Chapter 13

Bitcoin, Blockchains and Ethereum

13.1 Bitcoin overview 376
13.2 Transaction types and fields 379
13.3 ‡Bitcoin script execution (signature validation) 382
13.4 Block structure, Merkle trees and the blockchain 384
13.5 Mining of blocks, block preparation and hashing targets 386
13.6 Building the blockchain, validation, and full nodes 391
13.7 ‡Simple payment verification, user wallets, private keys 395
13.8 ‡Ethereum and smart contracts 399
13.9 ‡End notes and further reading 405
References 407

The official version of this book is available at
https://www.springer.com/gp/book/9783030834104

ISBN: 978-3-030-83410-4 (hardcopy), 978-3-030-83411-1 (eBook)

Copyright ©2020-2022 Paul C. van Oorschot. Under publishing license to Springer.

For personal use only.

This author-created, self-archived copy is from the author's web page.

Reposting, or any form of redistribution without permission, is strictly prohibited.

version: 15 Oct 2021

Chapter 13

Bitcoin, Blockchains and Ethereum

Bitcoin is a communication protocol and peer-based system supporting transfer of virtual currency units denominated in *bitcoin* (BTC). It uses hash functions and digital signatures to implement money—but distinct from a line of earlier proposals, it does not rely on central trusted authorities. Money is moved between parties by *transactions*, its ownership dictated by transaction records, public keys and control of matching private keys. System controls prevent fraudulent currency duplication. Rules (controlled by community consensus and subject to change) limit overall currency production; to experts viewing that as a requirement for legitimate currencies, this justifies calling Bitcoin a *cryptocurrency*.

This chapter explains how Bitcoin works, and its underlying *blockchain* technology, which delivers publically verifiable, immutable records whose integrity relies on neither trusted central parties nor secret keys. It also gives an overview of Ethereum, extending Bitcoin to a decentralized computing platform supporting what are called *smart contracts*. End notes provide references to explore underlying principles and further details.

13.1 Bitcoin overview

The basic idea of Bitcoin is as follows. Assume (without further explanation) the existence of a *transaction statement* (digital string), by which one party is recognized as the owner of some amount of virtual money. The owner can transfer this value to a new owner by digitally signing a second transaction statement that conveys this intention and includes the signature public key of the new owner (intended receiver), and a hash value that uniquely identifies the original statement (implying the amount to transfer). The second owner can then transfer the value to a third owner by digitally signing a third statement that includes the public key of the third owner (as intended receiver), and a hash value of the second statement (by which the second owner came to own the virtual *coin*).[1] Thus a coin is represented by a chain of signatures over data strings, and its owner is determined by signatures and public keys. We now consider how this idea is extended into a practical system that reduces the risks of cheating and theft.

[1] Here *coin* is a synonym for *money*, but may mislead readers—as virtual coins may be split or merged.

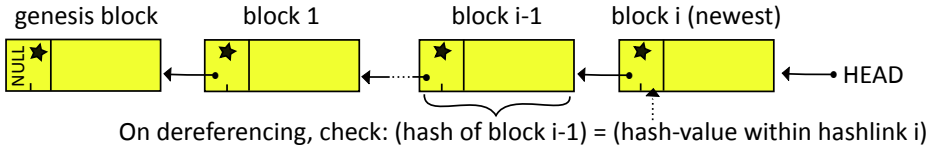

Figure 13.1: A blockchain of block headers. A *hashlink* (Fig. 13.2) includes a hash of the referenced item's content. The genesis block serves as a *trust sink*, an endpoint dual of *trust anchors* (Chapter 8). Later figures show that blocks also incorporate transactions.

PUBLICLY VERIFIABLE APPEND-ONLY LEDGER. Bitcoin uses a special type of linked list, called a *blockchain*. Each item or *block* in the list represents a set of transactions, and also points to its predecessor (Fig. 13.1). As we will see, the blockchain and supporting data structures provide a continually updated log of transactions that is:

- *publicly verifiable* (replicated and open for public view; no encryption is used; transactions can be verified to be unaltered and valid according to system-defined rules);
- *append-only* (no parts can be erased; all past transactions remain intact); and
- a *ledger* (with offsetting debits and credits, or inputs and outputs in Bitcoin terms).

The ledger dictates who owns the money, i.e., who has the authority to claim (spend, transfer) coins. The system records all valid transactions over time. It relies on peer *nodes* (page 394) that exchange information in a distributed network. Coded into peer node software is a small set of public trusted data, such as the first (*genesis*) block in Fig. 13.1; nodes can then independently confirm the validity of transactions, as will be explained.

LINKED LISTS. A *linked list* is a basic data structure used to traverse a sequence of items. Each item contains data plus a link to a next item. The link is implemented using either a *pointer* (memory address) or an *index* into a table. The end of the list is denoted by a reserved value (e.g., NULL); the start is accessed by a head link (HEAD).

HASHLINKS. While typical linked lists point forward to the newest item, a blockchain points backward in time, and uses a special type of link called a *hashlink*. Conceptually, each hashlink in a blockchain has two fields (Fig. 13.2):
1) a reference to a target item; and
2) the hash value of the target item itself (using a known hash function).
The hash input *includes the referenced item's own hashlink*. On dereferencing a hashlink, an integrity check is done: the retrieved item is hashed, and tested for equality with the hashlink's hash field (Fig. 13.1), as part of a built-in mechanism *to detect whether the referenced item has been altered*. To implement hashlinks efficiently, a supporting data structure is often used, e.g., list items are stored in a table indexed by their hash value.

ADDRESSES FROM PUBLIC KEYS. No owner name is attached to physical cash. This provides a degree of *untraceability* (albeit imperfect—serial numbers can be used to

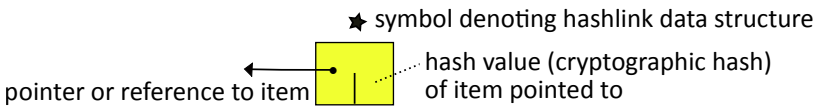

Figure 13.2: Hashlink data structure. In practice, the hash value itself serves as reference.

trace paper money). Many cryptocurrency designs strive for similar privacy-friendliness. Rather than associate coins with explicit user identities, Bitcoin uses *addresses* derived from public keys. The basic scheme for deriving an address from $K = (x, y)$, a representation of an ECDSA public key (page 397), is as follows; "::" denotes concatenation.

$$\begin{aligned} \mathit{mainPart} &= \text{RIPEMD160}(\ \text{SHA256}(\ K\ ))\qquad\qquad [\text{two hash functions are used}]\\ \mathit{checksum} &= \text{SHA256}(\ \text{SHA256}(\mathit{version} :: \mathit{mainPart}))\quad [\mathit{version}\text{: 1-byte format code}]\\ \mathit{address} &= \mathit{version} :: \mathit{mainPart} :: (\text{first 32 bits of } \mathit{checksum}) \end{aligned}$$

The 32-bit checksum can detect all but 1 in 2^{32} benign (e.g., typing) errors. Addresses are 26–35 bytes (depending on version and format) after base58 encoding (next), and are made available selectively to other parties, e.g., by email, text message, or 2D barcode.

‡**Encoding of binary strings.** The strings that comprise transactions include binary values (e.g., signatures, public keys, hashes). For display in human-readable form, these strings may be encoded as alphanumeric or hexadecimal characters. One encoding, called *base58*, uses 58 printable characters: 26 uppercase, 26 lowercase, and 10 digits, excluding 4 ambiguous characters: uppercase "I" and "O", lowercase "l", and the digit 0.

Single-use addresses. At a glance, Bitcoin addresses hide user identities, providing *anonymity*. However, as they are in essence *pseudonyms*, reusing one address across many transactions allows user actions to be linked; recall, all transactions are in a public ledger. Such *linkability* does not directly expose identities, but when combined with information beyond formal transaction details, anonymity might nonetheless be compromised. In practice, this motivates *per-transaction* addresses, e.g., users are encouraged to use software that creates a new Bitcoin address (as receiver) for each incoming transaction—implying a new key pair.

Coin owners represented by UTXOs. The simplest Bitcoin transaction transfers bitcoin ownership from one address (providing bitcoin as input), to another (receiving bitcoin as transaction output). The new output is *unspent* (available to spend); the input becomes *spent* as a result of the transaction. Transaction details allow *unspent transaction outputs* (UTXOs) to be associated with addresses. The blockchain itself neither explicitly tracks which user is associated with which addresses, nor keeps per-user UTXO balances; data structures external to the blockchain are used (by Bitcoin nodes) to keep track of UTXOs, i.e., which outputs have not yet been spent. Fig. 13.3 provides a model.

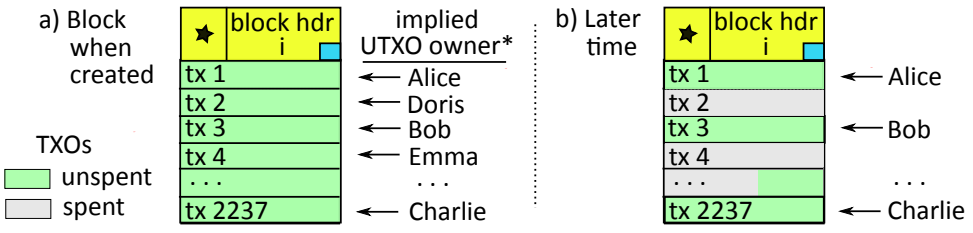

Figure 13.3: Unspent transaction outputs (UTXOs). Blocks added to the blockchain contain transaction outputs (TXOs), which are consumed as input by later transactions. *UTXOs are not directly associated with users, but owners are implied by details in each transaction (e.g., addresses derived from public keys). Section 13.2 explains transactions.

WALLETS. Use of per-transaction addresses and corresponding new key pairs (above) is supported by *wallet* software (Section 13.7), simplifying the perspective presented to users. Wallets keep track of all addresses associated with a user—and importantly, the corresponding private keys needed to claim access to coins sent to those addresses. Wallets use these to control and manage a user's coins, tracking all transaction outputs available to the user-owned addresses. This is essential, because no central database run by any official organization explicitly records Bitcoin accounts (owners) or their balances.

DOUBLE SPENDING AND TRANSACTION VALIDITY. What stops an electronic coin owner from *double spending*, that is, transferring the same coin to two different parties? Early *electronic cash* systems relied on a central authority to track who owns each coin at any instant. In contrast, Bitcoin avoids a central authority for this, instead relying on a novel consensus-based design (Section 13.6) that incentivizes and rewards cooperation.

VALIDATING TRANSACTIONS. What determines the validity of a Bitcoin transaction? We will find (Section 13.6) a checklist of items including: verifying cryptographic signatures, checking that claimed transaction outputs have not yet been spent (i.e., tracking UTXOs, which includes taking into account the time order of transactions), and confirming that transactions have been accepted by Bitcoin's consensus system.

SIMPLIFIED OVERVIEW. Transactions originate from user wallets, and are injected into and shared by a peer-to-peer network of Bitcoin nodes. These provide information to *miners*, which package sets of transactions into blocks (for efficiency), and compete to have their blocks added into the blockchain. The remainder of the chapter pursues details.

13.2 Transaction types and fields

Transactions are grouped into blocks (as detailed in Section 13.4), which are then linked to form the blockchain of Fig. 13.1. Here we consider the structure of individual Bitcoin transactions. For concreteness, we show representative data structures, reflecting relatively early implementations. While details will change over time (and already have),[2] this does not impact our main goal: introducing principal ideas and underlying concepts.

RELATING INPUTS TO OUTPUTS. A transaction transfers bitcoin from a sender (who owns the input, prior to the transaction) to a receiver (who will receive the input, as it turns into output). An input is itself the output of a prior transaction (Fig.13.4).

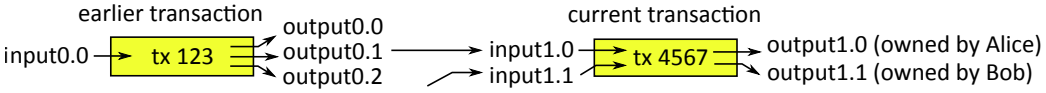

Figure 13.4: Model of how transaction inputs transition to outputs (becoming new inputs).

REGULAR TRANSACTION (DATA STRUCTURE). Table 13.1 shows the fields in a transaction with one input and one output. In general, there may be multiple inputs and outputs. A transaction's intended output value is explicitly stated (`btc-value`). The input value is implied, by reference to an earlier transaction (`prev-out-txID`). The total output

[2]For example, *SegWit* (BIP141, Section 13.9) moves unlocking scripts to the end of a transaction.

