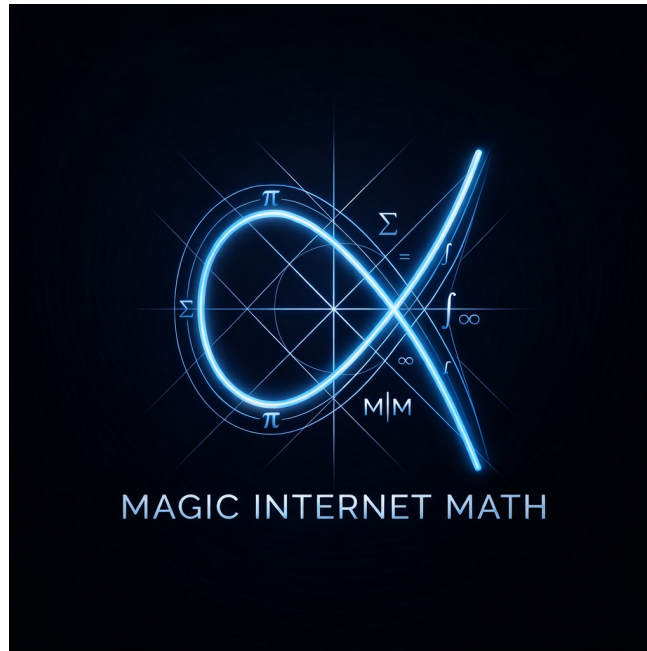




Magic Internet Math

presents:

On Entropy

Probability, randomness, and the safety of a Bitcoin key.

Imagined by Brian Hirschfield

Contents

Prologue: The Weakest Link	v
I Chance	1
1 Counting	2
1.1 The Card on the Table	2
1.2 Choices Multiply	3
1.3 The Doubling Ladder	3
1.4 Order Matters: Arrangements	4
1.5 Order Doesn't Matter: Choosing	5
1.6 The Twelve Words, Counted	5
1.7 Looking Ahead	7
2 Probability	9
2.1 From Counting to Chance	9
2.2 One in N	10
2.3 Independent Stages Multiply	10
2.4 At Least Once: The Complement	10
2.5 A Fair Coin Is a Physical Fact	11
2.6 Looking Ahead	12
3 Expectation and the Long Run	14
3.1 The Question the Thief Actually Asks	14
3.2 Expected Value	14
3.3 The Thief's Arithmetic	15
3.4 About a Million Tries	15
3.5 The Long Run	16
3.6 Looking Ahead	17
II Entropy	20
4 Information as Surprise	21
4.1 Twenty Questions	21
4.2 The Bit as a Measure of Surprise	21
4.3 Surprise, Weighted	22
4.4 Shannon's Measure	22
4.5 What Entropy Is Not	23

4.6	Looking Ahead	23
5	The Attacker's Metric	26
5.1	The First Guess Is Free	26
5.2	Min-Entropy	26
5.3	The RockYou Lesson	27
5.4	What the Metric Buys the Defender	27
5.5	Looking Ahead	28
6	Where Randomness Comes From	30
6.1	The Physical Question	30
6.2	True Randomness	30
6.3	The Computer's Well	31
6.4	Pseudorandom Is Not Random	31
6.5	Inside Bitcoin Core's Well	32
6.6	The Rules	33
6.7	Looking Ahead	33
III	The Key	36
7	A 256-Bit Number	37
7.1	The Number Itself	37
7.2	The Space You Hide In	37
7.3	Where the Key Lives in Bitcoin	38
7.4	How Bitcoin Core Draws a Key	38
7.5	Looking Ahead	39
8	Guessing Games at Scale	42
8.1	Pricing the Search	42
8.2	The Cosmic Rung	42
8.3	The Floor Under All Floors	43
8.4	Batch Attacks Buy Almost Nothing	43
8.5	Looking Ahead	43
IV	When Randomness Fails	46
9	Low-Entropy Keys in the Wild	47
9.1	The Graveyard Shape	47
9.2	Brainwallets: The Lyric That Emptied Wallets	47
9.3	The Wounded Generator	48
9.4	The Mempool Sharks	48
9.5	Looking Ahead	49
10	The Nonce Catastrophe	51
10.1	One Random Number Per Signature	51
10.2	Same k Twice = Game Over	51
10.3	The PlayStation and the Phone	52

10.4	The Fix: Deterministic Nonces	52
10.5	Looking Ahead	53
V	Protecting the Key	55
11	Seed Phrases	56
11.1	The Card, Revisited	56
11.2	Entropy to Words: The Mechanics	56
11.3	What the Checksum Buys	57
11.4	What Alphabetizing Costs You	57
11.5	Passphrases: The Optional Thirteenth Word	58
11.6	Why Bitcoin Core Doesn't Use It	58
11.7	Looking Ahead	59
12	The Operational Checklist	61
12.1	Generation	61
12.2	Storage	61
12.3	Testing and Use	62
12.4	Inheritance	62
12.5	The Checklist	62
12.6	Looking Back	62
A	Notation, Brick by Brick	64
B	Tables: Powers of Two and Probability	66
C	Specimens and References	67

Prologue: The Weakest Link

A Bitcoin private key is a number. Not a metaphor for a number, not a token that stands for one — an actual number, so large that no computer on Earth, nor all of them together, will ever count to it. If that number is chosen properly, guessing it is not hard, not expensive, not impractical: it is *not going to happen*, the way filling a thimble by waiting for the ocean to jump into it is not going to happen. The arithmetic of that safety is simple enough for a child to check, and this book checks it.

And yet Bitcoin keys get stolen. Not often, not randomly, and never by guessing the number — but stolen all the same, in ways that make the news or quietly empty a wallet at 3 a.m. Every one of those thefts has the same shape when you open it up: the number was fine, and the *choosing* of the number was not. Somebody’s phone had a wounded random number generator. Somebody made a key out of a song lyric. Somebody’s signing software rolled the same dice twice. The vault door held; the weakest link snapped somewhere else, and the weakest link is always the randomness.

That is what this book is about. Entropy — the degree to which a choice is genuinely, measurably unguessable — is the one property your Bitcoin security cannot do without and the one almost nobody teaches. Probability is the language entropy is measured in, and counting is where probability comes from. So the road runs: counting, then chance, then entropy, then the key, then the failures, then the defense.

The Method

The method of this book is Bitcoin’s own: don’t trust, verify. Every number printed in these pages is computed, and most can be regenerated with four lines of Python printed right beside them. When the book says there are 2^{132} twelve-word phrases, you can multiply it out yourself. When it says a thief’s million tries at a one-in-a-million lock buy about a 63% chance, you can run the experiment — Chapter 3 runs it. The habit this book wants to leave you with is exactly that one: check it, don’t take it.

No mathematics beyond arithmetic is assumed. Every symbol is built on the page before it is used, and the one time the book needs a formula from elsewhere, it says so and says where.

The Road

Part I, Chance (Chapters 1–3), builds the toolkit from zero: counting outcomes, turning counts into probabilities, and the arithmetic of “how many tries until the thief expects to win.” **Part II, Entropy** (Chapters 4–6), measures surprise itself — Shannon’s measure, the attacker’s measure, and where real randomness physically comes from. **Part III, The Key** (Chapters 7–8), meets the 256-bit number and prices the guessing game honestly. **Part IV, When Randomness Fails** (Chapters 9–10), autopsies the real thefts — brainwallets, broken generators, the nonce catastrophe. **Part V, Protecting the Key** (Chapters 11–12), turns all of it into practice: seed phrases done right, and the operational checklist.

How to Use This Book

Read the chapters in order; each leans on the ones before it. Every chapter ends with exercises, graded by difficulty stars (* to * * *), followed by solution sketches — enough to check your reasoning against, not

model answers to copy. The computational exercises embed their expected outputs, so you always know whether your machine agrees with the book. Boxes marked *The Idea in Code* hold short, runnable Python; run them. If you take one habit away, let it be this: **check it, don't take it.**

Part I

Chance

1

Counting

► The Story

Somewhere near you, perhaps in a desk drawer or a fire safe, there may be a small card with twelve ordinary English words written on it in careful handwriting. Whoever holds that card holds the money. There is no bank to call, no password to reset, no manager to vouch for you — the words are the vault door, and the card is the only key.

How safe is that card? Stripped of everything else, the question is this: if a thief set out to *guess* the words, how many combinations would they have to work through? That is not a question about cryptography, or computers, or anything you need a degree for. It is a question about counting — and counting is something you already know how to do.

This chapter does that counting carefully, and the card never leaves the table. By the end you will have counted the twelve-word phrases yourself, three different ways, and the exact numbers will be printed where you can check them. To get there you will pick up the three tools that answer every counting question in this book: choices that multiply, a ladder that doubles, and the difference between order mattering and order not mattering.

1.1 The Card on the Table

First, pin the question down, because a vague question gets a vague answer. The twelve words on the card are not just any words. They come from a standard, public list of exactly 2,048 English words, and they are written down *in order*. (You will meet the list itself, and the standard behind it, in Chapter 11. For now, all that matters is its size: 2,048 words.)

So the safety question becomes three honest counting questions, and they have three different answers:

- How many twelve-word phrases are there, if any word may be used, *repeats allowed*, order mattering?
- How many, if the twelve words must all be *different*, order still mattering?
- How many, if the words must be different *and order does not matter* — only the set of words counts?

Each of these is a precise question with an exact whole number for an answer, and by Section 1.6 you will have all three. But you cannot jump straight at them, because nobody can multiply twelve 2,048s in their head without a method. The method is the next three sections: a rule about multiplying, a ladder of doublings, and a way to handle order. Then the card gets counted.

1.2 Choices Multiply

Start smaller than a vault. Start with a closet. You own 3 shirts and 4 pairs of pants, and an outfit is one of each. How many outfits are there?

You could list them: the blue shirt with the first pair of pants, the blue shirt with the second, and so on — four outfits for the blue shirt, four for the gray shirt, four for the white one. Three shirts, each paired with all four pairs of pants, gives $3 \times 4 = 12$ outfits. Notice what did *not* happen: the choices did not add. There are not $3 + 4 = 7$ outfits. Each shirt is a complete set of four pant-choices, so the pants get counted once *per shirt* — three groups of four. Add a third stage, say 2 jackets, and every one of the 12 shirt-and-pant combinations pairs with both jackets: $12 \times 2 = 24$. Each new stage multiplies the running total by its number of options. That is the whole rule, and it is worth stating properly:

Definition 1.1 (The Multiplication Principle). If a choice is made in stages, one after another, and the first stage has some number of options, the second stage has some number of options, and so on — and what you pick at one stage does not change what is *available* at the next — then the total number of outcomes is the numbers of options **multiplied together**. Write the option-counts side by side and multiply.

