks are added to the the existing chain of blocks and contain enough data for anyone to audit and verify that the transactions they contain are valid, and the balances in the public key accounts are correct. Private blockchains are not auditable in general, but users with the correct permissions may be able to verify the correctness of the transactions that they are party to.

Immutable: At the highest level, it is difficult to rewrite transactions that are recorded in previously committed blocks (that is, blocks buried back in the chain) because of the way that they are cryptographically linked together. We will say more about this below.

Time-ordered: New blocks of transactions are created sequentially and appended, or committed to the end of the existing blockchain. Although this does not create a time-stamp that allows us to know exactly when the block was committed, it does tell us the order in which transactions were executed.

PPK Account Addresses

PPK pairs are at the heart of blockchain technology. Accounts kept on the ledger effectively belong to a PPK pair. Accounts are numbered or at least include a public key to indicate ownership. There is no record within the ledger of what human or other entity is associated with the account. Transactions from an account require a transaction request signed by the corresponding private key. Nodes use the public key in the account address to attempt to decrypt the transaction request, and if they are successful, they know that the requester has access to required private key. If an account holder loses his private key, it is impossible to access his account. The tokens are frozen forever and are effectively removed from the coinbase. If another agent gains access to the private key, he can use it to steal the tokens. From the standpoint of the validating nodes, anyone who can produce the private key owns the tokens in the account.

Hash Chains

Blocks are chained together using a kind of recursive hashing technique. The idea is ingenious, but very straight-forward. Blockchains begin with a genesis block (block 0) that contains a ledger of accounts defining the initial distribution of tokens. The first block could be constructed in any number of ways depending on the protocols the blockchain uses. However it is constructed, a set of valid transaction requests are grouped together as block 1 to be appended to the genesis block,

block 0. This is done by taking a hash of block 0 and including it in block 1. Block 2 is then built and a hash of block 1 included. In general, block B includes a hash of block $B-1$.

Now suppose that a chain has 1000 blocks and I want to go back and alter a transaction in block 600. Obviously, this changes the data in block 600 and so also changes the hash of block 600. But the hash of block 600 is included in block 601. Anyone who wanted to could take the hash of the altered block 600, compare it to the hash included in block 601, and thereby prove that the data in block 600 had been changed. The only fix would be to include the new hash of block 600 in block 601. But then the hash of block 601 contained in block 602 would be wrong. Thus, for the blockchain to have a consistent hash chain, all the hashes from block 600 to block 1000 would have to be recalculated.

What this buys us is two things. First, any attempt to alter data in the a blockchain is visible and provable hash chain would not be consistent. Second, it makes the blockchain "append only". That is, it is impossible to insert blocks or data into the middle of a blockchain without leaving evidence. Data can only be append the last block.

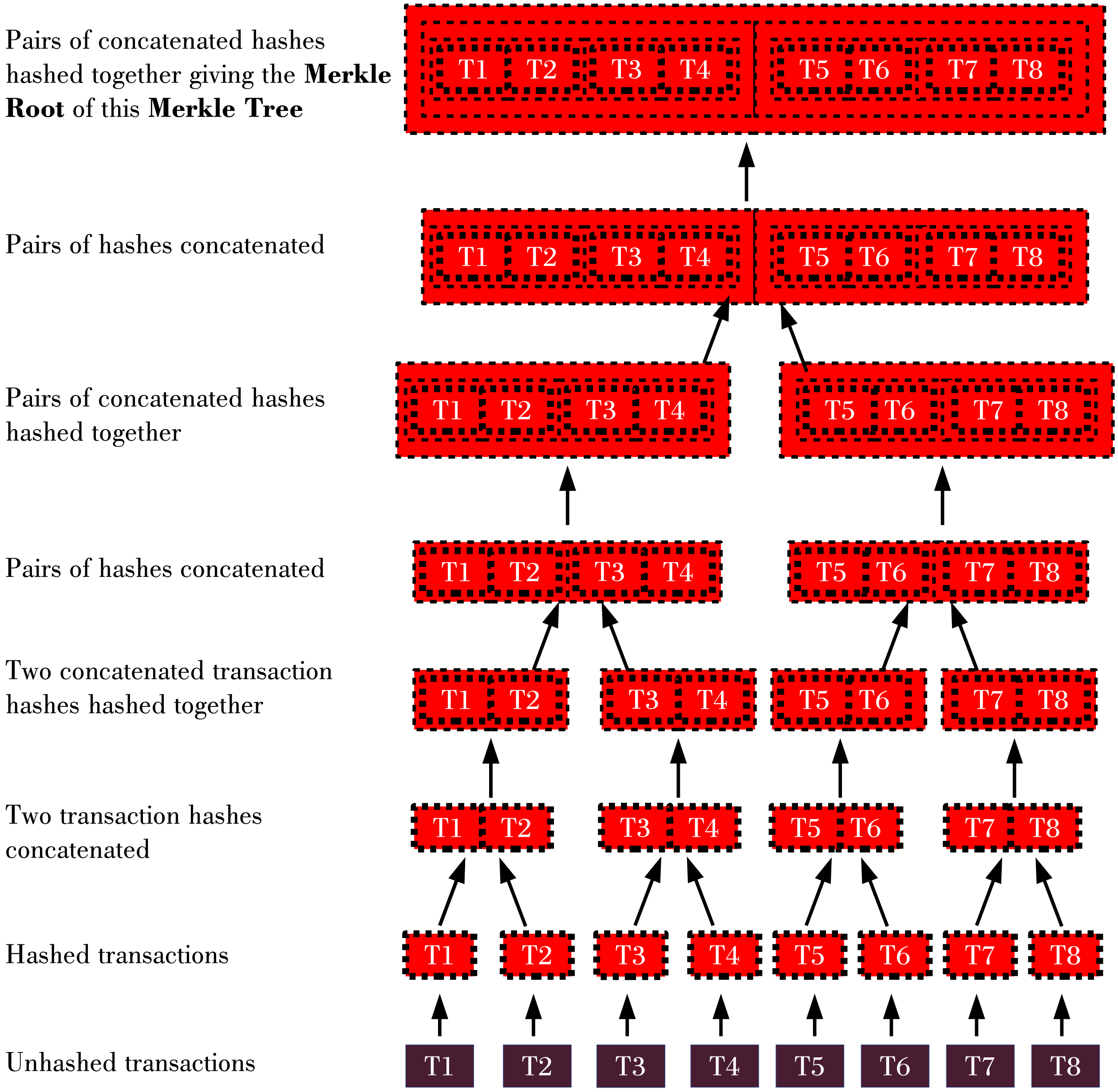
Both of these features depend on the knowledge that the hash tree has not been recalculated after data has been altered. If someone takes the trouble to recalculate all the hashes from block 600 onward, for example, he could erase evidence of his tampering. How can such a thing be prevented? We discuss this below.