Section	Field Name	Description
top	`txID`†	(implicit) hash of rest of transaction; effectively, a serial no.
	`version`	protocol version (dictates block format and fields)
	`n-in`	number of inputs in this transaction
	`n-out`	number of outputs (indexed as: 0, 1, ...)
	`lock-time`	if 0, allows transaction to be included without delay
input		for each input, include:
	`prev-out-txID`	txID of earlier transaction whose output is to be used
	`prev-out-index`	which output of that earlier transaction (0 means first)
	`scriptSig`	an *unlocking script* (e.g., with signature to claim ownership)
output		for each output, include:
	`out-index`	(implicit) used as reference when this output is later spent
	`btc-value`	bitcoin value intended for receiver
	`scriptPubKey`	a *locking script* (e.g., with receiver's hashed public key)

Table 13.1: Regular transaction (representative Bitcoin data structure). †`txID` is not an explicit field, but is computed by software on retrieving a transaction (and also used as an index to retrieve it). The other fields in the top section can be skipped over for now.

value must not exceed the total input (lest money be arbitrarily created). The hash of a transaction serves as its *transaction ID* (`txID`), and is how transactions are referenced. The input section's field `prev-out-txID` is the hash of a previous transaction.

Example *(Scripts and their semantics).* The following example transaction presents the most common script type.[3] It has an input section *unlocking script* (field `scriptSig`), and an output section *locking script* (`scriptPubKey`):

$$\texttt{scriptSig}: \texttt{<sig1><pubkey1>} \tag{13.1}$$

$$\texttt{scriptPubKey}: \text{OPDUP OPHASH160 } \texttt{<addr1>} \text{ OPEQUALVERIFY OPCHECKSIG} \tag{13.2}$$

The `scriptSig` field contains the sender's signature over selected transaction data, and a corresponding public key (it will be compared to a key from an earlier transaction); as Fig. 13.4 shows, a transaction's input is drawn from the output section of an earlier transaction (we'll give more details soon). The `scriptPubKey` field includes the receiver address (plus items noted shortly). The semantic meaning of this pair of scripts is:

> By signature `<sig1>`, related to `<pubkey1>`, I claim (unlock) an output value of an earlier transaction, and transfer the stated amount to a receiver whose public-key hash is `<addr1>`. The receiver, to later claim this new output, must produce a signature that is valid relative to this same public-key hash.

The strange-looking *opcodes* (operation codes) in (13.2) are instructions that will be executed within the script processing part of transaction validation (Section 13.3).

[3] ‡This script has pattern type *Pay-to-Public-Key-Hash* (P2PKH). It is more compact than and replaced older *Pay-to-Public-Key* (P2PK) scripts, as hashes are shorter than public keys themselves. P2PKH also conceals the payee public key until used to claim an output. See also *Pay-to-Script-Hash* (P2SH), page 383.

Section	Field Name	Description
top	→	same fields as regular transaction (Table 13.1)
input	`prev-out-txID`	`0x0000...` (32 zero bytes; no prior output claimed)
	`prev-out-index`	`0xFFFFFFFF`
	`coinbase`	unlocking script, also contains field for `blockheight`
output	→	same fields as regular tx; miner itself is the receiver; `btc-value` = block reward per Equation (13.3)

Table 13.2: Coinbase transaction (data structure) contrasted with regular transaction.

‡**LOCK TIME.** A `lock-time` field value d implies a block number (height) if $d < 5$ million, otherwise a Unix time in seconds. This delays a transaction if needed to ensure it appears only in a block of height $> d$ or with timestamp $> d$. If $d = 0$, there is no effect.

‡**Exercise** (Lock time). Look up possible reasons for using a non-zero lock time. (Hint: this can allow a payer to change their mind, or make conditional payments.)

COINBASE TRANSACTION. Table 13.2 outlines the data structure for a special type of transaction, the *coinbase transaction*. Rather than consume output from an earlier transaction, it creates (*mints*) units of currency as part of a *block reward*, specified in equation (13.3). The reward value is conveyed using a `btc-value` field, as found also in Table 13.1. The reward is for solving a puzzle task that requires a huge number of computational steps, called a *proof of work* within *block mining* (Section 13.5). The entity (*miner*) completing this task specifies who the reward goes to (e.g., its own address). As a coinbase transaction claims no previous outputs, its input fields `prev-out-txID` and `prev-out-index` are given special values (Table 13.2). A `coinbase` field plays the role of the usual `scriptSig` field (unlocking script) of regular transactions, and has a `blockheight` field set during mining (page 387). The coinbase transaction is listed as a block's first transaction. The process for recognizing and rewarding entities for generating blocks is part of Bitcoin's *consensus mechanism* (Section 13.6), which over the long term rewards miners based on their computational power.

BLOCK REWARD AND CHANGE. A transaction consumes the sum of its inputs. An entity authorizing a transaction that pays out to other receivers less than this sum, may give itself *change* by specifying an extra output to an address of its own. Any excess of inputs over specified outputs serves as a *transaction fee* (page 388). The sum of all fees is claimable by the block miner as part of the *block reward* in the coinbase transaction:

$$\text{block reward} \;=\; \text{block subsidy} + \text{transaction fees} \tag{13.3}$$

The *block subsidy* started at 50 bitcoin in 2009. It is halved every 210,000 blocks (this works out to be about every four years); it became 6.25 bitcoin in May 2020. This is part of a ruleset limiting total currency units to 21 million bitcoin; at the current target average of one block every 10 minutes, the last bitcoin units will be minted in the year 2140, having value 1 *satoshi* = 10^{-8} BTC = 0.00000001 BTC (the smallest bitcoin unit). This is 1 one-hundred-millionth of a bitcoin, i.e., 1 BTC = 100 million satoshis.

13.3 ‡Bitcoin script execution (signature validation)

We have seen that Bitcoin transactions contain scripts. Here we work through an example to explain how these are one component used to check the validity of transactions.

STACK-BASED MACHINE MODEL. Bitcoin scripts process strings of tokens using an efficient, well-known *stack-based* computational model. (Loops are not supported; running time is bounded by the script size.) The standard approach is as follows. If the next token is a *binary* operator: perform the operation, consuming as operands the top two items on the stack (then place the result back on the stack, i.e., PUSH it). If the next token is a *unary* operator: perform the operation, consuming the top stack item (PUSH the result). If the next token is an *operand* (not an operator): PUSH it onto the top of the stack.

EVALUATING INPUT-OUTPUT SCRIPTS. For a transaction to be valid, all of its execution scripts must evaluate to TRUE (i.e., with final value TRUE atop the stack, and no error exceptions).[4] More specifically: for each *output* script value claimed from a previous unspent transaction, an *input* script must be provided that makes the joined script TRUE, as now explained. An *execution script* is the concatenation of two pieces identified in the input section of the current transaction. Consider the pieces:

script1 = <sig1> <pubkey1>

script0 = OPDUP OPHASH160 <addr0> OPEQUALVERIFY OPCHECKSIG

script1 is from our earlier `scriptSig` field (page 380); script0 is from the `scriptPubKey` of a *previous* transaction output identified by the `prev-out-txID`, `prev-out-index` fields in the current transaction (thus denoted <addr0>, distinct from <addr1> on page 380). Semantically, signature <sig1> claims the right to redeem a coin; <pubkey1> is provided to validate this claim; and <addr0> is an address (coin owner) from the earlier transaction. The Bitcoin *opcodes* (operation codes) used below have these meanings: OPDUP (duplicate the top item on the stack), OPHASH160 (derive an address from a public key), OPEQUALVERIFY (test whether two values are equal), OPCHECKSIG (verify a digital signature). The example script execution (below) will test two main things:

a) that the signature verifies mathematically, using <pubkey1>; and

b) the hash of <pubkey1> matches that of the key for the coin in the old transaction.

This ensures that the sender in the new transaction owns the coin being spent (has access to the private key). The comparison of public keys involves checking that the (old) receiver address from script0 agrees with the address newly derived from the public key in script1—in which case the latter is safe to use for signature verification. We next step through how Bitcoin script engines execute these checks.

Example *(Script execution).* Scripts are evaluated linearly in one pass as *postfix expressions* on a stack-based machine (Table 13.3). It processes the 7-token sequence

<sig1> <pubkey1> OPDUP OPHASH160 <addr0> OPEQUALVERIFY OPCHECKSIG

from the concatenation script1::script0 of our scripts above by the following steps (K2A denotes hashing, to convey the semantics of deriving Bitcoin addresses from public keys):

[4] Script engine rules also treat early exit as failure (ruling out tactics such as: push TRUE then exit).

1. PUSH <sig1>
2. PUSH <pubkey1>
3. duplicate the operand currently on top of the stack
4. POP the stack's top item into *opnd*; compute K2A(*opnd*); PUSH result onto stack
5. PUSH <addr0>
6. POP top two operands, compare them, if unequal then (PUSH FALSE, then EXIT)
7. POP into *opnd1*, POP into *opnd2*, then test whether *opnd1* (public key) confirms *opnd2* (signature) to be a valid signature. The signature usually covers[5] fields (Table 13.1) from the new transaction (`prev-out-txID`, `prev-out-index`, `btc-value`, `scriptPubKey`, `lock-time`) and the previous transaction (`scriptPubKey` corresponding to script0, identified by these `prev-out` fields). For a valid signature PUSH TRUE, else PUSH FALSE.

				addr0		
		pubkey1	K2A(pubkey1)	K2A(pubkey1)		
	pubkey1	pubkey1	pubkey1	pubkey1	pubkey1	
sig1	sig1	sig1	sig1	sig1	sig1	TRUE
1	2	3	4	5	6	7

Table 13.3: Stack frames after each of the above seven script execution steps.

Script validation is carried out by various entities, including miners before selecting any transaction for a new block (Section 13.5), and payee software before acknowledging receipt. The above opcodes are a small subset of those possible in Bitcoin scripts, and some—such as OPCHECKSIG in step 7—are elaborately tailored to specific data structures.

WRITING ARBITRARY DATA TO THE BLOCKCHAIN. Bitcoin's success has led to unrelated applications using its blockchain to store and publicly timestamp data, e.g., by creating transactions that write non-Bitcoin data into address fields with no intention of coin transfer.[6] This results in unspent transactions being stored forever (in RAM or disk memory) by Bitcoin nodes that track UTXOs, wasting resources (transmission, storage, CPU cycles) while advancing no Bitcoin goals. In response, Bitcoin officially now allows (although some used to discourage it) the insertion of arbitrary data (at the time of writing, it is often 40 bytes, but up to 83) after the *data-carrying* script opcode OPRETURN. Scripts including OPRETURN by definition evaluate to FALSE, and their transactions are called *provably unspendable*. As a key point: Bitcoin nodes internally maintain UTXO sets (indexed by `txID` for efficiency), and unspendable transactions are removed from these.

Exercise (Zero-value outputs). Given that OPRETURN is supported, a possible mining policy is to include (in new blocks) transactions with zero-value outputs only if they have OPRETURN scripts. Explain why this might be considered a reasonable policy.

P2SH. *Pay-to-Script-Hash* (P2SH) is a powerful alternative to *P2PKH* scripts (page 380). When a receiver provides a *P2SH* address to a payer, the payer uses it as a regular

[5] Appended to each signature is a SIGHASH byte code specifying which fields the signature covers.

[6] As such *data-encoding addresses* are not the hash of a public key or script, the TXOs are not spendable.

address. To redeem the payment, the receiver must produce an *unlocking script* that:
1) hashes to the specified P2SH address (possibly not even involving signatures); and
2) evaluates to TRUE when combined with the *locking script*.
This allows complex retrieval conditions, e.g., requiring multiple signatures, while isolating the script complexity from both the payer and the blockchain until payments are claimed (with scripts represented on the blockchain by their hash). Had P2SH scripts been in Bitcoin's original design, P2PKH (and P2PK, which it replaced) would not have been needed. (As a side point, the design of P2SH itself would also have been simpler.)

Exercise (P2SH). Transactions are redeemed by combining unlocking and locking scripts. With a P2SH address, the locking script becomes the *redeem script* (whose hash is the P2SH address), which becomes part of the unlocking script, and the overall transaction locking script becomes simpler (shorter). (a) Give an example of a Bitcoin locking script using OPCHECKMULTISIG, that requires at least two signatures, and a corresponding unlocking script. (b) Using a P2SH address, give a corresponding set of: redeem script, locking script, and unlocking script. (c) List and explain the advantages of using the P2SH version. (Hint: [1, pp.132–135].)

13.4 Block structure, Merkle trees and the blockchain

Here we describe the fields within blocks, including lists of transactions bundled into them. Blocks are back-linked to form the *blockchain*. First, it helps to understand Merkle hash trees, a data structure used here to efficiently, securely bind transactions into blocks.

MERKLE TREES. To construct a *Merkle hash tree* of transactions for a specific block, the transactions are first given a fixed order by a *block miner* (Section 13.5). The `txID`s (Table 13.1) of these ordered transactions serve as a tree's leaf nodes (Fig. 13.5, level 3). The leaf nodes are paired, and the concatenation of each pair's values is hashed. (For missing items on the right side of an incomplete tree, any singleton item is concatenated to itself.) The resulting hashes produce level 2 entries, which are likewise paired and concatenated, and so on. The resulting final hash value is called the *Merkle root*.