Now the vault’s little cousin: the 4-digit bank card PIN. A PIN is four stages — first digit, second digit, third, fourth — and each stage has 10 options, the digits 0 through 9. By the multiplication principle there are $10 \times 10 \times 10 \times 10 = 10,000$ possible PINs. Four tens multiplied together comes up so often it gets a shorthand: 10^4 , read “ten to the fourth,” means four tens multiplied. (In that notation, the little number up top just counts how many copies of the base are being multiplied: 10^4 is $10 \times 10 \times 10 \times 10$.)

Is 10,000 a lot? Compare it to the 6-digit version some banks now use: $10^6 = 10 \times 10 \times 10 \times 10 \times 10 \times 10 = 1,000,000$. Going from four digits to six *multiplies* the count by $10^6/10^4 = 100$ — a hundred times as many codes for a thief to try. Two extra digits did not add two units of safety; they multiplied the space by a hundred. That is the first thing to feel about counting: *more stages multiply, and multiplying is strong*.

Example 1.1 (License plates). A state issues plates with three letters followed by three digits (ABC 123). Letters give 26 options per stage, digits give 10, so the multiplication principle gives

$$26^3 \times 10^3 = 17,576 \times 1,000 = 17,576,000$$

possible plates — about seventeen and a half million. When a state runs out, it does not invent longer plates by adding one more *option*; it adds one more *character*, one more stage, and the whole space multiplies by 26 or 10 again.

1.3 The Doubling Ladder

One kind of stage matters more to this book than any other: the stage with exactly two options. A coin flip. Heads or tails, nothing else. Count the outcomes as flips are added:

- **One flip:** heads, tails — 2 outcomes.
- **Two flips:** HH, HT, TH, TT — 4 outcomes. Each of the two first-flips pairs with both second-flips: 2×2 .
- **Three flips:** each of the 4 two-flip sequences can be extended by a head or a tail — $4 \times 2 = 8$.

Every extra flip *doubles* the count. That is just the multiplication principle with 2 options at every stage: n flips give $2 \times 2 \times \cdots \times 2$, with n twos, which we write 2^n and read “two to the n .” Here is the ladder, climbed twenty rungs — every entry computed, none estimated:

n	2^n	n	2^n
1	2	11	2,048
2	4	12	4,096
3	8	13	8,192
4	16	14	16,384
5	32	15	32,768
6	64	16	65,536
7	128	17	131,072
8	256	18	262,144
9	512	19	524,288
10	1,024	20	1,048,576

Two rungs are worth knowing by heart. $2^{10} = 1,024$ — a shade over a thousand, close enough that “about a thousand” is an honest approximation. And $2^{20} = 1,048,576$, about a million with change. Notice the eleventh rung: $2^{11} = 2,048$. The wordlist from Section 1.1 has exactly 2,048 words, which means *one word from the list carries exactly as much choice as eleven coin flips*. Hold that fact; Section 1.6 spends it.

Every ten flips, a thousand-fold

Since 2^{10} is about a thousand, every ten extra flips multiply the count by about a thousand. Ten flips, a thousand. Twenty flips, a million. Forty flips, a shade over a trillion. You run out of *names* for the numbers long before the ladder runs out of rungs. A Bitcoin key lives at 256 flips — what *that* height means is Chapter 7’s question, and this ladder is the way there.

A single two-option stage has a name in computing: a **bit** — one coin flip’s worth of choice, written 0 or 1. Eight bits make a **byte**, and the ladder tells you a byte holds $2^8 = 256$ different values. Nothing in this chapter needs more than that: a bit is one flip, and flips double.

1.4 Order Matters: Arrangements

Back to the closet, then to the bookshelf. You have 3 paperbacks — call them *A*, *B*, and *C* — and you are arranging them left to right on a shelf. How many arrangements are there?

Use the multiplication principle, but watch what happens to the stage sizes. Leftmost slot: 3 choices, any of the books. Middle slot: only 2 choices remain, because one book is already shelved. Last slot: 1 book left. The total is $3 \times 2 \times 1 = 6$ arrangements — and you can check by listing: ABC, ACB, BAC, BCA, CAB, CBA. Six, as promised.

The stages *shrank* as books were used up, because arranging does not allow repeats: once *A* is shelved it cannot be shelved again. Arrangements of all n distinct things therefore come to $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$, a product that gets its own name and its own punctuation:

Definition 1.2 (Factorial). For a whole number n , the product of all whole numbers from n down to 1 is written $n!$ and read “ n **factorial**”:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1.$$

It counts the ways to arrange n distinct things in a row. So $3! = 6$, and twelve distinct books can be shelved in $12! = 479,001,600$ ways — nearly half a billion arrangements of one little shelf.

The exclamation mark

The $n!$ notation is younger than the United States: the French mathematician Christian Kramp introduced it in 1808, in a textbook on arithmetic. It caught on because the product it names comes up constantly — every time order matters and nothing may repeat, a factorial is somewhere nearby.

One refinement, and then the shelf is fully understood. What if you arrange only *some* of the books — say 3 books chosen from 5, in order? The same shrinking-stages argument gives $5 \times 4 \times 3 = 60$: three stages, starting at 5. In general, ordered arrangements of k things drawn from n distinct things come to a *falling product* of k factors starting from n :

$$n \times (n-1) \times \cdots \times (n-k+1) = \frac{n!}{(n-k)!}.$$

The fraction form is worth one slow look: $n!$ arranges *all* n things, and dividing by $(n-k)!$ cancels the arrangements of the $n-k$ things you left behind. You will use this on the card in Section 1.6, where $n = 2,048$ and $k = 12$.

1.5 Order Doesn't Matter: Choosing

Now change the question one more time. Five friends — Ann, Bo, Cy, Di, Ed — and you need a team of 2 for a three-legged race. How many teams are there?

Start with what you know: ordered picks of 2 from 5 are a falling product, $5 \times 4 = 20$. But a team has no order. Ann-then-Bo and Bo-then-Ann are the same team wearing different name tags, and in the count of 20 that team was counted *twice* — once for each of its $2! = 2$ internal arrangements. Every team was counted twice. So divide the duplicates out: $20/2 = 10$ teams. (Check by listing: AB, AC, AD, AE, BC, BD, BE, CD, CE, DE. Ten.)

That is the whole move: **count as if order mattered, then divide by the arrangements of each unordered group**. Stated generally it earns its own name and notation:

Definition 1.3 (Choosing: n choose k). The number of ways to choose k things from n distinct things, when *order does not matter*, is the falling product of k factors from n , divided by the $k!$ arrangements of each chosen group:

$$\binom{n}{k} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k!} = \frac{n!}{k!(n-k)!}.$$

The tall-parenthesis symbol is read “ n **choose** k .” So 2 from 5 is $\binom{5}{2} = 10$, and 2 from 10 is $\binom{10}{2} = 45$.

One special case does heavy lifting later in this book. Choosing *two* from n — counting pairs — is $\binom{n}{2} = \frac{n(n-1)}{2}$: n choices for the first member, $n-1$ for the second, divided by 2 because a pair has no first and second. Keep the formula; nothing more about pairs is needed yet.

1.6 The Twelve Words, Counted

Every tool is now on the table. Back to the card. Three questions from Section 1.1, three counts — each one exact, each one checkable on your own machine at the end of this section.

First question: any words, repeats allowed, order mattering. Twelve stages, each with 2,048 options, nothing used up:

$$\underbrace{2,048 \times 2,048 \times \cdots \times 2,048}_{\text{twelve copies}} = 2,048^{12}.$$

Here the ladder pays its dividend. The wordlist has $2,048 = 2^{11}$ words, so each word is eleven twos multiplied; twelve words is twelve groups of eleven twos, which is $12 \times 11 = 132$ twos all told:

$$2,048^{12} = 2^{132}.$$

The full count: twelve words from 2,048, repeats allowed

$2048^{12} = 2^{132} =$
5,444,517,870,735,015,415,413,993,718,908,291,383,296

That number is forty digits long. Written out with all its digits it does not look like anything — the mind has no shelf for it, and Chapter 8 is devoted to feeling just how big such numbers are. For now the point is simpler: you know *exactly* what it is, and you could recompute it.

Second question: all twelve words different, order mattering. Now the stages shrink: 2,048 options for the first word, 2,047 for the second, down to $2,048 - 11 = 2,037$ for the twelfth. A falling product of twelve factors:

$$2,048 \times 2,047 \times \cdots \times 2,037 = 5,271,537,971,301,488,476,000,309,317,528,177,868,800.$$

Place it next to the full count: forbidding repeats shrank the space by only about three percent. That makes sense — with only twelve draws from a list of 2,048, repeats were rare to begin with. Rules that *forbid* things barely move a count; the next rule moves one enormously.

Third question: all twelve words different, and order does not matter. Choosing 12 from 2,048:

$$\binom{2,048}{12} = \frac{2,048!}{12! 2,036!} = 11,005,261,717,918,037,175,659,349,191,168.$$

Still thirty-two digits — but the ordered count had forty. Dividing out the $12!$ internal arrangements cost eight digits, a factor of $12! = 479,001,600$. Order, it turns out, is worth a great deal.

Which of the three counts a thief must actually face — and what it would cost them to walk through it — is a question about attackers and money, not about counting. Chapter 11 answers it, using exactly these three numbers. And one warning belongs here, before anyone gets clever:

Do not make up the words yourself

This chapter counts phrases; it says nothing about how to *choose* one. The one thing you must never do is invent the twelve words out of your own head — human-picked words are not the count you just computed, not anywhere close. Chapter 9 shows what happens, in the wild, to people who do.

The Idea in Code: Count the card yourself

```
>>> 2048 ** 12          # any words, repeats allowed
5444517870735015415413993718908291383296
>>> import math
>>> math.perm(2048, 12) # all different, order matters
5271537971301488476000309317528177868800
>>> math.comb(2048, 12) # all different, order doesn't matter
11005261717918037175659349191168
>>> math.factorial(12)  # arrangements of 12 distinct words
479001600
```

Every count printed in this section appears in that four-line session. Run it; the card on the table becomes a number you own.