Merkle Trees[6]

Hash chains are closely related to a more general notion called **Merkle Trees**. The idea here is that two data items (two individual transactions in a block, for example) are hashed individually and then joined together (more formally, **Concatenated**) and then hashed again. This joint hash is then concatenated with another joint hash of two transactions hashes. Not that if there are N transaction hashes, there are $N/2$ hashes at the next level, Pairs of these hashes are concatenated and hashed together which yields $N/4$ hashes at the next level. This process continues until there is only one hash called the **Merkle Root**. Together with all the hashes the came together at each level, this forms a Merkle tree with the root pointing to two parent hashes, each of which, in turn, points to two parent hashes of their own, and so on. See the figure below for an example.

Merkle trees are used to construct **Merkle Proofs** as follows. Suppose that I wanted to know if transaction $T3$ was in a certain block of a blockchain. I could just download the whole block and check, but if the block contained thousands of transactions this might require many MBs of bandwidth. I could instead download the block's Merkle root, the $[T5,T6,T7,T8]$,$[T1,T2]$, and $[T4]$ hashes. Since I know the clear text of $T3$, I could then take its hash, and then hash this together with the $T4$ hash to get the hash of $[T3,T4]$. Next, I could hash this with the hash of $[T1,T2]$ to get the hash of $[T1,T2,T3,T4]$, and finally I could hash this together with the hash of $[T5,T6,T7,T8]$ and compare the result to the Merkle root. Sounds like a lot of work, but really, this is only four hashing operation, which are computationally cheap. Even better, I only had to download four hashes, each of which is 256 bits or $32Bs$. Thus, I only would only need to download a total of $128Bs$.

[6] This a more advanced use of hashing in computer science, so feel free to skip it if you wish. It is used to verify the existence of transactions in blockchains, however, so it does have widespread practical value. We won't deal with unbalanced Merkle trees here for simplicity.


Pairs of concatenated hashes hashed together giving the **Merkle Root** of this **Merkle Tree**
Pairs of hashes concatenated
Pairs of concatenated hashes hashed together
Pairs of hashes concatenated
Two concatenated transaction hashes hashed together
Two transaction hashes concatenated
Hashed transactions
Unhashed transactions
T1
T2
T3
T4
T5
T6
T7
T8

An example of a Merkle Tree

To understand this better, suppose that the block included some $T3'$ which was different from $T3$ and so $T3$ was not in the block at all. The hash of $T3'$ would be different from the hash of $T3$, which means the hash of the concatenated hash of $T3'$ and $T4$ would be different from the hash you just took of the concatenated hashes of $T3$ and $T4$. This goes up the line to ultimately give a different Merkle root. The only way the Merkle root could agree with the one you just calculated is if the same $T3$ was used in its construction. Thus, if the Merkle roots agree, you have a Merkle proof that $T3$ is in the corresponding block.