BLOCK AS DATA STRUCTURE WITH TRANSACTIONS. A Bitcoin *block* has two main parts (Table 13.4): a *block header*, and an ordered set of detailed *transactions* each of length typically 250 to 1000 bytes. The `merkle-root` field in the header serves to

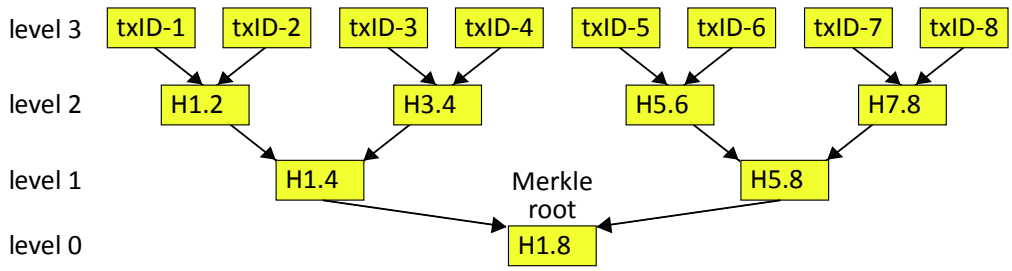

Figure 13.5: *Merkle tree* of transaction hashes (hash tree, *authentication tree*). Each node holds a 32-byte hash value. The leaf nodes are the hashes of respective transactions.

Block Part (bytes)		Field Name	Description
top	(4)	block-size	number of bytes, excluding this field
block header	(80)		
	(4)	`version`	protocol version (dictates block format)
	(32)	`prev-block-hash`	hash of previous block's block header
	(32)	`merkle-root`	hash from Merkle tree of this block's txs
	(4)	`timestamp`	approx. Unix time block hashing is done
	(4)	`mining-bound`	compact format of *hashing target* (page 389)
	(4)	`mining-nonce`	modified in trials to satisfy hashing target
transaction list			
	(varies)	`tx-count`	number of transactions in this block
	(varies)	`tx[0]`	*coinbase transaction* (bytestring)
	(varies)	`tx[1]`	regular transaction number 1 (bytestring)
	(varies)	`tx[2]`	regular transaction number 2 (bytestring)
	...	...	... note: tx order affects `merkle-root` value
	(varies)	`tx[n-1]`	last regular transaction (bytestring)

Table 13.4: Block (data structure) with transactions. Block-related data commonly stored externally to a block itself includes the hash of the block's header, and the block number (called *height*). *Unix time* corresponds to the number of seconds since January 1, 1970.

incorporate the entire ordered list of transactions into the header; it *commits*[7] the header to precisely these (ordered) transactions, down to their bit-level representation. The header field `prev-block-hash` identifies the previous block in the chain, and is a *hashlink* (page 377) specifying the hash of that previous block's header. The hash of its block header is used to uniquely identify each block; this is based on the negligible probability of two randomly selected blocks having the same 256-bit header hash (i.e., $1/2^{256} \approx 0$ in practice). Because `merkle-root` is included, the hash of a block header provides a unique *fingerprint* for not only the block header, but for the full set of transactions in the block.

BLOCKCHAIN OF BLOCK HEADERS. The Bitcoin system is engineered such that a new block is produced roughly every ten minutes. In practice, each block typically holds 1000 to 2500 transactions. As discussed, a block header's Merkle root incorporates that block's transactions, while each new block's hashlink points back to the most recent block before it, all the way back to the original *genesis block*. This orders transactions both within and across blocks. The result is the *blockchain*: a back-linked list of blocks ordering and integrating all transactions in time, made available as a public ledger (Fig. 13.6). While the blockchain is a linked list of block headers, it effectively also links all transactions by virtue of the Merkle root fields. The combined structure of hashlinks and Merkle tree roots allows integrity verification of all transactions in the ledger. Again, this relies on the *collision resistance* property of cryptographic hash functions, not secret keys.

[7] Assuming the hash function used is *collision resistant* (Chapter 2), it can be proven that changing even a single bit in any transaction would, with overwhelming probability, change that transaction's `txID`, which would change the `merkle-root` value, which would change the hash of the block header (which a hashlink verifies). In this way, the `merkle-root` value binds (*commits*) the block header to a precise set of transactions.

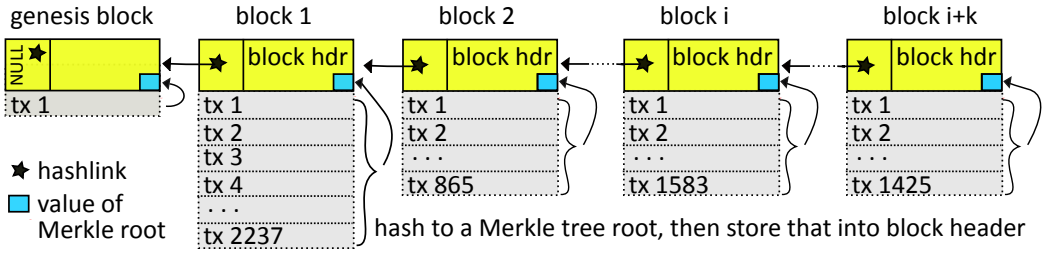

Figure 13.6: The full blockchain. A block header's hashlink (Fig. 13.1) holds the hash of the block header pointed to, allowing verification of the integrity of the pointed-to block. The genesis block (hard-coded into nodes) serves to validate the final integrity check.

How data integrity is checked. Verifying data integrity of a given blockchain requires holding a reference to the current *head block*, and a trusted genesis block. Starting from the head, as each block's `prev-block-hash` field is dereferenced to get the preceding block, this hashlink's hash field is compared to the hash of that previous block's header (hashed after retrieval). In a full blockchain check, this process ends with a match against the hard-coded genesis block, validation failing if any integrity check fails along the way. As the blockchain is a chain of block headers, this process detects modification of any block header fields, including `merkle-root` fields. Note, however, that this does not itself directly verify integrity of individual transactions, or that a given transaction is part of a given block—that requires recomputing a Merkle root value, or a separate process called *Merkle path authentication* as explained later in Section 13.7.

Additional issues. We note here two related issues. (1) Independent of integrity checks, an attacker aiming to inject false transactions (bitcoin transfers) must also arrange valid corresponding transaction scripts, with valid digital signatures. (2) Can attackers cause Bitcoin nodes to accept a false *head block*? This threat relates to how Bitcoin consensus emerges in accepting new blocks, and what it means for a block to be "on the blockchain". Section 13.6 pursues these important questions.

13.5 Mining of blocks, block preparation and hashing targets

The next two sections consider the process of preparing blocks, and how they are selected for inclusion on the blockchain. A simplified overview sets the context (cf. page 379).

Mining and block preparation (overview). Bitcoin user transactions are submitted to a peer-to-peer network for packaging into blocks by *miners*, who compete in a race to construct *candidate blocks* that meet eligibility rules for inclusion in the blockchain. Block construction requires preparing block headers, after first selecting a set of transactions (needed to compute a Merkle root) from a larger available set that miners have collected in *memory pools*. One rule requires completing a computational task (*puzzle*), which on average rewards miners in proportion to their effort invested. The task is executed by a search loop (which updates a counter input or *nonce*), seeking a hash output with numerical value below a *mining bound*. This bound is periodically ad-

justed to control the block production rate. Overall, this bizarre-sounding process results in consensus-based decisions on which next block gets into the blockchain, with a novel incentive mechanism driving behavior aligned with system goals. Details are next.

BLOCK MINING PSEUDOCODE. A simplified mining process overview is as follows (review Table 13.2 for coinbase transaction fields).

```
1  /* Illustrative block mining pseudocode. Here r denotes the block header */
2  k := 1 + (blockheight of current top block)   /* k is target block height
3  boundB := hashing target from mining bound    /* recalibrated every 2 weeks
4  prepareRegularTxs()        /* select + finalize regular txs for this block
5  prepareCoinbaseTx(k)       /* set locking script; coinbase.blockheight := k
6  r := prepareBlockHdr()     /* compute merkle-root then set up header fields
7  LOOP                       /* timestamp doubles as second nonce, outer loop
8  { r.timestamp := getTime() /* update (approx. Unix time of hashing)
9    r.nonce := 0             /* primary mining nonce, indexes inner loop
10   LOOP{                    /* outer loop expands search space of inner
11     trialHash := h(h(r))   /* h is SHA256; hash the entire block header r
12     IF( trialHash <= boundB )THEN (quit both loops)  /* joy, block is valid
13     r.nonce := r.nonce + 1 /* updating nonce changes header being hashed
14   }UNTIL(r.nonce = 2^32)   /* inner loop nonce is 32 bits
15 }
```

Both loops are exited on finding a satisfying nonce (success, line 12), or on an interrupting announcement that a competing miner has found a valid block at the same block *height*.[8] The inner loop increments `mining-nonce` in the block header (line 13). The outer loop updates the timestamp (line 8), which also serves as a second-level nonce—note that when the inner loop repeats starting from 0, the timestamp field ensures an altered header. Mining and validation of a block header r use the SHA256 hash function (SHA-2 family, Chapter 2), with 256-bit hash values, and nested use as follows: SHA256(SHA256(r)).

PREPARING A BLOCK HEADER. To prepare a block header for (and alter during) the hashing loops, six fields are involved (see pseudocode line 6; review Table 13.4).

1. Version. This is 1 or 2 (at the present time of writing).
2. Prev-block-hash. This is the hash of the blockchain's current top block (main branch), which will be called the *parent* of the *candidate block*[9] of the successful miner.
3. Merkle root. This is computed per Fig. 13.5, after first selecting (pseudocode line 4) all the regular transactions for the block from the miner's *memory pool* (below), and then preparing the coinbase transaction (which yields `txID-1` in Fig. 13.5).
4. Timestamp. This estimates the time a block emerges (Unix format, seconds). The pseudocode assumes the inner loop takes at least 1 s, changing `timestamp` at line 8.
5. Mining bound. The hash target's 4-byte form (page 389) is written into this field.
6. Mining nonce. This is used mainly as a counter (pseudocode lines 9, 13). The final nonce value for a candidate block allows the hash of the header to satisfy the bound.

[8] The block *height* (pseudocode line 2) is as shown in Fig. 13.7, page 389.

[9] We call an in-progress block a *candidate block* once it has a valid *proof of work* (i.e., solved puzzle).

‡**NONCE SOLUTIONS DIFFER ACROSS MINERS.** Are miners all searching for the same nonce? No. A nonce solution for one in-progress block does not work for others, as blocks being hashed have different header `merkle-root` values—due to differing coinbase transactions (e.g., miners use different addresses). Miners also tend to select different candidate transactions, in different orders. Thus, mining is not a computing power footrace to find the same nonce—but over time, success correlates with computing power.

MEMORY POOL. As each new individual transaction is submitted by a user, it is immediately propagated by Bitcoin peer nodes. For official recognition, a transaction:
1) must be embedded into a valid block, which includes a proof of work; and
2) this block must become part of the settled blockchain, i.e., a *confirmed* block (p.392).
Miners store received transactions in a *memory pool* or *transaction pool*, for selection into their own in-progress blocks (different miners tending to have slightly different pools). When a miner receives a new valid block from peers, it removes from its local memory pool all the transactions in that block. Transactions in the memory pool are also filtered to remove any that have become ineligible due to the new block, e.g., whose claimed outputs (previously UTXOs) were consumed by a transaction in the block. If the new block results in a miner abandoning its own in-progress block, then any transactions in the abandoned block that are not in the new block, remain eligible for the next block to be built.

PRIORITIZING MEMORY POOL TRANSACTIONS. Miners select transactions for in-progress blocks from their memory pool. Only a subset are typically selected, for reasons including a maximum block size. Recalling equation (13.3), the reward for mining a block includes not only a block subsidy, but all fees offered by transactions in the block. Thus an obvious approach is to select transactions with the highest *transaction fee* per byte of transaction size. Beyond fees, miners may rank each transaction in their memory pool using a priority formula. One such formula might be (square brackets indicate units):

$$priority = \sum_{all\ inputs} \frac{\text{input_value [satoshis]} \times \text{input_age [depth]}}{\text{transaction_size [bytes]}} \tag{13.4}$$

This sum weights each of a transaction's inputs both by value, and by age measured as a *depth* below the blockchain's current top block (Fig. 13.7). An input of depth k has a block number k less than the top block. The Bitcoin reference software at one time selected a transaction even if it offered no fee, provided that its priority was high enough.

Note that equation (13.4) gives higher priority to transactions that are higher value, older, and shorter—by this, unselected transactions (even zero-fee) increase in priority over time, as their depth increases. Individual miners using their own prioritization rules may exclude zero-fee transactions.[10] It is naturally expected that transactions offering higher fees will appear on the blockchain sooner. *Wallet* software is configured to offer fees that are appropriate for its users (or to prompt a user to offer suitable fees).