1.7 Looking Ahead

You now have the three tools. Choices multiply (definition 1.1) — stages side by side, totals multiplied. A two-option stage doubles, and n of them give 2^n (Section 1.3). Order mattering gives falling products and factorials (definition 1.2); order not mattering divides the duplicates out and gives “ n choose k ” (definition 1.3). And the card on the table is counted: $2,048^{12}$, a falling product, and a choice — forty digits, forty digits, thirty-two.

Counting answers *how many*. It does not answer *how likely* — and “how likely” is the question the thief actually poses, because they will not try every phrase; they will try a few and hope. Turning counts into chances is Chapter 2’s work, and it uses nothing you do not already have: every probability in this book is one counting answer divided by another.

Key Takeaways

- **Choices multiply, they don’t add.** Three shirts and four pants are $3 \times 4 = 12$ outfits, not 7 — each shirt carries a full set of pant-choices with it. More stages multiply the total, and multiplying is strong: two extra PIN digits multiply the space by 100.
- **The doubling ladder.** Every two-option stage doubles the outcomes, so n coin flips give 2^n results. Two rungs to keep: $2^{10} = 1,024 \approx$ a thousand, $2^{20} \approx$ a million. One word from the 2,048-word list is eleven flips’ worth of choice.
- **Order matters $\Rightarrow$ falling products.** Arranging all n distinct things gives $n!$ outcomes; arranging k of n gives $n \times (n-1) \times \cdots \times (n-k+1) = \frac{n!}{(n-k)!}$.
- **Order doesn’t matter $\Rightarrow$ divide out the duplicates.** “ n choose k ” is the falling product divided by the $k!$ internal arrangements of each chosen group: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
- **The card, counted.** Twelve words from 2,048: $2^{132} = 5,444,517,870,735,015,415,413,993,718,908,291,383,296$ with repeats allowed; about 3% fewer if the words must differ; and $\binom{2048}{12}$, eight digits smaller still, if order doesn’t matter. What those numbers are *worth* is a later chapter’s question — the counts themselves are now yours.
- **Every number here is reproducible.** Four lines of Python in Section 1.6 regenerate every count in the chapter. Check them; don’t take them.

Exercises

Exercise 1.1 (★). *Plates, by the principle.* Compute the number of license plates with three letters followed by three digits, by hand or in Python. You should get exactly 17,576,000. Then: the state adds a fourth *letter* at the end. By what factor does the space grow? You should get exactly 26.

Exercise 1.2 (★). *PIN arithmetic.* How many more 6-digit PINs are there than 4-digit PINs — state it as a factor (“... times as many”). You should get exactly 100.

Exercise 1.3 (★★). *Past a million.* Using the doubling table in Section 1.3 — no calculator — find the smallest number of coin flips whose outcome-count exceeds one million. You should land on exactly 20 flips, at 1,048,576.

Exercise 1.4 (★★). *A shelf and a loan.* You have 5 paperbacks. (a) In how many orders can you shelve all five? (b) In how many ways can you choose 2 of them to lend a friend, if which-gets-shelved-first doesn't matter? You should get 120 and 10.

Exercise 1.5 (★ ★ ★). *Twelve different words.* Suppose the twelve words on the card must all be different, order still mattering. In Python, compute `math.perm(2048, 12)` and confirm it equals

5,271,537,971,301,488,476,
000,309,317,528,177,868,800

Then divide it by `2048 ** 12` and confirm the ratio is about 0.968 — the repeat-free space is only about 3% smaller than the full one.

Exercise 1.6 (★ ★ ★). A friend says: “Choosing is just arranging with the duplicates divided out.” Using the five-friends example of Section 1.5 — 20 ordered picks shrinking to 10 teams — explain precisely *which* duplicates the $k!$ divides out, and why each unordered team is counted exactly $k!$ times in the ordered count.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 1.6. In the ordered count of $5 \times 4 = 20$, a team is a *sequence*: Ann-then-Bo and Bo-then-Ann appear as two different entries. But a team has no sequence — those two entries are one team. How many entries does each team produce? As many as there are ways to arrange its own members, and k members arrange in $k!$ ways; for $k = 2$ that is $2! = 2$ entries per team, so the 20 ordered picks contain each team twice and $20/2 = 10$ is the true count. Nothing else is divided out — the $(n - k)!$ in the factorial form handles the *unpicked* friends, a separate matter: it is what converts “arrange all five” into “arrange just two” before the duplicate-division even begins.

2

Probability

► The Story

A thief has your bank card and one try at the PIN pad before the machine swallows the card. He types four digits. What are his chances?

You already know the ingredients: Chapter 1 counted 10,000 possible PINs, and exactly one of them opens the account. If the thief's guess is just one of those ten thousand possibilities — one ticket in a fair raffle of ten thousand — his chance is *1 in 10,000*. Small. Not zero.

That move — from *how many* to *how likely* — is all this chapter adds, and it is the move that turns Chapter 1's counts into a thief's odds. Probability, in this book, is never a mysterious fog machine: it is one counting answer divided by another. But two honest subtleties live here — when you may multiply chances, and why “at least once” questions are best answered backwards — plus one physical fact that the whole book leans on: a chance is only as good as the fairness of the process behind it.

2.1 From Counting to Chance

Roll a fair die. There are 6 faces, all built alike, so all six outcomes are *equally likely*. What is the chance of an even number? Count the favorable faces — 2, 4, 6, that's 3 — and divide by the total: $3/6 = 1/2$. The chance is one half, said aloud as “1 in 2.”

That is the whole definition, and it is only Chapter 1 wearing a new word:

Definition 2.1 (*Probability, equally likely outcomes*). When every outcome of a process is equally likely, the **probability** of an event is

$$\frac{\text{the number of favorable outcomes}}{\text{the number of possible outcomes}}.$$

It is always a number from 0 (no favorable outcomes — impossible) to 1 (every outcome favorable — certain). Read $1/4$ as “1 in 4,” $1/10,000$ as “1 in ten thousand.”

The equally-likely clause is doing real work, and Section 2.5 will stare at it. For now, the examples it earns are the symmetric ones: a fair coin, a fair die, a well-shuffled deck, a thief with no information guessing one possibility out of N . In every case the probability is a ratio of two counts, and Chapter 1 supplies both.

2.2 One in N

The frame that matters for this entire book is the raffle. N tickets, one winner, one draw: the chance of holding the winner is 1 in N . Every guessing game in Bitcoin is a raffle in this shape:

- One try at a 4-digit PIN: 1 in 10,000.
- One try at a 6-digit PIN: 1 in 1,000,000.
- One try at the card on the table — one specific twelve-word phrase out of Chapter 1’s count: 1 in $2,048^{12}$, which is 1 in 2^{132} .

That last one deserves a slow look. Written as a fraction it has a 1 on top and the forty-digit number on the bottom. There is no everyday name for a chance that small, and decimal notation for it needs thirty-nine zeros after the point — so this book simply refuses to write it any other way than *1 in 2^{132}* . When the numbers get cosmic, “1 in N ” is the honest notation: it keeps the size of N in view instead of hiding it in a fog of zeros.

A probability of exactly 1 in N also has a quieter reading: it is what *no information* looks like. The thief who knows nothing about your PIN faces the full 10,000. Knowledge shrinks raffles — knowing the PIN isn’t 0000 leaves 9,999 — and measuring how much knowledge shrinks them is where Part II is heading.

2.3 Independent Stages Multiply

What is the chance of two heads in two flips? Count: 4 sequences, 1 favorable, so $1/4$. Now notice the shortcut: the chance of heads is $1/2$ per flip, and $1/2 \times 1/2 = 1/4$. The chances multiplied — for exactly the reason the counts multiplied in Chapter 1. Each flip is its own stage, and what happens at one stage does not change what is available at the next. The coin has no memory.

Proposition 2.1 (Multiplying chances). *If a process runs in independent stages — what happens at one stage does not influence the others — then the probability of a full sequence of outcomes is the product of the stage probabilities.*

Independence is the load-bearing word. Two flips of one fair coin: independent. Two cards pulled from one deck *without replacement*: not independent — the first pull changes what is left for the second, exactly like Chapter 1’s bookshelf. When in doubt, go back to counting; the counts never lie.

The proposition turns long shots into very long shots fast. Guessing a 4-digit PIN *and then* a second card’s 4-digit PIN: $1/10,000 \times 1/10,000$, one in 10^8 — 1 in a hundred million. Guessing a full sequence of ten coin flips: $(1/2)^{10}$, which is 1 in 2^{10} , or 1 in 1,024. And guessing all twelve words on the card in one try: 1 in 2^{132} . The exponent arithmetic from Chapter 1 — twelve words of eleven flips each — is doing the lifting; the chances inherit it.

2.4 At Least Once: The Complement

Some questions are natural to ask and horrible to count directly. “At least one head in ten flips” — count the favorable sequences and you are counting sequences with one head, two heads, three heads... a mess of cases. Turn the question around. The *opposite* of “at least one head” is “all tails,” and all-tails is a single sequence out of $2^{10} = 1,024$:

$$\text{chance of at least one head} = 1 - \frac{1}{1,024} = \frac{1,023}{1,024} \approx 0.999.$$

About 999 in 1,000. The backwards count took one line.

Definition 2.2 (The complement rule). The probability that something happens *at least once* is 1 minus the probability that it *never* happens: count the “never” case and subtract from 1.

Here is the rule earning its keep on a question that started this entire subject. Throw a die four times; what is the chance of at least one six? “Never” is four non-sixes, and each throw keeps 5 of its 6 faces, independently: $(5/6)^4 = 625/1,296$. So the answer is

$$1 - \frac{625}{1,296} = \frac{671}{1,296} \approx 0.518$$

— a touch better than half. Now the twin: throw a *pair* of dice twenty-four times; what is the chance of at least one double-six? A double-six is 1 of 36 pair-outcomes, so “never” is $(35/36)^{24}$, and

$$1 - \left(\frac{35}{36}\right)^{24} \approx 0.491$$

— a touch *worse* than half. Two bets that look identical in feel, on opposite sides of an even chance.

The gambler who started it all

Those two bets are the actual question the Chevalier de Méré — a French gambler, not a mathematician — put to Blaise Pascal in 1654. De Méré had made money on the first bet and was losing it on the second, and he wanted to know why two such similar wagers behaved differently. Pascal wrote to Fermat, and their letters worked out the arithmetic of chances: counting outcomes, equally likely cases, the complement. Historians date probability theory to that correspondence. It began, literally, with a gambler asking why his edge had flipped — and the answer was counting.