A similar technique can be used to prove that a transaction is in a block anywhere in the history of the chain. A Merkle tree needs to be constructed of the blockheaders of each block to prove that the each block is in the chain the corresponds the current Merkle root of the blockchain. This Merkle tree is then connected to each transaction by hashing the concatenation of the each blockheader with the Merkle root of the transactions it contains. This makes it possible to get a Merkle proof that a transaction exists in a block, and then that block exists in the chain.

The appeal of this approach is that it is efficient for both a computational and bandwidth standpoint. Suppose that a block or any data set had 4,294,967,296 items (conveniently, 2^{32}) then you would only need download 33 hashes and perform 33 hash operation to prove that the data item was part of the data set. To see this, note that the number of branches on the Merkle tree doubles with each level. By the time each branch has split 32 times, there are 4,294,967,296 ends. Thus, proving one data items is part of a set of more than 4B items only requires a Merkle proof of 33 × 32Bs ≈ 1kB. As with hash chains, all of this relies on knowing that the Merkle root is correct and has not been reconstructed or rewriten after the underlying data was altered in some way. The next section discusses why this is difficult to do on a blockchain.

Proof of Work

The Bitcoin protocol was invented by Satoshi Nakamoto.[7] The cleverest thing in Nakamoto's protocol is the cryptographic puzzle he developed which is the foundation of Proof of Work. The idea is the following:

Suppose we took a hash of a block we wished to append to a chain. Recall that a hash is a 256 bit binary string that is effectively random. Thus, the probability that the first digits in the hash of any object is 0 (instead of 1) is 50%. The odds that the first two digits are 0 is 25%. In general, the odds that the first n digits of hash are 0 is $\frac{1}{2^n}$. For example, the odds that a random data object would have a hash of 00000???????... (five leading zeros) is one in 32.

Now suppose I took my block and I added some random data to it called a "nonce" and took the hash again. The hash of the block with the new data would also be a random 256 bit binary number. If I did this often enough, I would eventually find a hash that had three leading zeros. In fact, since the odds of this are one in eight. If I added random data four times and took the hash, I would have a 50% chance of finding a hash with three leading zeros.

In the case of Bitcoin, nodes engage in a "guess and check" procedure in which they add random data, take the hash, and see if they have found a nonce that generates a hash with n leading zeros.[8] For example, the odds of guessing at the nonce needed to generate a hash with 70 leading zeros is about one in 10^{21} . There is no way to find the nonce without going through repeated guessing, hashing, and checking. In other words, any node that finds the nonce must have expended considerable computational effort. There are no shortcuts. This cost is the foundation of the claim that the Bitcoin blockchain is "immutable".

A good way to understand the immutability claim for PoW blockchains is to think of each block of transactions as a page in a paper ledger book. This ledger book has three special characteristics.

[7] Satoshi Nakamoto is a pseudonym. The true author or authors of this paper are unknown.

[8] The actual Nakamoto PoW system is a bit more complicated, but this is the essential idea.

First, each page is made up of thousand dollar bills (or gold foil, or must be handmade from rare materials at high cost). Second, all transactions are recorded in indelible ink. Pages cannot be altered or reused once transactions are written on them. Finally, a tiny copy of the previous page is written at the top of the next page in the ledger (actually, this is the "hash" discussed above).

Suppose that I wanted to change a transaction recorded on a page 50 back from the most recent one. First, I would have to create a new blank page of thousand dollar bills to write my new transition on. But since this would change the hash of page, I would also have create 49 additional new blank pages so that the recursive hash tree was consistent. This implies that the more deeply a transaction gets buried under new pages in the ledger, the more expensive it is to alter it without detection. Thus, the ledger is not immutable, it is simply very expensive to mutate. Eventually these costs become prohibitive. Unfortunately, this security guarantee comes at a high cost. Each page costs tens of thousands of dollars to create, mostly in the form of wasted electricity.

Other Blockchain Protocols

Proof of Work is only one approach to verifying transactions and achieving consensus over nodes about a correct ledger state. Other approaches include Proof of Stake, Delegated Proof of Stake, Proof of Authority, Practical and Delegated Byzantine Fault Tolerance, Directed Acyclic Graphs, and Proof of Honesty, to name only a few. All have their strengths and weaknesses. Broadly speaking, they share the following elements:

- They keep copies of the ledger and transactions distributed in several locations (that is, they are forms of Distributed Ledger Technology (DLT).
- They use cryptography techniques (mostly PPK) to verify that a user who submits a transaction owns or is authorized to use an account.
- They use some sort of recursive hashing technique to link current transactions to historical transactions to keep them ordered and make rewriting history difficult.
- They use some sort of consensus or governance mechanism to come to agreement about which transactions, blocks, or chains are canonical (that is, they identify one ledger state that is definitive and authoritative).

Conclusion

Encryption and hashing are the fundamental building blocks of many of the essential enabling technologies that make the Internet, information storage and exchange, and electronic communication and commerce possible. Developing a basic understanding of how these two cryptographic ideas work makes it much easier to understand the key technologies that use them as a foundation.