‡**Exercise** (Transaction fees). a) What is the current value of a bitcoin, in US dollars? b) What is currently considered to be a standard transaction fee for Bitcoin transactions (in bitcoin)? c) Why might one expect transaction fees to increase over time?

[10] Zero-fee transactions may now be a thing of the past, with the role of age also at the discretion of miners.

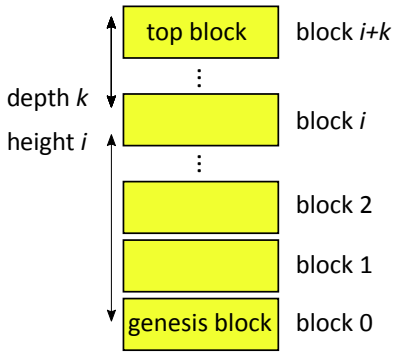

Figure 13.7: Bitcoin blocks (conceptual model, latest block stacked on top). A block i that has k blocks on top of it is said to be at *depth* k, meaning it also has k *confirmations*.

‡**ORPHAN TRANSACTIONS.** A transaction may be otherwise suitable for the memory pool, except that it claims one or more outputs that do not (yet) exist, neither on the blockchain nor in a memory pool. This may occur legitimately, e.g., if two transactions occur within a short time, the second claims an output from the first, and the second reaches some nodes before the first. Such *orphan transactions* are held in an *orphan pool* distinct from the main memory pool. Upon receiving a new block, nodes check whether any transactions therein allow any transactions to be rescued from the orphan pool to the memory pool.

MINING NONCE AND LOOP (DETAILS). We now look at `mining-nonce` details. Once an in-progress block is finalized, including all header fields except the mining nonce field, this nonce is set to zero. It is then used as a counter in the mining task (inner loop, page 387): compute the hash H of the block header, quit if $H \leq b$ (this implies success), otherwise increment the nonce and try again. As now explained, b is a hashing target.

MINING BOUND. Within block headers (Table 13.4), the 4-byte *mining bound* field is encoded as an 8-bit exponent t and 24-bit coefficient c, defining a 32-byte *hashing target*

$$b = c * 2^{8(t-3)} \tag{13.5}$$

as a 256-bit unsigned integer. The 32-byte hash H of a valid block header must be $\leq b$, as the *proof of work*; in practice, H then has many leading zero bits. To illustrate (Fig. 13.8):

$$\texttt{mining-bound} = \texttt{0x1711A333} \quad \text{yields target} \quad b = \texttt{0x11A333} * 2^{8(\texttt{0x17}-3)} \tag{13.6}$$

Note: `0x17` is decimal 23. Here the coefficient is multiplied by 2^{8*20} or logically shifted 20 bytes left. The formula slides a coefficient with 24 bits of precision between leading and trailing zeros (see the figure to understand this). Decreasing t yields a smaller bound, requiring more leading zeros. (Aside: the -3 in b's formula is a normalization relative to the 3-byte coefficient: an exponent of $t = 3$ leaves c unshifted as b's lowest 3 bytes.)

RECALIBRATING MINING BOUND. Bitcoin aims to generate one new block every 10 minutes on average. To this end, the *mining bound* (which determines the mining difficulty) is updated every 2016 blocks; the difficulty is increased if miners invest more resources (miner investment cannot itself be directly controlled). Note that $2016 = 10 \cdot 6 \cdot 24 \cdot 14$ is the number of blocks expected in 14 days at exactly 10 minutes per block.

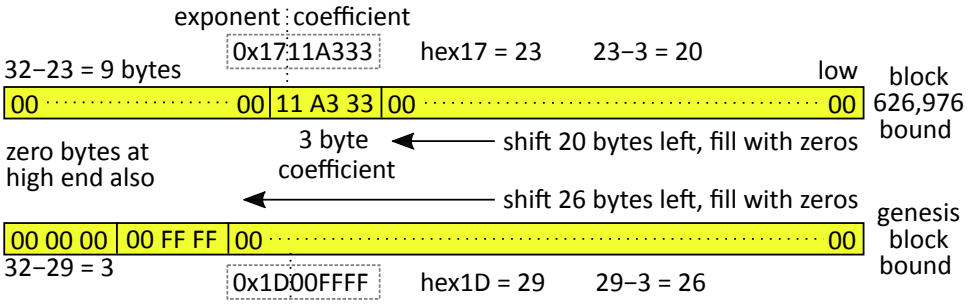

Figure 13.8: Converting 4-byte mining-bound fields to 32-byte *hashing targets*. The genesis block target bound, with 3 high-end leading zero bytes plus a fourth from the coefficient `00FFFF` itself, implies that a valid hash must have at least 32 leading zero bits. The mining bound from block 626,976 (April 2020) has a smaller exponent `0x17` (versus `0x1D`). This results in a smaller left-shift of the coefficient and thus more leading zeros, here requiring at least $9 \cdot 8 + 3 = 75$. (Why $+3$? Byte `0x11` itself has 3 leading zero bits.)

At recalibration, let (t_2, t_1) be the timestamps from the blocks (ending, starting) the two week period (i.e., the top block and the one 2015 blocks earlier). Let $b_{current}$ be the current hash target for blocks in this period, from their identical `mining-bound` fields. The desired time for 2016 blocks is $t_p = 2016 \cdot 10 \cdot 60$ seconds (block timestamps are in seconds). The actual measured time is $t_m = t_2 - t_1$. A scaling factor s is then computed as

$$s = \frac{t_{measured}}{t_{planned}} = \frac{t_m}{t_p} = \frac{t_2 - t_1}{t_p} \qquad \text{and} \qquad b_{new} = s \cdot b_{current} \tag{13.7}$$

The 4-byte `mining-bound` field value for blocks in the new period is formatted from b_{new}. If s is not in the range $1/4 \le s \le 4$, then s is reset to the boundary exceeded. (This same target b_{new} is independently computed by all Bitcoin nodes, using public blockchain data.)

Relative mining difficulty. To express the difficulty of finding a nonce satisfying the hashing target b_L for any block L, compared to that for satisfying the target b_G for the genesis block G, the *relative mining difficulty* D_L of block L is defined to be:

$$D_L = \frac{b_G}{b_L}, \qquad \text{or equivalently} \qquad D_L \cdot b_L = b_G \text{ (a constant)} \tag{13.8}$$

Note: b_G is in the numerator (left equation). From this definition, the genesis block itself has relative mining difficulty 1 (case $L = G$). $D_L > 1$ for all other blocks, as the target b_L is generally decreased over time, to make the hashing constraint harder (more leading zeros). From the right of equation (13.8), note: as block L's relative mining difficulty D_L goes up, its hash target b_L goes down (and vice versa). Note also that while mining involves enormous computational effort (on average), verifying that a block header satisfies the hash bound requires little work (hash the header once, then compare to the target).

Example *(Relative mining difficulty).* For block $L = 626,976$ (Fig. 13.8), and using equation (13.5) for b, we compute: $D_L = b_G/b_L = (c_G \cdot 2^{8 \cdot 26})/(c_L \cdot 2^{8 \cdot 20}) = (c_G/c_L)2^{48}$ for $c_G =$ `0x00FFFF`, $c_L =$ `0x11A333`. Using a hex calculator (or site), $D_L \approx 2^{43.8594}$.

Example *(Mining difficulty, bits and leading zeros).* The difficulty of mining can be expressed—albeit less precisely—solely by a lower bound on the number of leading zero

bits in the relevant hash. From Fig. 13.8, the genesis block requires 32 zero bits, and 75 are required by block $L = 626,976$. From our example, this block L has mining difficulty $D_L \approx 2^{43.8594}$. Note that $\log_2(D_L) = 43.8594$, while $75 - 32 = 43$. This illustrates the general relationship: a block with difficulty D_L corresponds to a target bound b with z leading zero bits, for $z = 32 + \lfloor \log_2(D_L) \rfloor$, i.e., dropping the fractional part. The number of hashing trials (nonce trials) expected *on average* is then approximated by $T = 2^z$.

‡**Exercise** (Mining is not for the meek). If you try to mine a block, might you get lucky and make some easy money? Not likely. Using the mining bound to find the expected number of SHA256 hashes needed, estimate your expected time if using one modern desktop CPU. (Hint: [53]; look up how many SHA256 hashes/s can be done. With money up for grabs, you face many competitors with heavy investments in specialized hardware.)

13.6 Building the blockchain, validation, and full nodes

Here we discuss how it is decided which blocks actually end up on the blockchain, through Bitcoin's decentralized consensus process. Once a miner has *mined* a block (i.e., has a proof of work—a block header satisfying the hash target), it sends the block to its peers, hoping that they start building on it rather than a competing miner's block. Peers independently validate this block and, if it is valid, add it to their local view of the blockchain and share it with further peers. On receiving a new valid block, it is expected that most miners abandon their own effort to find a block at that *height*, clean up their memory pool, and start mining a fresh block. We now look at how transactions and blocks are validated.

TRANSACTION VALIDATION. A peer network propagates new transactions as candidates for embedding into a next block. *Full nodes* (page 394) receive and, by design, share them further if they pass validity tests. Transaction validation checks include:

T1. size checks (e.g., transaction bytesize is in current min-max range; min ≈ 100);
T2. sanity checks on transaction format, field syntax, script structure;
T3. value checks (non-negative, < 21 million, sum of outputs $\leq$ sum of inputs);
T4. check whether the `lock-time` field implies that the transaction is not yet valid; and
T5. financial viability—each claimed output must: (i) exist, (ii) be unspent, i.e., in the UTXO pool, and (iii) have its locking script satisfied by an unlocking script.

Transactions previously seen are not re-forwarded. These checks are also used by miners before admitting transactions into their memory pool.

BLOCK VALIDATION CHECKS. Testing whether a new block is valid requires checks on its transactions and header. The main items to fully verify that a block is valid are:

B1. total block bytesize (originally max 1 Mb, now at most 2–4 Mb; see Section 13.9);
B2. sanity checks on the block header (format, syntax, timestamp reasonable);
B3. proof of work in the block header (the authentic hash bound is satisfied);
B4. Merkle root in the header matches hash of all transactions;
B5. each individual transaction validates (see transaction checks above); and
B6. a single coinbase transaction is included, listed first, and claims the correct value.

Once a block passes all these checks, a node is expected to propagate it—but only if, in the node's local view, the block is on the *main branch* (below). This requires a further check: the block header must have a valid hashlink to a parent, at the head of the main branch. (This parent should, based on earlier checks, also chain back to a trusted checkpoint block, if not the genesis block itself.) By propagating the block to peers, a node enables others to build on the same main branch. Since no individual node can trust that others follow these rules, each carries out independent checks—and a consensus emerges (below).

MAIN AND SIDE BRANCHES. Despite the simplified model (Fig. 13.7, page 389), the tip of the blockchain has a *main branch* (main chain) and side branches (Fig. 13.9). The *main chain* tends to be the longest chain (tallest stack), but by definition is the branch with greatest *aggregate proof of work* (smaller hash targets carry more weight). Note however that nodes in the distributed peer network may have different local views, based on incomplete information (e.g., due to temporary link failures, or propagation delays related to network topology); for competing new valid blocks at the same height and `mining-bound`, a miner tends to build on the one first received. Thus a single node's view is not definitive or universal; rather than all miners working on the main branch, some temporarily mine competing branches—possibly at different heights. A side branch *could* grow to overtake the main chain. As we shall see, in practice side branches either fall away or overtake the main branch quite quickly (e.g., within one or two blocks).

ORPHAN AND UNRECOGNIZED BLOCKS. Suppose a node receives a block that verifies as valid (above). The block may extend the main chain, a side branch, or neither (if its header hashlink matches no parent block previously seen, including all side branches). The first case is common; the main chain remains unchanged. In the second case, it is possible that a side branch now overtakes the main branch; a miner with this view is expected to switch over to the new main branch. In the third case, with no parent in sight from the local view, the block goes into a holding tank called the *orphan block* pool. One source of orphan blocks is a block emerging relatively quickly by chance, and arriving at a peer ahead of its parent; when the parent block finally arrives, the orphan is moved into the relevant case. Blocks that fit side branches (the `prev-block-hash` field fitting a side-chain parent)—let's call them *side-pool* blocks—are, like orphan pool blocks, kept in the short term, as they may come into play. But as a main branch emerges to be a clear winner over increasingly distant side branches, blocks on distant side branches in essence become permanently excluded from the main branch. In this way, some candidate blocks, *despite a valid proof of work*, fail to be included in the consensus blockchain. Sadly, the unlucky miners of such *unrecognized blocks* receive no block rewards;[11] and over time, such blocks effectively cease to exist, as they are not "on" the blockchain. *C'est la vie.*

CONFIRMED BLOCKS. As an increasing number of blocks are built on top of a given block, the probability of it failing to become part of the consensus blockchain diminishes rapidly, and eventually it is considered "on the blockchain" as an unchangeable outcome.