The Idea in Code: Fractions, not floats

```
>>> from fractions import Fraction as F
>>> 1 - F(1,2)**10      # at least one head in ten flips
1023/1024
>>> 1 - F(5,6)**4      # at least one six in four throws
671/1296
>>> float(1 - F(35,36)**24) # de Méré's second bet
0.49140387613090325
>>> F(1, 2048**12)      # the card, one try
1/5444517870735015415413993718908291383296
```

Python’s Fraction keeps the arithmetic exact: what prints here is what the chapter prints, no rounding fog.

2.5 A Fair Coin Is a Physical Fact

Go back to the equally-likely clause, because everything above stands on it. The die gives each face $1/6$ *because the cube is symmetric*; a shaved die does not. The raffle gives each ticket $1/N$ because the tickets are mixed; a rigged drum does not. Probability is arithmetic *plus* a physical promise that the outcomes are equally likely, and the arithmetic cannot supply the promise.

Hold onto that when the raffle is a Bitcoin key. “1 in 2^{132} ” describes a thief facing twelve words *chosen fairly* — every phrase exactly as likely as every other. Whether real key generation keeps that promise is a physical question about where randomness comes from (Chapter 6), and the answer is sometimes no, expensively (Chapter 9). The count is Chapter 1’s; the chance is this chapter’s; the fairness must be earned, and earning it is most of the rest of the book.

2.6 Looking Ahead

You can now turn any count into a chance: favorable over total (definition 2.1), “1 in N ” for the raffle frame, multiplication for independent stages (proposition 2.1), and the complement for “at least once” (definition 2.2). What you cannot yet answer is the question a patient thief actually asks. The thief does not expect to win in one try; the thief expects to *try* — a thousand times, a million times — and wants to know how the tries accumulate. How many guesses until he should *expect* a hit? Is that expectation a promise? Chapter 3 builds the arithmetic of exactly that.

Key Takeaways

- **Probability is a ratio of counts.** Favorable over possible, when outcomes are equally likely — 0 means impossible, 1 means certain, and “1 in N ” is the honest way to say it.
- **The raffle frame.** One try at one secret out of N possibilities is a chance of 1 in N . One try at a twelve-word phrase is 1 in 2^{132} — a fraction best left spoken, not decimalized.
- **Independent stages multiply.** The chances multiply for the same reason the counts did: stage by stage, no memory. Two 4-digit PINs, 1 in 10^8 ; ten flips, 1 in 1,024.
- **Answer “at least once” backwards.** 1 minus the chance it never happens: at least one head in ten flips is $1,023/1,024$; at least one six in four throws is about 0.518.
- **Equal likelihood is physical, not arithmetic.** The counts assume a fair process. A biased source silently rewrites the chances — which is why where randomness *comes from* gets its own chapter.

Exercises

Exercise 2.1 (★). *One try.* What is the chance of guessing a 6-digit PIN in a single attempt? You should get exactly 1 in 1,000,000.

Exercise 2.2 (★). *Two heads.* What is the chance of two heads in two fair flips? Get it two ways — by counting the four sequences, and by multiplying stage chances — and confirm both give $1/4$.

Exercise 2.3 (★★). *At least one.* What is the chance of at least one head in four flips? Use the complement. You should get $15/16 = 0.9375$.

Exercise 2.4 (★★). *The card’s chance, in code.* In Python: `from fractions import Fraction as F`, then `F(1, 2048**12)`. Confirm the denominator is exactly Chapter 1’s forty-digit count:

5,444,517,870,735,015,415,
413,993,718,908,291,383,296

Exercise 2.5 (★★). *De Méré’s pair*. Compute both of de Méré’s bets yourself: $1 - F(5, 6)^{**4}$ and $1 - F(35, 36)^{**24}$, with `float(...)` around each. You should get about 0.5177 and about 0.4914 — and the first bet is the better one.

Exercise 2.6 (★ ★ ★). A thief somehow learns that your 4-digit PIN is *not* 0000. How many possibilities remain, and what is his one-try chance now? More generally: explain, in a sentence or two, why any solid information about a secret changes the thief’s raffle — and why this makes “how much does he already know?” the question that matters.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 2.6. Ruling out 0000 leaves $10,000 - 1 = 9,999$ possibilities, all still equally likely, so the one-try chance rises from 1 in 10,000 to 1 in 9,999 — barely better, because one fact barely shrank the space. The general point is the definition itself: probability is favorable over total, and information is anything that shrinks the *total*. A little information moves the raffle a little; enough information — the kind Chapter 9 shows leaking out of bad randomness — collapses it. That is why the security question is never just “how big is the space” but “how much of it is he really facing.”

3

Expectation and the Long Run

► The Story

The thief from Chapter 2 is back at the PIN pad, but this time the machine does not swallow the card. This time he can try as often as he likes — a thousand tries, a million, a billion — and he is in no hurry. His question is no longer “what are my chances on one try?” It is the question every patient attacker asks: *how long until I win?*

That question has an arithmetic of its own, and it is the arithmetic that prices every attack in this book. It runs on two ideas. The first is *expected value* — the average outcome, computed before anything happens. The second is the *long run* — the fact that averages, over enough trials, stop wandering and settle. Between them sits one honest surprise that most people get wrong: run exactly enough tries to “expect one win,” and the win is still not certain. It is not even close to certain.

3.1 The Question the Thief Actually Asks

Chapter 2’s raffle answers one-try questions. But nobody robs a raffle by buying one ticket and going home; the serious adversary buys tickets every day. The questions that matter for security are all about *accumulation*:

- If each try has a 1-in- N chance, how many tries until the thief wins *on average*?
- After k tries, what has he bought — a 1% chance, a 50% chance, a certainty?
- If tries cost money, what does a win cost?

These are the questions this chapter answers, and they answer with two tools: the expected value, and the behavior of averages over long runs.

3.2 Expected Value

Roll a die once. What do you expect? Not 3 or 4 — those are just faces. Here is the way to compute what a single roll is *worth on average*. Imagine 6,000 rolls. Fairness means each face shows up about 1,000 times, so the total of all rolls is about

$$1,000 \times 1 + 1,000 \times 2 + 1,000 \times 3 + 1,000 \times 4 + 1,000 \times 5 + 1,000 \times 6 = 21,000,$$

and the average per roll is $21,000/6,000 = 3.5$. No die shows 3.5, yet 3.5 is the average — the value each roll contributes to the long-run total. Factoring out the 1,000s shows the shape: each face weighted by its *share* of the rolls, which is its probability:

$$1 \times \frac{1}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6} + 4 \times \frac{1}{6} + 5 \times \frac{1}{6} + 6 \times \frac{1}{6} = \frac{21}{6} = 3.5.$$

Definition 3.1 (Expected value). The **expected value** of a numerical outcome is each possible value multiplied by its probability, all added together: multiply each value by its chance, then sum. It is the long-run average, computed in advance. It need not be a value that can actually occur — the die expects 3.5 and shows it never.

One property makes expected value the workhorse of this book: **expectations add, always**. The expected total of two dice is $3.5 + 3.5 = 7$; of a hundred dice, 350. No independence required, no fine print — expected values simply add across trials. The thief's arithmetic is one line away.

3.3 The Thief's Arithmetic

Watch the thief work. Each try at the PIN is a tiny lottery ticket worth 1 win with probability $1/10,000$ and 0 wins otherwise, so its expected yield is $1/10,000$ of a win. Expectations add: over k tries, the expected number of wins is

$$k \times \frac{1}{10,000} = \frac{k}{10,000}.$$

A hundred tries: expected wins $100/10,000 = 0.01$. Ten thousand tries: $10,000/10,000 = 1$ — one expected win. That is the entire pricing formula for brute force: **expected wins equal tries times chance-per-try**.

For a thief guessing *distinct* possibilities — never repeating himself, the way any competent key-search works — something even cleaner happens. His k distinct guesses cover k of the N possibilities, so his chance of having won is *exactly* k in N : 100 distinct tries at the PIN is exactly a $100/10,000 = 1/100$ chance, no approximation. Expected value and exact chance coincide here, because at most one guess can be right.

Hold both facts for Part III, where $N = 2^{256}$: an attacker's success chance after k well-chosen guesses is at most k in N — and his *expected* wins are exactly k/N . Pricing an attack means dividing the attacker's budget by this arithmetic.

3.4 About a Million Tries

So how long does the thief wait, on average? Two models, two answers — and the difference matters.

The systematic thief — distinct guesses, no repeats, the way any competent search runs — finds the target somewhere in his search order, and it is equally likely to be found on try 1, try 2, ..., try N . The average of those is about $N/2$: *he expects to be done halfway through the space*, and he is certain to be done by the end of it. For the 4-digit PIN: about 5,000 tries on average, 10,000 at worst. For a 6-digit PIN: half a million on average, a million at worst — at one guess per second, about 5.8 days on average, 11.6 days to be sure. That is the sense in which six digits are a hundred times stronger than four: the wait stretches by a hundred.

The independent-guess thief — each try a fresh 1-in- N shot, repeats allowed — waits exactly N tries on average, and here comes the honest surprise that nearly everyone gets wrong. **Averaging N tries per win does not mean winning within N tries**. Run N independent tries at a 1-in- N chance and the chance of *never* winning is:

$$\left(1 - \frac{1}{N}\right)^N \approx 0.368 \quad \text{when } N \text{ is large,}$$

That 0.368 is the one formula this book imports without proof — its exact name is $1/e$, and deriving it is calculus, not arithmetic; checking it is arithmetic, so check it. So the chance of at least one win is only about 0.632 — roughly 63 in 100, not 100 in 100. (Compute it: `1 - (1 - 1/10**6)**(10**6)` prints 0.6321207427789335.) Ten thousand PIN tries “expect” a win, yet more than a third of the time no win has happened at all. And the symmetric gift to the thief: some of the time the win lands *early*. Expectation is an average, not a schedule.

The Idea in Code: The waiting game, simulated

```
>>> import random
>>> random.seed(0)
>>> def one_experiment():
...     target = random.randrange(10000)    # the PIN
...     tries = 0
...     while True:
...         tries += 1
...         if random.randrange(10000) == target:
...             return tries
...
>>> results = [one_experiment() for _ in range(2000)]
>>> sum(results) / len(results)
10089.9145
>>> max(results), min(results)
(79791, 2)
```

Two thousand simulated thieves, each guessing a random 4-digit PIN: the average wait is 10,089.9 tries — the theory’s ten thousand — while one thief hits at try 2 and another is still guessing at try 79,791. Same game, same odds, wildly different fates. The average is real; the spread is real too.

Keep the two models straight, because they answer different questions with different numbers. *Systematic* search: exact chance k/N after covering k of N possibilities, an average wait of about $N/2$, certainty at N . *Independent* tries: an average wait of exactly N , no certainty at all, and the ≈ 0.368 never-win chance after N tries. The first model governs brute force on one key; the second fits things like a botnet hammering thousands of different wallets, where each wallet is its own raffle. Both appear in Part IV’s case files.

3.5 The Long Run

Why trust expected value at all, given that spread? Because of what averages do over long runs: they settle. Roll a die ten times and the average might be 3.1; keep rolling and the wandering dies down:

The Idea in Code: The settling, on your machine

```
>>> random.seed(7)
>>> total = 0
>>> for i in range(1, 100001):
...     total += random.randrange(6) + 1    # add one die roll
...     if i in (10, 100, 1000, 10000, 100000):
...         print(i, total / i)
...
10 3.1
100 3.34
1000 3.435
10000 3.4714
100000 3.49743
```

The running average creeps toward 3.5 — the expected value from Section 3.2 — and it does not leave again. Mathematicians call this the **law of large numbers**: over enough independent trials, the average of what happened closes in on the average of what could happen. This book treats it the way you just saw it: not as a theorem to admire but as an experiment you can rerun with any seed.

The long run is why the thief’s arithmetic matters in practice. A single attack is luck; a campaign of attacks is a business, and businesses run on averages. When Part III prices brute force against a 256-bit key, the pricing works *because* the attacker cannot argue with the long run: at cosmic N , the expected cost is the whole story, and no lucky early hit is coming to anyone’s rescue. (Why the early-hit caveat stops mattering at cosmic scale is Chapter 8’s job to make you feel.)

3.6 Looking Ahead

Everything so far shares one quiet assumption: the outcomes are equally likely. The fair coin, the fair die, the thief facing the full 10,000 PINs. But secrets chosen by people are not like that. Ask a million humans for a four-digit code and 1234 shows up far more than its fair 1/10,000 share; ask for a password and password beats xj9#q2 every time. When outcomes are not uniform, the good guesses come first — and a thief who guesses in popularity order does far better than k/N .

So Part II needs a way to *measure* how unfair a choice process is: how much surprise does the answer actually contain, and how many guesses does the smartest thief need? Those are the two entropy questions, and they are Chapters 4 and 5.

Key Takeaways

- **Expected value** is each value times its chance, summed — the long-run average computed in advance. The die expects 3.5 and shows it never; expectation is not a prediction of any single trial.
- **Expectations add, always.** That one property turns per-try chances into attack budgets: expected wins = tries $\times$ chance-per-try.
- **Distinct guesses are exact.** Covering k of N possibilities is exactly a k/N chance; the systematic search finishes on average halfway through, about $N/2$ tries, and is certain by try N . Brute force against a key is priced in this currency.
- **Independent guesses: average N tries, and no schedule.** Each try is a fresh 1-in- N raffle: the average wait is exactly N tries, yet the never-win chance after N tries stays ≈ 0.368 , so the win chance is only ≈ 0.632 . Averages are not schedules.
- **The long run settles.** Running averages close in on the expected value — the law of large numbers, watchable on your own machine. Attack economics are long-run economics; that is why the arithmetic, not the anecdote, prices security.

Exercises

Exercise 3.1 (*). *The die's center.* Compute the expected value of one fair die roll from the definition — each face times its chance, summed. You should get exactly 3.5.

Exercise 3.2 (*). *Coverage.* A thief makes 500 *distinct* tries at a 4-digit PIN. What is his exact chance of success? You should get $500/10,000 = 1/20$, a 5% chance.

Exercise 3.3 (**). *The patient machine.* A computer makes one guess per second at a 6-digit PIN, working through the space *without repeating itself*. About how many seconds to cover the whole space — the point of certainty? And about how long does the search take *on average*? You should get about 1,000,000 seconds ≈ 11.6 days for the full space, and about half that — 5.8 days — on average (a day holds 86,400 seconds).

Exercise 3.4 (**). *Rerun the experiment.* Run the PIN simulation from Section 3.4 exactly as printed, seed and all. Confirm the average wait is 10,089.9145 tries, the luckiest thief hit at try 2, and the unluckiest needed 79,791.

Exercise 3.5 (**). *The 63% fact.* In Python, compute $1 - (1 - 1/10**6)**(10**6)$ and confirm 0.6321207427789335. Then explain in one sentence why a million independent tries at a one-in-a-million game do *not* make the win certain.

Exercise 3.6 (* * *). “The expected value is not the value you should expect.” Defend that sentence in a short paragraph, using two exhibits from the chapter: the die that never shows 3.5, and the simulated thieves whose actual waits ranged from 2 to 79,791 around an average of about ten thousand.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 3.5. Each try misses with chance $1 - 1/N$, independent misses multiply (Chapter 2), so N tries all miss with chance $(1 - 1/N)^N \approx 0.368$ — and “at least one win” is the complement, about 0.632. Certainty would need the never-chance to be 0, and independent tries never drive it there; each try faces the same full N again.

Exercise 3.6. The expected value is a property of the *average over many trials*, not of any one trial. The die’s 3.5 never appears on a face, because it lives one level up: in the total of thousands of rolls, where the wandering cancels. The simulated thieves make the same point in attacker’s clothing — the average wait of 10,089.9 describes no one thief; the shortest wait was 2 tries and the longest nearly eighty thousand. Expectation prices the campaign, not the attempt. Confusing the two is how “a 1-in-a-million chance” gets misread as either “impossible” (too pessimistic about the thief who hits at try 2) or “safe to ignore for a million tries” (too optimistic about the same thief).

Part II

Entropy

4

Information as Surprise

► The Story

A child thinks of a whole number between 1 and a million and dares you to find it. You may ask twenty yes-or-no questions. She answers honestly every time. Can you corner the number? You can, and most readers already feel how: halve the space each time. “Is it bigger than five hundred thousand?” — one answer, and a million possibilities become five hundred thousand. Nineteen more halvings and exactly one number remains, because Chapter 1’s ladder says twenty doublings make a million: $2^{20} = 1,048,576$. Twenty yes-or-no answers — twenty *bits* — locate one choice out of a million.

That is the deepest thing a bit ever gets: a bit is an answer that cuts the possibilities in half. This chapter turns that game into a measure — a way to say, in bits, how much surprise a choice contains — and the measure turns out to be the one Bitcoin’s security is written in. The word for it is *entropy*.

4.1 Twenty Questions

The twenty-questions trick works because each answer splits the remaining space in two. One answer: 2 possibilities. Two answers: 4. Twenty answers: over a million. The ladder from Section 1.3 runs the whole game — n answers distinguish 2^n things, no more, no less.

Flip the game around and it becomes a *measure*. If a choice can be cornered by twenty yes-or-no questions and no fewer, the choice carries twenty bits of picking-out. A coin flip’s outcome: 1 question, 1 bit. A die’s outcome: between 2 and 3 questions’ worth — the ladder says $2^2 = 4$ is too few faces and $2^3 = 8$ is more than enough, so a die carries more than 2 and less than 3 bits. The exact figure needs one refinement, arriving in Section 4.3.

Notice what the measure is made of: *the number of halvings the choice is worth*. A choice from a bigger fair space is more bits of surprise, by exactly the ladder’s count.

4.2 The Bit as a Measure of Surprise

Make the measure official for the clean case — every outcome equally likely:

Definition 4.1 (Entropy of a fair space). A choice made uniformly from 2^n possibilities carries exactly n **bits of entropy**. It is the number of fair coin flips the choice is worth — the number of halvings needed to corner it.

The card on the table is now measurable with zero new machinery. Twelve words from 2,048, uniformly chosen: Chapter 1 counted $2,048^{12} = 2^{132}$ phrases, so the phrase carries **132 bits** of entropy. Each word is 11 bits — eleven halvings — and twelve of them multiply the space, so the bits add: $12 \times 11 = 132$. Entropy in bits is just the doubling ladder read as a measure of choice.

One word of vocabulary, for the record. The number of doublings needed to reach a number x — what power of 2 it is — has a name: the *base-2 logarithm* of x . This book almost never needs the name, because the ladder picture (“how many halvings?”) always suffices. Where a machine must compute it, Python’s `math.log2` does the work: `math.log2(2048**12)` prints `132.0`.

4.3 Surprise, Weighted

The fair-space definition covers coins and fair dice. Real choice processes are usually unfair — and unfairness changes the surprise. Consider a loaded four-sided die rigged to show its faces with these chances:

$$1: \frac{1}{2} \quad 2: \frac{1}{4} \quad 3: \frac{1}{8} \quad 4: \frac{1}{8}$$

A roll of 1 carries little surprise — it happens half the time; a 50–50 event is one flip’s worth, 1 bit. A roll of 3 is an 8-to-1 long shot: three halvings, 3 bits. The rule: an outcome with chance p carries the flip-count of $1/p$ bits of surprise — halving the chance adds exactly one bit of surprise. (For chances that are exact powers of $1/2$, like these, the counts are exact: $1/2 \rightarrow 1$ bit, $1/4 \rightarrow 2$ bits, $1/8 \rightarrow 3$ bits.)

So the faces of the loaded die deliver different surprises per roll. What is the surprise of *the die as a process*? Average it — Chapter 3’s expected value, each face’s surprise times its chance:

$$\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \frac{1}{8} \times 3 = 1.75 \text{ bits.}$$

A *fair* four-sided die, every face 2 bits of surprise, averages exactly 2 bits. The loaded die averages less. Unfairness leaks surprise: the rigged die is more predictable, and the entropy says by how much — a quarter of a bit per roll.

And the fair six-sided die from Section 4.1 gets its exact figure at last: each face carries the flip-count of 6 — `math.log2(6)` — about 2.585 bits per roll, between 2 and 3 as promised.

4.4 Shannon’s Measure

Definition 4.2 (Shannon entropy). The **entropy** of a choice process is its *expected surprise*: for each outcome, multiply its chance by the flip-count of 1 over that chance, and sum over all outcomes. For a fair space of 2^n outcomes the result is exactly n bits, matching definition 4.1. Among all processes with the same number of outcomes, the fair one has the highest entropy; every bias lowers it.