[11] ‡Some experts use the word *stale* for a block not on the main chain, while others use it for any block arriving after a same-height block was already seen (such a block often ending up matching the first case). Other terms for side-chain blocks are *uncles* or *ommers* (in Ethereum) and *orphans* (in Bitcoin)—but the latter causes confusion with a dictionary interpretation of (true) *orphan* to mean *parentless* (as in our use).

The block—and its transactions—are then said to be *confirmed*. By rule of thumb, a depth of 6 suffices in practice for confirmation; less cautious users may go with fewer. An *unconfirmed* block remains subject to the risk of *double spending* or becoming *unrecognized* due to branch reorganization, in which case its transactions will not be valid.

‡**COINBASE OUTPUT HOLD.** As an extra rule, no coinbase transaction output can be spent until the coinbase block is confirmed at least N times—to rule out any practical chance that after the block reward is spent, the blockchain is reorganized such that the coinbase block is no longer on the main chain (thereby becoming invalid). $N = 100$ is used at the time of writing, implying on average a wait of 16 hrs 40 min at 10 min/block.

INCENTIVE TO EXTEND MAIN BRANCH. As a heuristic expectation, *miners will build on the main branch*. This is supported by two circumstances. 1) In preparing a block header, a miner must explicitly choose a branch to extend, as a block's `prev-block-hash` field specifies a parent. 2) If a miner completes a proof of work but their block perishes on a side branch, the miner loses the block reward. Thus committing to build on a known side branch reduces expected earnings (odds are, it is wasted effort). The rule that a block reward stands only if its block settles on the main chain incentivizes *compliant* behavior.[12] Note: if a non-compliant miner includes an invalid transaction in its block, other miners lose incentive to build on it, as doing so would forfeit *their own* potential block reward, as other compliant peers will abandon that branch, on seeing the block fail validity tests.

BLOCK SELECTION AS VOTING. In review, as nodes see new blocks, the invalid are dropped, the others fitted to a branch at a unique fitting point using the block header's parent pointer. Each new block may rearrange leadership between main and side branches. Using local information, miners independently choose which branch to build on—each incentivized to build on what is the main branch from their view. Each miner, by choosing a block to build on, in essence casts a vote for a main branch, "voting with their feet".

Example *(How forks resolve quickly).* Suppose two miners are building on the same parent block Z (Fig. 13.9a), and at about the same time, each completes their block and sends it to their peers. What will happen? We expect peers to begin mining new blocks on top of whichever of these two blocks, A or B, they receive first. Now some miners are working to extend branches with parent A, and some with parent B (Fig. 13.9b); call them branchA miners, and branchB miners. We say the blockchain has a *fork*.

Now a miner in one of the two sets will succeed first—say branchB—and immediately propagate new block B'. Both branchA and branchB miners receive B'. BranchB miners see their (main) branch has been extended, and immediately start building new blocks with B' as parent, to extend branchB yet further. BranchA miners see that "their" main branch has now fallen behind; they could continue, hoping to get lucky twice (first finding a block A' to catch up, and then a second to re-take the lead), but they are already one block behind, which may be a sign that more computing power was already on branch B. Even if a valid A' is found, by that time, branchB miners may have a B'' (Fig. 13.9c). And branchA miners recall: no block reward can be claimed unless your block ends up

[12] Here *compliant* means consistent with a Bitcoin reference implementation. Aside from the reward incentive, Bitcoin also relies on an assumption that a majority of computing power rests with compliant miners.

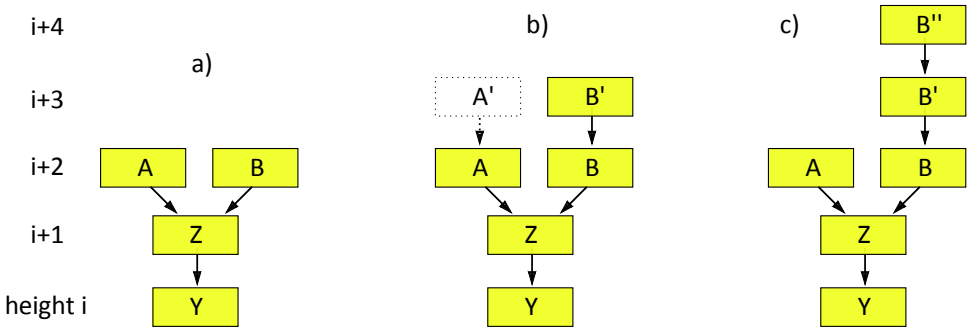

Figure 13.9: Resolution of a blockchain fork (model scenario). (a) If blocks A and B emerge around the same time (say this event happens with constant probability p), miners start building on A or B, whichever they receive first. (b) Blocks A' and B' may again emerge at about the same time, but the probability of t such events in a row is an increasingly unlikely p^t (decaying exponentially as t grows). Typically one block, say B', emerges sufficiently ahead of others, that a majority of miners receive it first and start building on it. (c) By the time block B'' emerges, extending B', almost all miners have abandoned work on side branch A and recognize B as the main branch.

on the main branch. Alas, the odds suggest that it is time to switch to branch B, and leave gambling to the weekend horse races. As some branchA miners switch to join branch B, the probability increases that branch B pulls further ahead. Branch A becomes a side branch with fewer miners, and soon none (if they are paying attention and receive timely updates from their network links). The fork is now resolved, in favor of branchB miners.

Full nodes and mining pools. Devices may connect to the Bitcoin network as *nodes*. (For *light nodes*, see Section 13.7.) *Full nodes* validate and forward individual transactions and blocks, and maintain UTXO pools. They are the foundation of the peer network, and miners rely on access to their functionality—including to manage transactions in memory pools, for inclusion in new blocks. The core mining activity (building new blocks) is distinct from peer sharing of data, and not all full nodes are also miners. One might thus expect fewer miners than full nodes. However, miners may form a *mining pool* (also page 404), to collectively work on the same hashing puzzle for an in-progress block proposed by a *pool leader* (with block rewards split by agreement); one full node then serves many miners. By some estimates, there are many more miners than full nodes; in practice, the ratio of full nodes to miners is not precisely understood.

Exercise (Non-compliant miner). If a miner prepares a candidate block (with valid proof of work) whose coinbase transaction claims a larger block reward than allowed, and circulates this block while no other candidate blocks at that height are also ready, will this block be accepted into the blockchain? Why or why not?

‡**Exercise** (Blockchain forks reconverging). Using a series of diagrams, explain the sequential stages as two groups of miners compete as the blockchain forks, work to extend different branches, then quickly reconverge on the main branch. (Hint: [1, pp.199-204].)

13.7 ‡Simple payment verification, user wallets, private keys

Here we discuss *thin clients* and issues related to Bitcoin *wallets* and storing private keys. A summary of cryptographic algorithms used in Bitcoin is also given.

LIGHT NODES AND SPV. A prudent seller who receives bitcoin as payment in a transaction T may wish to wait until T is confirmed before releasing goods—in case the buyer is double spending. A confirmation can be observed without running a full node by a method called *Simplified Payment Verification* (SPV). Indeed, typical client software is not run as a full Bitcoin node processing all blocks and their transactions; that requires a significant investment of computing time and memory (the blockchain is relatively large), and regular users are interested in their own transactions, not mining. In contrast, SPV requires only the block headers from the top block to the block containing the target transaction, plus minor further data as explained next. SPV clients (also called *thin clients* or light nodes) may be used in wallet software (page 397); newly originated retail transactions sent into the network are processed by full nodes. Thin clients have a reduced data footprint—aside from using block headers (smaller by a factor of about 1000 than full blocks), they collect and keep transaction details only for transactions of specific interest, e.g., payments for a given user's addresses (keys).

SPV PROCESS. The idea is to establish that transaction T is legitimately part of block L. If L is then observed to be at depth k, this gives confidence that T has k confirmations. Inclusion of T in L is established by Merkle *path authentication* (below). The chain from the top block to L is verified through block headers (the proofs of work are checked). For confidence that these headers are themselves part of the authentic chain, it suffices to have confidence in the authenticity of the top block retrieved; hashlinks are used to verify the rest. One method is to retrieve and cross-check the top block from multiple nodes (assuming at least one source is honest and not under attacker influence); another is to obtain the top block from a single trusted source and channel.

MERKLE PATH AUTHENTICATION. Relying on collision resistance properties of hash functions, the block header `merkle-root` field is used to verify that a transaction belongs to a block as follows. Suppose a given transaction is asserted to be part of a block (say transaction 3 in it), and a client wishes to confirm this. Let the block header's `merkle-root` value be x, and txID-3 the transaction's hash. Using txID-3 along with the off-path values (txID-4, H1.2, H5.8) from the *authentication path* in Fig. 13.10, the client proceeds to recompute a candidate root value, and as the final test compares that for equality to x (from the integrity-checked block header). The client may obtain the off-path values by a query to peers—any values that result in matching x suffice for a *proof of inclusion*, confirming that the transaction is part of the block. (The peers need not be trusted—the assurance relies again on cryptographic hash function *collision resistance*, and an authentic root value.)

Exercise (SPV confirmation). SPV verifies block headers (from top to target block), but not the individual transactions in intervening blocks. Why is that not necessary?

Exercise (Authenticity of block headers). Two methods to ensure authenticity of block headers are noted above. A third is to obtain the full set from any source (not nec-

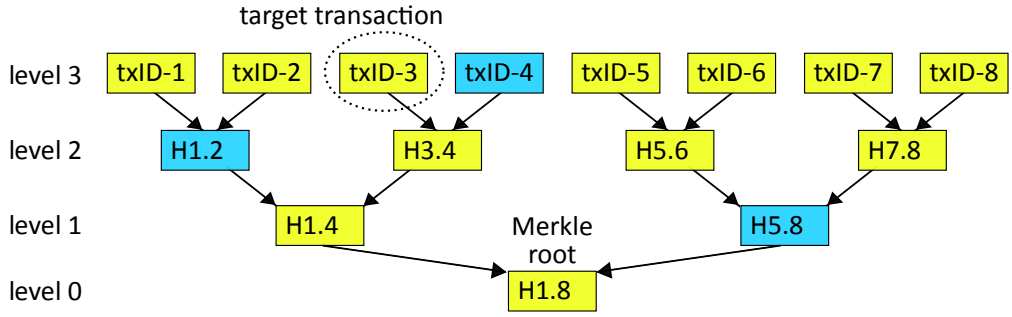

Figure 13.10: Merkle path authentication. Off-path values for txID-3 are shaded (blue). A block having N transactions has an *authentication path* involving only $\lg_2(N)$ values.

essarily trusted), and confirm by hashlinks that they lead into a genesis block hard-coded into the verifying client. Discuss advantages and disadvantages of the three alternatives.

Exercise (SPV and UTXOs). The input of a new transaction W claims an output from an old transaction T. The buyer is in a hurry; the seller prefers to wait for 6 confirmations of W before releasing any goods. The buyer suggests to the seller: "Use SPV to verify that T is a valid transaction; that will confirm that the output that I am claiming is valid cash recognized by Bitcoin." What should the seller do, and why?

PRIVATE KEY: GUARD AS CASH. Bitcoin transactions send currency to addresses. Digital signatures claim the currency, requiring access to private keys. Consequently:

1) loss of the private key means (unrecoverable) loss of the currency; and
2) theft of the private key allows theft of the currency.

Furthermore, transactions are irrevocable and irreversible. This makes Bitcoin attractive to criminals and thieves, and is in contrast to conventional banks, where regulations typically require long-term relationships with, and contact details for, real-world participants.

PRIVATE KEY RISKS AND PASSWORDS. The risk of losing a private key is increased by possible loss of any information or device used to generate or protect it. This includes passwords or passphrases used to derive a symmetric encryption key (*passkey*) to protect a private key, and seeds in hierarchical wallets (page 398). It is therefore risky (inviting loss) to store a private key encrypted under a passkey while relying entirely on human memory to recall the password. Specific risks to private keys thus include:

a) loss of access due to forgotten passwords,
b) loss of access due to hardware failure or lost information, and
c) theft, possibly combined with *offline guessing attacks* (Chapter 4).

As noted above, the risk is permanent loss of the associated currency. The disadvantages of passwords (Chapter 3) have severe implications here—if a private key (or related passkey) is misplaced, forgotten, or lost, there is no equivalent to the password recovery (rescue) services associated with web account passwords. The risk of theft (consider malware, device theft, cloud storage break-in, cloud backup) followed by successful offline guessing can be mitigated by avoiding user-chosen passwords.