The paper that named it

Claude Shannon built this measure in 1948 at Bell Labs, in a paper called *A Mathematical Theory of Communication* — the founding document of information theory, and the paper that put the word *bit* (coined a year earlier by his colleague John Tukey, squeezing “binary digit”) to work. Shannon was measuring how much a message *says*; the same number measures how much a choice *surprises*, because a message and a choice are the same kind of object: one possibility picked out of many.

The Idea in Code: Entropy, computed

```

>>> import math
>>> def surprise(p):      # flip-count of 1/p
...     return math.log2(1 / p)
...
>>> def entropy(outcomes):      # outcomes: list of chances
...     # zero-chance outcomes contribute nothing
...     return sum(p * surprise(p) for p in outcomes if p > 0)
...
>>> entropy([0.5, 0.25, 0.125, 0.125])  # the loaded die
1.75
>>> entropy([0.25, 0.25, 0.25, 0.25])    # the fair one
2.0
>>> entropy([0.9, 0.1])                  # a badly bent coin
0.46899559358928133

```

The bent coin is the warning: a process that picks heads nine times in ten carries less than half a bit per flip — not the full bit a fair coin pays. Unfair processes are cheap to guess, and entropy is the price tag. Nothing in this chapter yet says *who* is guessing or how; that is Chapter 5’s refinement.

4.5 What Entropy Is Not

One clarification before it causes harm, because it is the single most common confusion about the subject. Entropy is a property of the *process that chose*, never of the particular thing chosen. The twelve words `apple apple apple...` and the twelve words your wallet actually generated are, as strings, equally long and equally legal — but if the first came from a human picking a favorite fruit and the second from a fair roll of 2,048-sided dice, the first carries almost no entropy and the second carries 132 bits.

There is no machine that inspects a string and reports its entropy, the way there is no scale that weighs “how chosen” something was. Entropy lives in the *how*. Hold that thought: Chapter 9’s graveyard is full of keys that looked perfectly respectable as strings.

4.6 Looking Ahead

Shannon’s entropy is an average — the expected surprise over the whole process. Averages, as Chapter 3 kept warning, are not schedules, and they are not guarantees either. A process can have decent *average* surprise and still hand the thief a gift: one unusually likely outcome. The thief guesses the most likely value first, every time. So the security question is not “what is the average surprise” but “what does his *best first guess* buy?” That question has its own measure — leaner than Shannon’s, harder-edged, and the one that actually prices your key. Chapter 5 builds it.

Key Takeaways

- **A bit is a halving.** One yes-or-no answer that splits the possibilities in two. Twenty answers corner one thing in a million, because 2^{20} is a million-odd.
- **Entropy counts the halvings.** A uniform choice from 2^n possibilities carries exactly n bits — the card carries 132, eleven per word.
- **Surprise is per-outcome; entropy is the average.** An outcome with chance p surprises by the flip-count of $1/p$; the process's entropy is the expectation of that surprise.
- **Fair is maximal.** Among processes with the same number of outcomes, the uniform one has the most entropy; every bias leaks some away. A 9-to-1 coin pays less than half a bit per flip.
- **Entropy lives in the how, not the what.** No inspection of a finished string can measure it; the measure belongs to the choosing process. Respectable-looking keys can carry almost none.

Exercises

Exercise 4.1 (*). *Corner the number.* A number between 1 and 1,000,000 is chosen uniformly. How many halving questions guarantee you find it? Use Chapter 1's ladder. You should land on 20.

Exercise 4.2 (*). *Bits on the card.* How many bits of entropy does a uniformly chosen twelve-word phrase from the 2,048-word list carry? You should get exactly 132.

Exercise 4.3 (**). *The bent coin.* Compute the entropy of a coin that lands heads 9 times in 10: $\text{entropy}([0.9, 0.1])$ using the chapter's two functions. Confirm 0.46899559358928133 — less than half a bit.

Exercise 4.4 (**). *Two loaded dice.* Die A rolls its four faces with chances $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$; die B with chances $(\frac{1}{2}, \frac{1}{2}, 0, 0)$ — it never shows 3 or 4 at all. Which has more entropy? Compute both: 1.75 bits and exactly 1 bit. Then state in one sentence what the difference measures.

Exercise 4.5 (***). *The string tells you nothing.* Your friend shows you a twelve-word phrase and asks, “does this have 132 bits of entropy?” Explain why the correct answer is *you cannot tell from the string* — what would you need to know instead? Anchor your answer in Section 4.5's distinction between the process and the product.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 4.4. Die B has one bit: it is a coin in die's clothing, two equally likely faces. Die A has 1.75. The difference measures how much less predictable A is *on average* — A 's long shots (3 and 4, at 3 bits of surprise each) pull its average up even though both dice share the same most-likely face. Keep the two numbers in mind; the next chapter argues that for a thief, the shared $\frac{1}{2}$ face matters more than the 0.75-bit gap.

Exercise 4.5. Entropy counts halvings *in the choosing*, and the string does not record its choosing. The same twelve words could come from a fair roll (132 bits) or from a human's favorites (a few bits at most,

since favorite words cluster in a tiny corner of the phrase space). To answer the friend's question you need the process: what device, what generator, what source of physical noise. The string is the product; the entropy is the recipe. This is why the book's failure cases, when they come, are all stories about processes — never about strings that “looked random.”

5

The Attacker's Metric

► The Story

Two safes sit in a showroom. The salesman says the first has “an average combination complexity of 1.75 bits per wheel,” the second “2 bits per wheel.” You are not a safecracker, and the numbers mean nothing to you. So here is the question you should actually ask, the one a thief asks on his first walk past your house: *what is his best single try?*

Chapter 4 gave you Shannon's measure — the average surprise, an honest number with one fatal habit: it averages. A choice process that behaves well *on average* can still offer a thief a jackpot on his very first guess. This chapter builds the measure that does not look away: the one that watches the thief's *best* guess instead of the process's average behavior. It is called the min-entropy, and it is the number your Bitcoin key must actually maximize.

5.1 The First Guess Is Free

Back to Chapter 4's loaded die: faces with chances $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{8}$, and a respectable Shannon entropy of 1.75 bits. You are allowed *one guess* at the next roll. Which face do you call?

Face 1, obviously — it shows half the time. Your one-try success chance is $1/2$: fifty percent, against a fair die's 25%. Two guesses, face 1 then face 2, and you cover $3/4$ of the future. Now compare with the fair die: two guesses there cover only $2/4 = 1/2$. The loaded die is *easier to attack than its average suggests*, because the attacker does not experience the average — he experiences the most likely outcome first, the next-most-likely second, and so on, in popularity order.

That is the attacker's entire strategy, and it is free: **guess the likeliest first**. No mathematics beyond Chapter 2. The question for a secret is therefore not “how surprising is it on average” but “how much does the *best* guess buy?”

5.2 Min-Entropy

Definition 5.1 (Min-entropy). A choice process's **min-entropy** is the flip-count of 1 over the chance of its *most likely* outcome — the surprise of the attacker's best single guess. If the top outcome has chance $p_{\max}$, the min-entropy is the flip-count of $1/p_{\max}$ bits.

Run the definitions against each other. Loaded die: top chance $1/2$, so min-entropy is the flip-count of 2 — exactly 1 bit. Fair four-sided die: top chance $1/4$, min-entropy 2 bits. The uniform twelve-word phrase: top chance 1 in 2^{132} , min-entropy 132 bits. The pattern: *min-entropy never exceeds Shannon entropy, and the*

two are equal exactly when the process is uniform — when no outcome is any likelier than any other, the best guess and the average guess are the same guess.

So the gap between the two entropies measures the attacker’s discount. Loaded die: Shannon 1.75, min 1 — three quarters of a bit handed to him, a 50% first-guess success where “average behavior” advertised more. The slogan: **a secret is only as strong as its most likely value.**

5.3 The RockYou Lesson

This is not a parlor point. In December 2009, a company called RockYou — it made widgets for social networks — was breached, and about 32 million user passwords spilled into the open. They had been stored in readable form, and what they showed the world was the true shape of human choosing. The single most common password, 123456, appeared about 290,000 times — nearly one user in a hundred. The top few thousand choices covered millions of accounts.

Now measure that shape with the attacker’s metric. If a process’s most likely outcome comes up about nine times in a thousand, its min-entropy is the flip-count of $1/0.009 \approx 111$: `math.log2(1/0.009)` prints 6.7958 ... — under 7 bits. Under seven bits, against 132 for the fairly chosen phrase. A thief who tries 123456 against a hundred accounts wins about once; a thief who tries it against a million wins about nine thousand times. That is why trying spilled passwords everywhere they might be re-used — credential-stuffing — is an industry: human-picked secrets have catastrophically low min-entropy, because humans converge on the same favorites.

The attacker sets the order, not you

Defenders count the size of the space; attackers rank the contents by likelihood. A space of millions is no defense when a few favorites cover most of it. Whenever someone measures a secret by “how many possibilities there are,” ask the attacker’s question: how likely is the likeliest one?

The Idea in Code: The two measures, side by side

```
>>> import math
>>> from fractions import Fraction as F
>>> def min_entropy(outcomes):
...     return math.log2(1 / max(outcomes))
...
>>> min_entropy([0.5, 0.25, 0.125, 0.125])    # loaded die
1.0
>>> min_entropy([0.25, 0.25, 0.25, 0.25])    # fair die
2.0
>>> min_entropy([F(1, 2048**12)])             # the fair phrase
132.0
>>> math.log2(1/0.009)    # a favorite shared by ~0.9% of users
6.795859283219775
```

5.4 What the Metric Buys the Defender

Min-entropy is not just an attacker’s ruler; it is the defender’s design target. It tells you exactly what “chosen properly” must mean for a Bitcoin key: the process must make *every* outcome equally likely, so that the most likely value sits at 1 in 2^{256} and not one guess is any cheaper than the average. Uniform is not a

nicety — uniform is what makes min-entropy equal the full bit-length.

And it reframes the whole rest of the book. The defense of a key reduces to two questions: does the *process* have enough min-entropy (are all outcomes nearly equally likely?), and does anything *leak* that would let an attacker rank the outcomes (knowledge shrinking the raffle, as Chapter 2 put it)? Part IV’s failures are all failures of one of those two. Which raises the question this part of the book still owes: what does a fair process actually look like, physically? Where does anyone find 132 honest coin flips?

5.5 Looking Ahead

The attacker has a metric now, and it is unforgiving: the best guess prices the secret. What the metric assumes — as Chapter 2 warned — is that you can actually *get* a process whose outcomes are equally likely. Coins and dice are the ancient answer; computers collect stranger harvests. Chapter 6 goes to the source: where randomness physically comes from, how your operating system gathers it, why “random-looking” software is not random, and how many dice you’d roll to make a key by hand.