PRIVATE-KEY MANAGEMENT AND WALLETS. In the context of Bitcoin, *wallet* refers to a means to store and manage key pairs. Beyond storing private keys, wallet functionality may include a convenient user interface, and being able to track all transactions related to unspent outputs paid into a user's addresses. Wallet strategies, broadly either online (higher convenience, lower safety) or offline, imply two storage categories.

i) *Hot storage* (private keys reside on devices regularly connected to the Internet). This includes *client wallet* software running on a user's own device (e.g., local desktop, laptop, smartphone); and *cloud wallets*, with user private keys managed by (or available to be retrieved from) a web-based service. The risks here include theft by malware (e.g., local), Internet-based theft, and third-party risks (cf. exchanges, below).

ii) *Cold storage* (private keys are on offline media, ideally never directly connected to the Internet). This includes *paper wallets* (with keys printed then stored in a safe place, made human-readable by encoding as hexadecimal strings or base58 or 2D barcodes); storage on a USB key or other physical token; and *hardware wallets* (special-purpose hardware devices allowing private-key computations, but not private-key export). Payments can be made to an address whose private key is offline (since payment does not itself involve a receiver's private key), if the address itself is known.

CUSTODIANS AND EXCHANGES. As noted, coin owners should guard Bitcoin private keys as cash. Analogously to cash being deposited in traditional banks, Bitcoin *custodian services* are available, through access to users' private keys. Separately, Bitcoin *exchanges* facilitate conversion between fiat currencies and cryptocurrencies, and may also offer custodian services. As these introduce intermediaries as trusted third parties, this trades off one risk (a user losing its own private keys) for others (trustworthiness of a third party's integrity and viability, and its ability to protect its private keys from theft).

Exercise (Bitcoin exchange bankruptcies). (a) Summarize the popular media reports of the 2014 bankruptcy of the (Japanese) MtGox exchange (hint: wikipedia). (b) Give a technical summary of Bitcoin *transaction malleability* as it relates to MtGox (hint: [21]). (c) Summarize what is known about the 2019 bankruptcy of the (Canadian) QuadrigaCX exchange and CEO Gerald Cotten (hint: online resources).

HASH FUNCTIONS. Table 13.5 summarizes the use of hash functions in Bitcoin.

Use	Example	Algorithm
mining (header hash)	page 387	dhash = SHA256(SHA256(·))
`txID`, `prev-out-txID`	Table 13.1	dhash (for transaction identifiers)
`prev-block-hash`	Table 13.4	dhash (for block/block header identifiers)
`merkle-root`	Table 13.4	dhash
addresses	page 377	RIPEMD160(SHA256(·)); dhash for checksum
ECDSA signatures	page 397	dhash (may differ in special cases)

Table 13.5: Hashing algorithm use in Bitcoin. "dhash" is short for double-SHA256.

ELLIPTIC CURVE SIGNATURES. Hash functions are noted above, and Bitcoin does not specify encryption (its use by wallet software to protect private keys is independent of the Bitcoin protocol). For digital signatures, Bitcoin uses the elliptic curve digital

signature algorithm (ECDSA) and an elliptic curve identified as `secp256k1`. It is believed to provide 128-bit security, i.e., a successful attack should require about 2^{128} symmetric-key operations or SHA256 invocations. Private keys are 256 bits; signatures are 512 bits. Public keys are 512-bit points (x, y), or reduced to 257 bits in *compressed* format with the x component and a single bit allowing the y component to be computed from the curve equation $y^2 = x^3 + 7 \pmod{p}$; the extra bit selects one of the two square root solutions.

DETERMINISTIC KEY SEQUENCES. Bitcoin imposes a key management burden on users, by expecting use of a new key pair for each transaction. How should a normal person repeatedly generate public-private key pairs, make a new public key available (to receive each transfer) and securely back up each private key, without ever connecting cold to hot storage? As a popular response, Bitcoin wallets (software applications) support schemes deriving pseudo-random sequences of private keys from a (secret) master or seed key, providing a *master public-key property*: sequences of corresponding public keys (addresses) can be derived from a separate, non-secret seed. Giving the non-secret seed to a network-connected wallet enables public-key generation for countless incoming transfers; private keys remain offline (cold). Securely backing up one secret seed replaces backing up long lists of secrets. Extending such schemes to multi-level trees of keys results in *hierarchical deterministic* (HD) key schemes. A related concept is *watch-only* wallets (and addresses): a network-connected wallet (client or service) is given a set of public keys, in order to monitor for all transactions involving the corresponding addresses, without risk of outgoing transfers (private keys are withheld, thus *watch-only*). This works for any public key, but often involves seed-based sequences (i.e., master public keys, as above).

Example *(Deterministic key management).* Such deterministic schemes for DSA-type digital signatures rely on the simple relationship between a signature private key x and matching public key $g^x \pmod{p}$. (Here we use Diffie-Hellman notation; generator g and prime p are suitable public parameters.) For a sense of how such schemes work, consider a sequence of keys relying on two parameters: our previous x (a secret random number), and a new k (random number). Index i identifies a stage in the sequence, "::" denotes concatenation, and H is, say, SHA256. The ith private and public keys are now:

$$\text{(private)}\ \ x_i = x + H(i :: k) \qquad \text{and} \qquad \text{(public)}\ \ g^{x_i} = g^x \cdot g^{H(i::k)} \tag{13.9}$$

Addresses are derived from public keys in the usual way. Using $k = g^x \pmod{p}$ eliminates the need for a separate random number k. Generating the public-key sequence requires knowing g^x (and k if distinct); knowing that does not allow derivation of private keys, but does allow discovery that a sequence of keys (addresses) are related. Note: in the simple scheme of equation (13.9), every individual private key x_i should be protected as strongly as the master private key x, because compromise of any single x_i allows recovery of x (by the left equation), allowing an adversary to compute all private keys in the sequence.

ONGOING CHALLENGES. Many issues remain for Bitcoin and related cryptocurrency and blockchain systems. While our aim has been a relatively concise introduction, with more focus on understanding how things work than why or design options or ongoing research (Section 13.9), we note among the challenges: scalability (number of transactions per second or block), massive power consumption by mining, centralization due to mining

pools (page 404), privacy, use of cryptocurrencies in illegal activities, governance issues, usability and asset safety (including both theft and lost access to private keys).

13.8 ‡Ethereum and smart contracts

Ethereum has attracted enormous attention since its debut in July 2015. One of many Bitcoin-inspired cryptocurrencies, it builds on the idea of blockchain-based public ledgers. However, far beyond simple cryptocurrency transfers, it strives to deliver a decentralized computing platform for executing *smart contracts*—specialized programs designed to execute autonomously, recording data and implementing agreements involving virtual goods. Here we give an overview of how Ethereum works and a sense of its major differences from Bitcoin, in a selective tour to motivate readers to explore further.

Explicit account balances. Ethereum's currency, *ether* (ETH), is created by a block mining process resembling that of Bitcoin. ETH can be transferred across Ethereum accounts by transactions, and exchanged for other currencies. Unlike Bitcoin, which stores values as unspent transaction outputs, Ethereum explicitly tracks per-account balances, using nonces (as counters) to prevent replay of transactions as noted next.

Types of accounts. Ethereum has two types of accounts. An *externally-owned account* (EOA or user account) is controlled by a private key, and sends signed transactions to be included in blocks on Ethereum's blockchain. EOAs have three state values:

$$(\text{address},\ \text{balance},\ \text{nonce}) \tag{13.10}$$

An EOA identifier or *address* is the rightmost 20 bytes of the hash of the account's ECDSA public key (using KECCAK-256, the basis for SHA-3); *balance* is the account's current value in *wei* (1 ETH $= 10^{18}$ wei); *nonce* counts how many transactions an EOA has sent.

The second type of account is a *code account* (or *contract account*). It is controlled by an executable program called a *contract*, which runs on the Ethereum virtual machine (EVM, below). The code runs when a contract receives a *message* (below), and may access Ethereum storage, write new state, send new messages, and create further contracts (bottom left, Fig. 13.11, page 402). A code account has five defining state values:

$$(\text{address},\ \text{balance},\ \text{nonce},\ \text{acctStorageRoot},\ \text{codeHash}) \tag{13.11}$$

A code account's *nonce* counts how many contracts it has created; its *address* is the hash of the sender's address and nonce from the transaction that created it. An *acctStorageRoot* value binds storage (persistent state) to each code account; it is the 32-byte hash root of a Merkle Patricia Tree (page 403), allowing authentication of any-sized persistent state, with underlying data, e.g., stored in a database and retrieved as needed, its integrity verifiable using the root value. Finally, *codeHash* is the hash of the contract bytecode, uniquely identifying the executable code that runs when a message is sent to the account.

Transactions and messages. EOAs create Ethereum *transactions*, which are signed data structures (details below). They are commonly used to:

a) transfer value between EOAs (for simple payments),

b) invoke existing contracts, and

c) create new contracts (initializing them and setting up their executable code).

Both transactions and contract executions may result in *messages* to (other) contracts; these also transfer value and data between accounts. Messages to contracts may trigger responses; this instantiates functions, and is how contracts interact.

EVM EXECUTION, MEMORY AND LOCAL MACHINE STATE. Ethereum nodes independently execute contracts on their local copy of an *Ethereum virtual machine* (EVM). In contrast to Bitcoin's minimal scripting language with limited opcodes and no loops (albeit thereby avoiding infinite loops and related deliberate abuses), the EVM instruction set is *complete* in the sense that it enables general computer programs (called *Turing completeness*), with one major constraint: computation is gas-limited (*gas* is explained next). Local EVM execution involves three flavors of memory:

1) a conventional *run-time stack* (32-byte values are PUSHed and POPed);

2) (non-persistent) EVM *execution memory*; and

3) (persistent) EVM *storage*, holding key-value pairs (32 bytes for each of key, value).

A program counter (PC) indexes the next byte of EVM code to execute, and a GAS register tracks gas. We summarize an EVM's operational *machine state* as:

$$\text{local EVM machineState} = (\text{PC},\ \text{GAS},\ \text{exec_mem},\ \text{stack},\ \text{current_code}) \tag{13.12}$$

The machine has access to relevant account state, (13.10) and (13.11), including the persistent storage underlying `acctStorageRoot` and `codeHash`, which is the EVM bytecode denoted current_code in (13.12). EVM execution also has access to metadata from the current transaction and incoming messages resulting from it, and the header data of the block associated with the transaction; and can send a return value (as a byte array).

FEES AND GAS CONCEPT. Ethereum uses a fee model designed to protect resource consumption from both deliberate (malicious) actions and the effects of benign errors. The cost of resources used by transactions and contract execution must be paid for by accounts. A *fee schedule* (page 401) specifies the cost of each resource in units of *gas* (generalizing the idea of buying gas for a car); bills for resources consumed are in gas units. Gas is bought using ETH, but the price of gas is not fixed. Instead, a transaction offers *gasPrice*, a price per unit of gas used. By including a transaction in a block, the miner accepts the offered gas price. Minimum expected gas prices are known or advertised.

TRANSACTION FIELDS. We model a transaction by a data structure with fields:

$$(\text{toAddr},\ \text{valueTo},\ \text{data},\ \text{nonce},\ (\text{gasLimit},\ \text{gasPrice}),\ \text{sendersSig}) \tag{13.13}$$

Messages are similar, but without `sendersSig`. The fields are used as follows.

`toAddr`: the receiver's account address (page 399), or NUL for contract creation.

`valueTo`: amount to transfer to the receiver (in wei). If the receiver is a contract, its code is run. For a contract creation, this is the new account's starting balance.

`data`: for messages, this provides input to contracts. For contract creation, it provides startup code that is run once as initialization, and returns the long-term EVM code to be run on each subsequent invocation of the contract.

`nonce`: this is compared to the sender account's nonce value (counter) when a transaction is executed by a miner (and processed by peers), to stop transaction replays.

`gasLimit`: maximum number of gas units the sender releases to execute this transaction (including costs associated with any resulting messages).

`gasPrice`: offered payment per unit of gas (in wei per gas unit). The miner that mines a transaction collects fees including all gas spent executing the transaction. Any gas allocated but unused upon completion is returned to the sender. The *up-front cost* of a transaction is computed as: $v_0 = (gasLimit * gasPrice) + valueTo$.

`sendersSig`: this provides data allowing verification of the sender's signature, and also includes information identifying the sender's address.

RECEIPTS AND EXECUTION LOGS. Dedicated EVM opcodes are used to *log* or record, for external access, information on events related to contract execution. For each transaction involving contracts, a *receipt* is generated. It includes a status code, the set of logs generated, and a *Bloom filter* data structure for efficient use of the log information.

EVM FEE SCHEDULE. Gas charges (Table 13.6) based on instruction complexity or system cost apply to EVM opcodes, persistent memory used (EVM storage), and the number of bytes in a transaction (effectively a bandwidth cost). A contract does not pay for its own gas—rather, the invoking transaction does.

EVM charge	Description of Ethereum opcode or storage-related item (gas units: wei)
3–5 gas	ADD, SUB, NOT, XOR; MUL, DIV, MOD
8 gas	JUMP (+2 if conditional), ADDMOD (addition modulo 2^{256})
375 gas	base cost of a LOG operation; +375 per LOG topic, +8 per byte of LOG data
20,000 gas	to store a non-zero value in EVM persistent storage (contract memory)
4 gas	for every zero byte (code or data) in a transaction
68 gas	for every non-zero byte (code or data) in a transaction
21,000 gas	base cost for every transaction
32,000 gas	to create a contract (in addition to base cost)

Table 13.6: Fee schedule, illustrative items [55]. The last four are *intrinsic* gas costs.