Key Takeaways

- **The thief guesses in popularity order.** The best guess comes first; the average never gets a turn. Security against one guess is set by the most likely outcome.
- **Min-entropy** is the flip-count of $1/p_{\max}$ — the surprise of the best single guess. It never exceeds Shannon entropy, and equals it exactly for uniform processes.
- **The Shannon–min gap is the attacker’s discount.** Loaded die: 1.75 bits average, 1 bit against the best guess — a 50% first-try success hiding inside a respectable average.
- **Human choices collapse the metric.** In the 2009 RockYou spill, 123456 covered nearly one user in a hundred — under 7 bits of min-entropy where uniform would give dozens. People pick favorites; favorites get guessed first.
- **Uniform is the design target.** A key process is “chosen properly” exactly when its most likely value sits at 1 in the full space. Everything in Part IV is a story of that target being missed.

Exercises

Exercise 5.1 (★). *Best guess.* For Chapter 4’s loaded die ($\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}$): what is the attacker’s one-try success chance, and what does it become after two guesses? You should get $1/2$ and $3/4$.

Exercise 5.2 (★). *The metric on the card.* What is the min-entropy of a uniformly chosen twelve-word phrase from 2,048 words? You should get exactly 132 bits.

Exercise 5.3 (★★). *Gap analysis.* A process picks A with chance 0.9 and B with chance 0.1. Compute its Shannon entropy and its min-entropy. You should get about 0.469 bits and about 0.152 bits. Which number prices an attacker’s first guess?

Exercise 5.4 (★★). *Popularity order pays.* In the RockYou spill, the top password covered about 0.9% of 32 million accounts. About how many accounts fall to an attacker who tries only that one password against every account? You should get about 290,000.

Exercise 5.5 (★ ★ ★). A vendor tells you their key generator “has 128 bits of entropy.” They mean Shannon entropy. Your threat model is a thief who gets exactly one guess. Explain why the Shannon figure does not answer your question, what additional fact about the generator would, and why “the most likely key occurs with chance $1/2^{128}$ ” is the strongest possible statement of that fact.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 5.5. Shannon entropy averages the surprise over all outcomes, so a generator could pair one relatively likely key with a vast crowd of rare ones and still post a good average — while the one-guess thief walks off with the likely key. The figure that answers the one-guess threat is the min-entropy: the chance of the *most likely* key, which needs a fact about the *maximum* outcome probability, not the mean surprise. Saying the top key occurs with chance $1/2^{128}$ says every key does (none can be likelier than the likeliest), so the process is uniform and both entropies equal 128 — the best a 128-bit space can do. Anything weaker, and the vendor’s number is advertising the average while the thief shops the maximum.

6

Where Randomness Comes From

► The Story

Here is a question that sounds like a riddle. Your laptop is a deterministic machine — the same program run twice does the same thing twice; that is the entire point of a computer. Yet tonight, when you generate a Bitcoin wallet, that machine will produce a number no one in the universe can predict. Where does a perfectly predictable machine find *unpredictability*?

The short answer: it doesn't find it inside the program — it collects it from the world. This chapter follows the supply chain of randomness: the physical sources, the operating system's reservoir, the pretenders (software that stretches randomness without creating any), and the handful of rules that keep you on the right side of the difference. This is where Chapter 2's promise — *equally likely* — finally gets cashed in.

6.1 The Physical Question

Every tool so far assumed a fair process: fair coins, fair dice, a thief facing the full space. Chapters 4 and 5 then priced unfairness. But they left a hole in the argument: what *makes* a process fair? Not the arithmetic — arithmetic only describes the raffle once the tickets are mixed. The mixing is physical.

That reframing matters because it relocates the trust. When you generate a key, you are not trusting mathematics; you are trusting a physical process to be unpredictable and a chain of software to deliver that unpredictability intact. Meet the chain, link by link.

6.2 True Randomness

The oldest fair process is the die. A properly thrown die lands each face with chance $1/6$, not because anyone proved it but because the tumbling is chaotic: tiny differences in the throw — a hair's difference in angle, a whisper of air — grow into different faces. The unpredictability is manufactured by physics, fresh, with every throw.

Dice are not folklore; they are a serious way to make a key by hand. Chapter 4 measured a die at the flip-count of 6, about 2.585 bits per roll. A 256-bit key therefore needs

$$256 / 2.585 \approx 99 \text{ rolls}$$

— and rolls come whole: 99 rolls supply only 255.9 bits, a hair short, so round up — call it a hundred, an afternoon's honest work with pen and paper. (What to do with the hundred numbers once rolled is a

Chapter 11 question; the point here is the supply.) Coins work too, at one bit per flip: 256 flips. Slow, physical, and exactly as fair as your thumb.

Hardware does the same job faster. Modern chips carry circuits that harvest physical noise — thermal jitter, the timing of electrons — and serve it up as random bits. Cloudflare, which terminates a sizable share of the world's web traffic, famously keeps a wall of lava lamps in its San Francisco lobby and films them: the wax never rises the same way twice, and the camera turns that chaos into seed material. The principle across all of these is one: *the world is the only generator that cannot be predicted from its own source code.*

6.3 The Computer's Well

Your operating system keeps a reservoir of exactly this kind of physical unpredictability, gathered continuously: the precise timing of keystrokes and disk reads, network packet arrivals, hardware noise where the chip provides it. Programmers draw from the well through interfaces like Linux's `/dev/urandom` or the `getrandom` call — no lava lamp required.

This design — collect physical jitter from many sources, pool it, serve it to programs — is as old as Bitcoin itself — and the OS version is older still. Satoshi described the original client's version of it on the BitcoinTalk forum in 2010:

Satoshi, on the seed

"The random seed is very thorough. On Windows, it uses all the performance monitor data that measures every bit of disk performance, network card metrics, cpu time, paging etc. since your computer started. Linux has a built-in entropy collector. Adding to that, every time you move your mouse inside the Bitcoin window you're generating entropy, and entropy is captured from the timing of disk ops."

Read the care in that list: many small unpredictable streams, none of them trustworthy alone, pooled so that no single dry source empties the well. Sixteen-plus years later the architecture is unchanged in spirit — your phone is doing the same quiet collecting right now. When the collection fails, or is skipped, or is replaced by something thinner, the failure lands exactly where Chapter 9 begins.

6.4 Pseudorandom Is Not Random

Now the pretender. A **pseudorandom number generator** — PRNG — is a deterministic machine for *stretching*: feed it a short **seed**, and it grinds out an endless stream of numbers that pass every casual test for randomness. No physical noise involved; same seed, same stream, forever.

Stretching is not creating. The entire stream's unpredictability lives in the seed: a PRNG seeded with 32 bits can produce at most 2^{32} different streams — about 4.3 billion. That sounds like plenty until you price it with Chapter 3's arithmetic: at a million streams checked per second, a thief enumerates *every stream the generator can ever produce* in

$$2^{32} / 1,000,000 \approx 4,295 \text{ seconds}$$

— about 72 minutes. The numbers looked random to a human; to an attacker with the seed space, the generator is a vending machine with 4.3 billion buttons and all the time in the world.

Math.random() is not for keys

Every programming environment ships a convenience PRNG — `Math.random()` in the browser, `random.random()` in Python, `rand()` in C. These are built for simulations and games: fast, well-shaped, and *seeded with almost nothing*. They are not sources of entropy and must never touch a key, a nonce (Chapter 10’s subject), or a seed phrase. The cryptographic draw — `crypto.getRandomValues()`, Python’s `secrets` module, the OS well of Section 6.3 — is the only kind allowed near money.

The Idea in Code: A 256-bit draw, the right way

```
>>> import secrets
>>> k = secrets.randbits(256)      # from the OS well
>>> k.bit_length() <= 256         # always true
True
>>> hex(k)                        # your number, your output
0x... (64 hex digits, almost always -- fewer only when the
leading digits happen to be zeros)
```

Run it twice and the outputs differ; run it on any machine and no one can say what will print. That is the difference between `secrets` and `random`: one draws from the world’s noise, the other from a seed a thief could enumerate.

6.5 Inside Bitcoin Core’s Well

Satoshi’s 2010 list described a client that stirred whatever the operating system offered into OpenSSL’s generator. Today’s Bitcoin Core keeps a well of its own — since version 0.18 it no longer relies on OpenSSL’s generator — and its design is written down in a comment at the top of the source file `random.h`. It reads like this chapter, in code.

Core holds a single 256-bit secret *state*: the water already in its well. Whenever the wallet needs key material it calls a function named `GetStrongRandBytes`, and that call stirs fresh inputs into the state before anything comes out:

- **The operating system’s well** — 32 bytes from `getrandom` on Linux, `getentropy` on macOS, `BCryptGenRandom` on Windows. If this source fails, Core does not limp along on the others: the program stops.
- **The chip’s own noise** — the `RDRAND` instruction, on processors that have it.
- **Timing** — high-precision timestamps, one taken before the gathering and one after, so the second also records how long the gathering itself took.
- **Events** — the low bits of the times at which things happen inside the running program.
- **Everything that came before** — the previous state itself.

All of it goes into one hash, SHA-512 — treat it here as a blender whose 64-byte output looks unrelated to its input, however small the change in that input (the blender’s insides are *On Hashing*’s subject). Core then cuts the 64 bytes in two. The first 32 are handed out as the random bytes the wallet asked for; the last

32 become the new state. What you are given and what is kept are different halves of one blended output, so holding a handed-out value tells you nothing about the state that will produce the next one.

Two more habits finish the design. On first use, Core adds 256 bits from the chip’s slower RDSEED instruction where available, plus facts about the machine — clocks, resource usage, system details, the modern form of Satoshi’s performance counters — and then *strengthens*: it hashes and re-hashes for 100 milliseconds. Anyone trying to re-create that startup from guessed inputs must pay those 100 milliseconds of hashing again for every guess. And once a minute Core repeats a smaller stir with 10 milliseconds of strengthening, so the state keeps taking in fresh water even when nobody is drawing from it.

In the Codebase: Bitcoin Core v31.0, `src/random.h`

```
* When mixing in new entropy, H = SHA512(entropy || old_rng_state) is computed, and
* (up to) the first 32 bytes of H are produced as output, while the last 32 bytes
* become the new RNG state.
```

The source comment’s `||` means “followed by”: the fresh inputs and the old state go into the blender together. The same comment block lists every source above and the two strengthening times, and says of the slow sources that they are “designed to make sure the RNG state contains fresh data that is unpredictable to attackers.”