MINING REWARDS. The miner of an Ethereum block (or rather, the `beneficiary` specified, Table 13.7) receives the block's transaction fees (gas consumed in executing its transactions), plus a *block reward*[13] of 2 ETH, plus 1/32 of this reward value for including each of up to two *ommer* headers (next paragraph). The `beneficiary` of any so-included ommer block also receives between 7/8 and 2/8 of the block reward, the fraction calculated as $(8-i)/8$ where i counts (from 1 to 6) how many generations older the ommer block is than the mined block. (This rewards some miners even for a valid block that never makes it onto the blockchain.) Aside: whereas Bitcoin's (current) plan halves the block subsidy every four years until a fixed currency production end-date, Ethereum currently has no production end-date and somewhat *ad hoc* rule changes (e.g., the block reward was 5 ETH before dropping to 3, and now 2 as of the time of writing).

[13] Ethereum's *block reward* corresponds to what Bitcoin calls a block *subsidy* in (13.3), page 381.

Field Name	Description
`parentHash`	hashlink to parent block on blockchain
`ommersHash`	hash of the ommers list in the full block (explained inline)
`nonce`	8-byte counter, used in proof of work
`mixHash`	explicit representation of proof of work
`difficulty`	analogous to Bitcoin difficulty (but easier, producing blocks faster)
`block_number`	explicitly stated block number
`timestamp`	approximate Unix time of block birth
`beneficiary`	20-byte receiver address for block reward (e.g., miner's address)
`gasLimit`	bound on per-block gas costs
`gasUsed`	total gas used on this block, including executing its transactions
`logsBloom`	256-byte structure related to block's log entries/transaction receipts
`transactionsRoot`	hash of root of Merkle Patricia Tree of block's transactions (page 403)
`receiptsRoot`	analogous to transactionsRoot, but for transaction receipts in block
`stateRoot`	analogous to transactionsRoot, but for persistent state (world state)
`extraData`	up to 32 bytes further content (at miner's option)

Table 13.7: Ethereum block header. ⋆`Hash` and ⋆`Root` fields are 32 bytes. The header is about 500 bytes (`extraData` varies by 32 bytes); over half of this is `logsBloom`.

BLOCK HEADERS, COMPLETE BLOCKS AND OMMERS. The fields in an Ethereum block header are summarized in Table 13.7 and Fig. 13.11. A full block consists of:

$$(\text{block_header},\ \text{list_of_ommer_block_headers},\ \text{sequence_of_transactions}) \qquad (13.14)$$

The second item in (13.14) is a set of block headers that hash to `ommersHash`. An *ommer* block (gender-neutral for *uncle*, or off-chain sibling of an ancestor) is the sibling of an earlier main-chain block, but which (from the miner's local view) itself did not end up on the main chain despite being a valid candidate block. Ommers play a role in Ethereum's blockchain consensus protocol, which selects not the longest but the *heaviest chain* (Section 13.9). Ethereum gives miners an extra small reward for including block headers of up to 2 ommers in total from any of the past 6 generations. This heuristic aims to mitigate

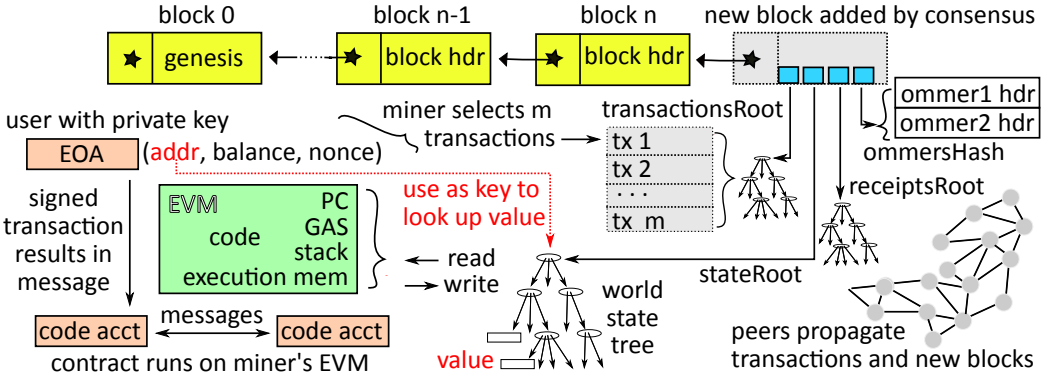

Figure 13.11: Overview of Ethereum components. Components are explained inline.

attacks enabled by more frequent blockchain forks due to the higher block production rate—Ethereum produces blocks every 13–14 seconds vs. Bitcoin's 10 min.

TRANSACTION VALIDATION AND EXECUTION. Following the blockchain model, to be officially recognized, an Ethereum transaction T must be validated and executed at some defining point. Here this implies: T being selected by a miner for inclusion into a new block B, the block header having a valid proof of work, and the block ending up on Ethereum's consensus heaviest chain. What follows is a list of major procedural steps for validating and executing a transaction T (validating an entire candidate block involves verifying the proof of work, the block header, and all transactions).

1. Do transaction validity checks (e.g., verify format, verify sender's signature, confirm the nonce is 1 more than in the sender's account, sender balance $\geq$ up-front cost).
2. Transfer *valueTo* to receiver (from sender).
3. Deduct $gasLimit * gasPrice$ from sender balance, and increment the sender's nonce.
4. Set GAS $:=$ *gasLimit* (as the number of units of gas available).
5. Deduct from GAS all relevant *intrinsic gas* costs (Table 13.6).
6. If *toAddr* is a contract, run its code until code exits or "out of gas" (OOG), deducting costs from GAS as execution runs (per fee schedule).
7. If there are insufficient funds or any other exceptions including OOG, restore all changed state (except the miner keeps fees for gas burned, and the nonce is not rolled back). Otherwise on exit, restore the cost of any unused gas to the sender's balance, and finalize the transaction receipt for this transaction T.

WORLD STATE AND THE BLOCKCHAIN. The blockchain defines what is valid in Ethereum's world, and this is captured as the *world state*—the data associated with Ethereum accounts. This state is publicly verifiable through a block header and a subset of its fields that embed the roots of externally maintained *search trees* (next) for persistent storage, providing a form of *authenticated data structure*. As a result, despite almost all this state being stored off the blockchain, the world state is carried forward in each block. Equation (13.14) and Table 13.7 help us see how Ethereum blocks represent this world state. Compared to 80 bytes in Bitcoin, Ethereum block headers are 500 bytes and rely on this combination of data structures and root values.

ETHEREUM MPT. A *Patricia Tree* data structure (below) can support storage and retrieval for a set of (key, value) pairs, with efficient update, insertion and deletion. Each tree node may hold part of a key (a partial *path*), pointers to other nodes, and a value. Replacing the pointers by hashlinks, so that nodes are referenced by the hash of their content (rather than address pointers) yields a *Merkle Patricia Tree* (MPT). Note that now a path from root to leaf resembles a blockchain; and although computed by a different process, the hash of the content of an MPT root has the same property as a Merkle hash tree root (`merkle-root` in Bitcoin): each is a fingerprint (cryptographically reliable representation) of the content of the entire tree. The hash of an MPT root's content can therefore be used to authenticate the complete (key, value) dataset comprising the tree.

Relying on this, Ethereum uses a custom MPT to store and update the state underlying `stateRoot`, the (hashed) root of the world state tree for accounts. As Fig. 13.11 models, an account *address* is used to look up a leaf value of (*balance, nonce, acctStorageRoot, codeHash*) from (13.11), where *acctStorageRoot* itself roots a further tree in the case of a code account. Similarly, the MPT under a block header's `transactionsRoot` is keyed by transaction indexes in the block. The content of each tree node (including root) is stored in a database and looked up by hash value. Confused? The next exercise may help.[14]

Exercise (MPT details). Given a set of (key, value) pairs, a *search tree* can be built, allowing *value* to be looked up at a leaf node, using symbols of *key* in sequence to trace a unique path from a tree root onward.[15] (a) Explain how *tries* (prefix trees) and *Patricia Trees* (compressed tries) work. (Hint: [39].) (b) Explain how Ethereum's *MPT* works, including its three node types (extension, branch, and leaf). (Hint: [16, 25, 57]; treat the prefix encoding for nodes as a minor implementation detail, rather than a major focus.)

TRANSACTIONAL STATE MACHINE. Ethereum transactions cause *state transitions* that transfer currency and data between accounts, optionally involving contract execution. At the world state level, Ethereum is a *transaction-based state machine*, with a *state transition function* F taking two inputs—a world state S_t at time t, and a next transaction T—and producing a new world state: $F(S_t, T) \to S_{t+1}$. We note the following.

a) Like Bitcoin, Ethereum relies on a *replicated state machine* and mining-based consensus, but in contrast its official world state is *explicitly* summarized as each block emerges.[16]

b) Each block defines a specific ordering of transactions and thus contract executions. Each contract execution runs atomically, in that the world state is not at risk of changing due to other transactions or contracts running in parallel. Thus the results of a miner's local EVM running contracts (with its local view of world state) become the world reality if the miner executing a transaction (in block mining) has their block accepted into the blockchain.

c) Contracts execute only as a result of EOA-triggered transactions, and cannot on their own schedule events to occur at future times, unrelated to the current transaction.

ETHASH AND MINING POOLS. Bitcoin mining is now dominated by *mining pools* (page 394), with collaborative groups of miners sharing resources and rewards, typically leveraging large investments in special-purpose hardware, notably ASICs optimized for SHA256. Centralization of control conflicts with the goal of decentralized consensus. Ethereum aims (perhaps optimistically) to address this by using *Ethash*, a mining hash function designed to be ASIC-unfriendly, favoring commodity architectures over customized hardware. It requires considerable memory, with internal operations heavily data-dependent and based on a seed updated every 30,000 blocks (about 5.2 days).

PROGRAMMING LANGUAGES. Several languages are available for writing Ethereum programs (contracts), including Solidity and Vyper (newer), and lower-level LLL (older). Contract programs are compiled to bytecode for EVM execution.

[14] As you will by now understand, our exercises often serve to cover additional details from further sources.
[15] Confusingly, such a search tree is also called a *trie*—which is still pronounced *tree*, as in re*trie*ve.
[16] That this represents the true world state is conditional on the block being accepted as a consensus block.

SMART CONTRACTS. Ethereum delivers a computing model for what are called *decentralized applications* (DApps), with the same program independently run on physically distant machines and computation verified by peer nodes and consensus, without human involvement. The vision is that arbitrary sets of rules, precisely captured by programs often called *smart contracts*, can programmatically enforce transfers or agreements involving digital assets, free of unpredictable regimes or jurisdictions, ambiguous laws and external control. For context, we note some prominent contract categories (use cases).

1. Financial contracts, including insurance and group-funded ventures (DAO, below). This goes beyond currency creation, transfer and management by simple exchange contracts.
2. Games and digital collectibles based on cryptocurrencies. (This may overlap the next category.) Some players may be motivated by economic gain more than conventional fun.
3. Registration, management and transfer of virtual properties. Examples are name registration (Namecoin, below); trading and ownership tracking of unique digital assets (NFTs, below); and *authorship attribution* systems that may reward content producer-publishers.
4. Digital notarization of records for real-world physical asset transfers. Ownership registration and history (provenance tracking) use the blockchain as an immutable, timestamped log; linking a physical asset to the digital world here requires human involvement.
5. Virtual gambling and program-based casinos. *Ponzi schemes* are a related sub-category.

Exercise (The DAO). Smart contracts have been heavily promoted, but come with challenges. For example, contract immutability is at odds with patching errors via updates. The 2016 attack on *The DAO* (Decentralized Autonomous Organization) highlighted this. Summarize this event and its warning for the future of smart contracts (hint: [50]).

Exercise (Namecoin). A Bitcoin-like system called Namecoin supports a decentralized name registration and resolution system, built as a registry of name-value pairs. Summarize its history, technical design, and deficiencies (hint: [31], also [42, §10.7]).

Exercise (Non-fungible tokens). (a) Explain the *non-fungible token* (NFT) concept. (b) Explain how Ethereum, NFTs, and EIP-721 are related (hint: [23]). (c) Give a technical summary of *CryptoKitties* and how digital cats are bred and traded (hint: [29, 49]).

Exercise (Heaviest chain). Compare, using a diagram, Bitcoin's main-branch consensus approach and Ethereum's heaviest-observed chain (hint: [51, Fig.3], [47]).

13.9 ‡End notes and further reading

Bitcoin was proposed in 2008, including Section 13.7's *SPV* process [41]; the genesis block emerged in January 2009. Greenberg [27] pursues connections between its inventor Satoshi Nakamoto (pseudonym) and Hal Finney. Bonneau [13] summarizes Bitcoin's first six years; for details see Antonopoulos [1] and Narayanan [42]. Zohar [58] discusses myths and notes challenges; for origins of the underlying technology, see Narayanan and Clark [43]. For when blockchain technology is applicable, see Ruoti [48]. Böhme [11] gives an economics-focused summary of Bitcoin properties and risks. Among the many empirical analyses of Bitcoin, Moore [38] found that 18 of 40 *Bitcoin exchanges* failed

over a three-year period (2010–2012), and 38 of 80 over 2010–2015; see also Böhme [12] for case studies on cryptocurrency vulnerabilities. Möser [40] analyzed all transaction fees from 2009–2014. Bartoletti [6] studied practical use of the script opcode OPRETURN. For mining hardware including ASICs, see Taylor [53] (cf. [42, Chapter 5]).

Bitcoin has a well-known *reference implementation* (Ethereum has several). The *Protocol documentation* page of the Bitcoin community wiki [9] describes the reference client behavior, which implicitly specifies the Bitcoin protocol. A Bitcoin *Developer Reference* [10] provides technical details and API information to encourage development of Bitcoin applications. *Blockchain explorers* (an extensive list is available [8]) and related tools (e.g., BlockSci [32]) enable search, display and analysis of public blockchain content. Hashlinks may be characterized as specifying a verifiably unique object as their target, and are also called *hash pointers*; linked lists using them are called *authenticated linked lists*. Merkle *hash trees* (authentication trees) date to a 1979 thesis [37, Chapter 5]. *Distributed consensus* dates back to Lamport's *Byzantine Generals problem* [33].

The *Bitcoin Improvement Proposal* (BIP) process is explained by Dashjr [20]. BIP32 [56] popularized *hierarchical deterministic* (HD) and *watch-only* wallets; see Gutoski [28] for HD wallet flaws and references, and Eskandari [24] for Bitcoin key management, the usability of client software, and wallet strategies. BIP141 [34] (*SegWit* for segregated witness), activated 24 August 2017, separates unlocking signatures and scripts from a transaction's input section, allowing maximum block size to increase from 1 Mb to 2–4 Mb (depending on transaction details). On Bitcoin cryptography, for ECDSA see Johnson [30], FIPS 186 [44], and Brown [15] for the `secp256k1` curve; for SHA256 see FIPS 180 [45]. Menezes [36, page 350] summarizes the RIPEMD160 specification [14]. KECCAK-256 is the basis for SHA-3 [46] algorithms, and used in Ethereum.

Ethereum's white paper [17] giving Buterin's 2013 vision includes two short contract code examples. The yellow paper [55] formally specifies the Ethereum protocol, including its modified *Merkle Patricia Tree* (cf. exercise, page 404),[17] fee schedule, EVM and *Ethash* (resp. in appendices D, G, H, J); a companion beige paper [19] offers a gentler overview. Informally, Ethereum *messages* are sometimes called *internal transactions* (but never appear on the blockchain). `Ethereum.org` [26] provides collected resources, promoting Ethereum as "the world's programmable blockchain". Ethereum uses a simplification [47] of a heuristic called Greedy Heaviest-Observed Sub-Tree (GHOST) [51], basing chain selection on the *heaviest path* (involving the most computation); and plans to migrate to a *proof-of-stake* mining model [5]. Atzei [2] systematizes programming pitfalls and attacks on Ethereum smart contracts, as well as Bitcoin's use for smart contracts [3] despite a primitive scripting language and vulnerabilities (cf. [4]). Regarding *smart contracts*—which are arguably neither smart nor contracts—for early formative ideas see Szabo [52], for an introduction to programming them see Delmolino [22], for empirical analysis see Bartoletti [7], and for broader discussion of *decentralized applications* see Cai [18]. Potentially irrecoverable financial losses (as with The DAO, page 405) motivate greater attention to security analysis methodologies and tools for smart contracts [54, 35].

[17] *Patricia* is from: Practical Algorithm to Retrieve Information Coded in Alphanumeric, a 1968 paper.

References (Chapter 13)

[1] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly. Dec 2014, openly available, https://en.bitcoin.it/wiki/Mastering_Bitcoin. (Second edition: 2017).

[2] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust (POST)*, pages 164–186, 2017. Springer LNCS 10204.

[3] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino. SoK: Unraveling Bitcoin smart contracts. In *Principles of Security and Trust (POST)*, pages 217–242, 2018. Springer LNCS 10804.

[4] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino. A formal model of Bitcoin transactions. In *Financial Crypto*, pages 541–560, 2018.

[5] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. SoK: Consensus in the age of blockchains. In *ACM Advances Financial Tech. (AFT)*, pages 183–198, 2019.

[6] M. Bartoletti and L. Pompianu. An analysis of Bitcoin OP_RETURN metadata. In *Financial Cryptography Workshops*, pages 218–230, 2017. Springer LNCS 10323.

[7] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Financial Cryptography Workshops*, pages 494–509, 2017. Springer LNCS 10323.

[8] Bitcoin community. "Block chain browser" page. https://en.bitcoin.it/wiki/Block_chain_browser. An example of a popular block explorer is: https://blockchain.com/explorer.

[9] Bitcoin community. Bitcoin wiki, 2020. https://en.bitcoin.it/wiki/ (with "Protocol documentation" page describing the reference client behavior).

[10] bitcoin.org. Bitcoin Developer Reference and Guides (for diagrams). https://developer.bitcoin.org/reference/ and https://developer.bitcoin.org/devguide/index.html.

[11] R. Böhme, N. Christin, B. Edelman, and T. Moore. Bitcoin: Economics, technology and governance. *Journal of Economic Perspectives*, 29(2):213–238, 2015.

[12] R. Böhme, L. Eckey, T. Moore, N. Narula, T. Ruffing, and A. Zohar. Responsible vulnerability disclosure for cryptocurrencies. *Comm. ACM*, 63(10):62–71, 2020.

[13] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *IEEE Symp. Security and Privacy*, 2015.

[14] A. Bosselaers and B. Preneel. *Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*. Springer LNCS 1007, 1995.

[15] D. R. Brown. Standards for Efficient Cryptography (SEC 2): Recommended Elliptic Curve Domain Parameters. Certicom Research, 27 Jan 2010 (version 2.0).

[16] E. Buchman. Understanding the Ethereum trie, Jan. 2014. Online, https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie.

[17] V. Buterin. A next generation smart contract and decentralized application platform, 2013. Ethereum whitepaper.

[18] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung. Decentralized applications: The blockchain-empowered software system. *IEEE Access*, 6:53019–53033, 2018.

[19] M. Dameron. Beigepaper: An Ethereum technical specification. v0.8.5 (15 Aug 2019).

[20] L. Dashjr. BIP2: BIP process, revised. 3 Feb 2016, https://github.com/bitcoin/bips/blob/master/bip-0002.mediawiki. For other BIPs see: https://github.com/bitcoin/bips/.

[21] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and MtGox. In *Eur. Symp. Res. in Comp. Security (ESORICS)*, pages 313–326, 2014. Springer LNCS 8713 (proceedings, part II).

[22] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Financial Cryptography Workshops*, pages 79–94, 2016. Springer LNCS 9604.

[23] W. Entriken, D. Shirley, J. Evans, and N. Sachs. EIP-721: ERC-721 Non-Fungible Token Standard. Ethereum Improvement Proposal no.721, Jan 2018, https://eips.ethereum.org/EIPS/eip-721.

[24] S. Eskandari, D. Barrera, E. Stobert, and J. Clark. A first look at the usability of Bitcoin key management. In *NDSS Workshop on Usable Security (USEC)*, 2015.

[25] Ethereum Wiki. Patricia Tree—Modified Merkle Patricia Trie Specification (also Merkle Patricia Tree). April 2021, https://eth.wiki/en/fundamentals/patricia-tree.

[26] ethereum.org. Online resources from the Ethereum community. https://www.ethereum.org.

[27] A. Greenberg. Nakamoto's neighbor: My hunt for Bitcoin's creator led to a paralyzed crypto genius. *Forbes*, 25 Mar 2014.

[28] G. Gutoski and D. Stebila. Hierarchical deterministic Bitcoin wallets that tolerate key leakage. In *Financial Crypto*, pages 497–504, 2015.

[29] X.-J. Jiang and X. F. Liu. CryptoKitties transaction network analysis: The rise and fall of the first blockchain game mania. *Frontiers in Physics*, 9. Article 631665, 1–12 (Mar 2021).

[30] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International J. of Information Security*, 1(1):36–63, 2001.

[31] H. A. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan. An empirical study of Namecoin and lessons for decentralized namespace design. In *WEIS*, 2015.

[32] H. A. Kalodner, M. Möser, K. Lee, S. Goldfeder, M. Plattner, A. Chator, and A. Narayanan. BlockSci: Design and applications of a blockchain analysis platform. In *USENIX Security*, 2020.

[33] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine Generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. Reprinted in *Concurrency: The Works of Leslie Lamport*, ACM, 2019.

[34] E. Lombrozo, J. Lau, and P. Wuille. BIP141: Segregated Witness (Consensus Layer). 21 Dec 2015, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[35] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *ACM Comp. & Comm. Security (CCS)*, pages 254–269, 2016.

[36] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Openly available, https://cacr.uwaterloo.ca/hac/.

[37] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Electrical Engineering, Stanford University, June 1979.

[38] T. Moore and N. Christin. Beware the middleman: Empirical analysis of Bitcoin-exchange risk. In *Financial Crypto*, pages 25–33, 2013. Updated with J. Szurdi in: "Revisiting the risks of Bitcoin currency exchange closure", *ACM TOIT* 18(4) 50:1–18, 2018.

[39] P. Morin. Chapter 7: Data Structures for Strings. Section 7.3: Tries, and Section 7.4: Patricia Trees. Lecture notes—COMP 5408, Carleton University (Canada), Jan 2014, https://cglab.ca/~morin/teaching/5408/notes/strings.pdf.

[40] M. Möser and R. Böhme. Trends, tips, tolls: A longitudinal study of Bitcoin transaction fees. In *Financial Cryptography Workshops*, pages 19–33, 2015. Springer LNCS 8976.

[41] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Oct 2008. Unrefereed paper.

[42] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016. Open prepublication draft (9 Feb 2016) available via the Princeton page: `http://bitcoinbook.cs.princeton.edu/`.

[43] A. Narayanan and J. Clark. Bitcoin's academic pedigree. *Comm. ACM*, 60(12):36–45, 2017.

[44] NIST. FIPS 186-4: Digital Signature Standard. U.S. Dept. of Commerce, July 2013.

[45] NIST. FIPS 180-4: Secure Hash Standard (SHS). U.S. Dept. of Commerce, Aug 2015.

[46] NIST. FIPS 202: SHA-3 Standard—Permutation-Based Hash and Extendable-Output Functions. U.S. Dept. of Commerce, Aug 2015. SHA-3 functions are each based on KECCAK algorithm instances.

[47] F. Ritz and A. Zugenmaier. The impact of uncle rewards on selfish mining in Ethereum. In *IEEE EuroS&P Workshops*, pages 50–57, 2018.

[48] S. Ruoti, B. Kaiser, A. Yerukhimovich, J. Clark, and R. K. Cunningham. Blockchain technology: What is it good for? *Comm. ACM*, 63(1):46–53, 2020.

[49] A. Serada, T. Sihvonen, and J. T. Harviainen. CryptoKitties and the new ludic economy: How blockchain introduces value, ownership, and scarcity in digital gaming. *Games and Culture*, 16(4):457–480, 2021.

[50] D. Siegel. Understanding the DAO attack. CoinDesk media, 25 June 2016, `https://www.coindesk.com/understanding-dao-hack-journalists`.

[51] Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in Bitcoin. In *Financial Crypto*, pages 507–527, 2015.

[52] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. `https://firstmonday.org/ojs/index.php/fm/article/view/548`.

[53] M. B. Taylor. The evolution of Bitcoin hardware. *IEEE Computer*, 50(9):58–66, 2017.

[54] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. In *ACM Comp. & Comm. Security (CCS)*, 2018.

[55] G. Wood. Ethereum: A secure decentralized generalised transaction ledger, 2019. Ethereum yellowpaper, Petersburg version, 8 Jun 2020.

[56] P. Wuille. BIP32: Hierarchical Deterministic Wallets. 11 Feb 2002, `https://en.bitcoin.it/wiki/BIP_0032`. Cf. "Deterministic wallet", `https://en.bitcoin.it/wiki/Deterministic_wallet`.

[57] C. Yue, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, S. Wang, and X. Xiao. Analysis of indexing structures for immutable data. In *ACM Int. Conf. on Management of Data (SIGMOD)*, pages 925–935, 2020.

[58] A. Zohar. Bitcoin: Under the hood. *Comm. ACM*, 58(9):104–113, 2015.