Read this against the third rule in Section 6.6. Core does not brew its own randomness *instead of* the operating system’s well; it pools that well *with* other streams, so a faulty chip instruction, a frozen clock, or a cloned virtual machine has to fail *together with* everything else before the output becomes guessable. That is the pooling property Exercise 6.5 asks about, built into the software that guards the most bitcoin. Chapter 7 follows 32 of these bytes out of the well and into a key.

6.6 The Rules

The chapter compresses to four rules, in order of how much money rides on them:

1. **Use the cryptographic draw only.** Python’s `secrets`, the browser’s `crypto.getRandomValues()`, Linux’s `getrandom` — or a hardware wallet’s certified generator, never a convenience PRNG.
2. **Never seed by hand anything that matters.** A human-picked seed is a human-picked key — Chapter 5 measured what those are worth.
3. **More stages multiply trust, not safety.** Ten weak sources pooled the *wrong* way stay weak; the well’s job is to pool them so no single dry source matters. That pooling is engineering, not improvisation — use the OS’s, don’t brew your own.
4. **When the stakes are highest, add physics you can see.** Dice and a pencil are slow, old, and beyond remote attack. Chapter 12 returns to when that is worth doing.

6.7 Looking Ahead

You now hold the whole supply chain: physics makes the unpredictability, the OS pools it, the cryptographic draw serves it, and a proper key is 256 honest bits of it. Part III finally meets the number itself — what a 256-bit key is, what the space it hides in actually costs to search, and why the arithmetic you now own says the search will never happen.

Key Takeaways

- **Randomness is physical.** Only the world — dice, thermal noise, lava lamps, timing jitter — supplies true unpredictability. Programs alone cannot; determinism is what programs *are*.
- **The OS keeps a well.** Many small physical streams pooled so no single dry source empties it; Satoshi’s 2010 seed list reads the same way. Draw keys only from the cryptographic tap: secrets, `crypto.getRandomValues()`, `getrandom`.
- **Pseudorandom $\neq$ random.** A PRNG stretches a seed; the stream inherits the seed’s entropy and no more. A 32-bit seed falls to full enumeration in about 72 minutes at a million tries a second.
- **A key by hand is about a hundred dice rolls.** At 2.585 bits a roll, an afternoon of honest rolling matches a 256-bit machine draw — slow physics, fully auditable.
- **Never seed by hand what matters.** Human-picked seeds are human-picked secrets, and Chapter 5 already priced those.
- **Bitcoin Core pools, never replaces.** Each strong draw stirs the OS’s well together with chip noise, timestamps, events, and the previous state through SHA-512; half the output is handed out, half kept as the next state, with 100 ms of strengthening at startup and 10 ms every minute.

Exercises

Exercise 6.1 (*). *The hundred rolls.* A fair die supplies about 2.585 bits per roll. How many rolls supply at least 256 bits? You should get 100 rolls at minimum — the quotient is 99.03 ..., rolls come whole, and 99 rolls give only 255.9 bits, so round up.

Exercise 6.2 (*). *The pretender’s price.* A PRNG is seeded with 32 bits. At a million candidate streams checked per second, about how long does enumerating *every* stream take? You should get about 4,295 seconds — roughly 72 minutes.

Exercise 6.3 (**). *Draw your own.* Run Section 6.4’s code box as printed. Confirm `k.bit_length() <= 256` is True, count your hex digits (usually 64), and run it a second time to confirm the draw changes.

Exercise 6.4 (**). *The wrong tool.* Run `import random; random.seed(42); [random.random() for _ in range(3)]` twice, re-seeding each time. Confirm the two streams are identical, digit for digit. In one sentence: why does that reproducibility disqualify random from key generation?

Exercise 6.5 (***). *Satoshi’s list* — performance counters, mouse timing, disk timing, the OS collector — pools *many mediocre sources* rather than trusting one good one. Using Chapter 5’s metric, explain the defensive logic: what does pooling buy when an attacker might know or influence *some* of the streams, and what property must the pool have for that logic to hold?

Exercise 6.6 (***). *The two halves.* Section 6.5’s generator blends its inputs into 64 bytes, hands out the first 32, and keeps the last 32 as its new state. A thief steals one handed-out value. Explain why it does not let him compute the *next* value the generator will hand out. Then suppose a careless redesign handed out all 64 bytes: what would the thief learn, and what would still protect the next draw?

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 6.4. Identical seed, identical stream: the generator’s output is fully determined by 42, so anyone who learns or guesses the seed owns the whole stream — past and future. A key drawn from it is not a draw from the world’s noise but from the seed’s small space, and small spaces get enumerated. (For `random`, the seed space and predictability are documented features, not bugs — it was built for simulations, where reproducibility is the point.)

Exercise 6.5. Pooling is min-entropy insurance. If an attacker knows some streams, then any single stream’s unpredictability might be zero for him — its best guess certain. But a pool that mixes the streams so the *joint* output stays unpredictable whenever *any one* input was unpredictable keeps the attacker’s best guess at the chance of the strongest unknown stream. The property that must hold: the mixing must not let a known or influenced stream cancel the others — the unknown one must still flow through. That is engineering with proofs attached, which is exactly why the rules say to use the OS’s well rather than brewing your own.

Exercise 6.6. The handed-out half and the kept half are different pieces of one SHA-512 output, and a good hash gives no way to get from one half to the other — so the stolen value says nothing about the state, and the next value is computed from that unknown state. Hand out all 64 bytes and the thief holds the new state itself: everything the generator would produce next is predictable to him *until* fresh inputs are stirred in. What still protects the next strong draw is exactly the pooling — each `GetStrongRandBytes` call mixes in fresh OS bytes, timestamps, and events the thief does not know. Keeping half the output secret is a second line of defense on top of that, so that one leak is not a gift.

Part III

The Key

7

A 256-Bit Number

► The Story

Here is a Bitcoin private key, in full: a number. Not yet — first you need to see the size of the space it comes from:

115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,
665,640,564,039,457,584,007,913,129,639,936

That is 2^{256} — seventy-eight digits, the count of every possible 256-bit string. A private key is one number chosen fairly from a space essentially that large. This chapter is about what those two facts — *a number, that space* — conspire to produce: the unguessable vault at the center of digital money. If it feels anticlimactic that the vault door is “just a number,” good. The anticlimax is the design.

7.1 The Number Itself

A Bitcoin private key is a whole number from 1 up to a ceiling called n — a parameter of Bitcoin’s curve, secp256k1, whose precise value is

$n = 115,792,089,237,316,195,423,570,985,008,687,907,852,837,564,$
 $279,074,904,382,605,163,141,518,161,494,337$

That is 78 digits, and it is a hair under 2^{256} : the difference $2^{256} - n$ is a 39-digit number — enormous in absolute terms, invisible relatively, a rounding that shows up only in the 39th digit. For every purpose in this book, *the keyspace* is 2^{256} . (The curve that fixes n , and what keys are *for* arithmetically, is the companion volume’s subject — *The Curve That Powers Bitcoin* opens secp256k1 parameter by parameter. This book’s subject is the choosing of the number, not the curve arithmetic built on it.)

So a private key is an integer in a range. No structure, no format, no checksum at this level — a number. What makes it a *key* is only how it was chosen: uniformly, so that every one of the roughly 2^{256} values is exactly as likely as every other. Chapters 4 and 5 already priced what that means: 256 bits of entropy, and 256 bits of min-entropy. The most likely key is no likelier than any other.

7.2 The Space You Hide In

How big is 2^{256} ? Written out:

115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936.

Seventy-eight digits — a 1 followed by 77 more digits, or about 10^{77} (a 1 with 77 zeros; Chapter 1’s power notation, scaled cosmic). Comparisons are the only way in:

- The observable universe contains about 10^{80} atoms. The keyspace is about a tenth of a percent of *that* — roughly one key for every thousand atoms in everything you can see through any telescope.
- The universe is about 13.8 billion years old, which is about 4.4×10^{17} seconds. There have been fewer seconds since the Big Bang than there are keys in a 2^{59} corner of the space — and 2^{59} is what you get if you throw away 197 of the 256 bits.
- A billion computers each checking a billion keys per second for a billion years check about 3×10^{34} keys. The space has 10^{77} . That search covers a fraction so small there is no word for it: about one part in 3×10^{42} .

These comparisons are not security poetry; they are Chapter 3’s arithmetic with cosmic inputs. The next chapter makes the pricing rigorous, including the energy floor under it. For now, hold the shape of it: *the space is not large the way a nation is large, it is large the way physics is large.*

7.3 Where the Key Lives in Bitcoin

What does the number *do*? One paragraph, because the machinery is the companion volume’s. From your private key, wallet software derives a public key (multiplication on secp256k1 — easy forward, infeasible backward), and from the public key an address (hashes and encodings — *On Hashing*’s territory). The address is what the world sees; the public key surfaces when you spend; the private key never leaves your side, because it *signs*: it produces a proof that the holder of the number authorized a payment, and anyone can check the proof without learning the number.

That is the entire custody model of Bitcoin compressed: **possession of the number is ownership**. No registry, no account, no institution mediating. Which is why the choosing of the number is the whole ballgame — and why this book spends twelve chapters on 256 bits.

7.4 How Bitcoin Core Draws a Key

Section 6.5 left 32 fresh bytes at the lip of Bitcoin Core’s well. Thirty-two bytes are 256 bits: a whole number from 0 up to $2^{256} - 1$. But a private key must be at least 1 and below n , so two kinds of draw are not keys — zero, and anything at or above n . Core’s answer is the bluntest one available: check the number, and if it fails, throw it away and draw again.

In the Codebase: Bitcoin Core v31.0, `src/key.cpp`

```
void CKey::MakeNewKey(bool fCompressedIn) {
    MakeKeyData();
    do {
        GetStrongRandBytes(*keydata);
    } while (!Check(keydata->data()));
```

`GetStrongRandBytes` is Section 6.5’s well. `Check` hands the 32 bytes to the curve library, `lib-secp256k1`, which accepts them only if the number is not zero and is less than n .

How often does the loop run twice? The rejected values are the $2^{256} - n$ numbers at or above n — the 39-digit gap from Section 7.1 — plus zero. Against a space of 2^{256} , that is about 1 draw in 2.7×10^{38} . In practice the loop runs once; the check is there so that “almost always a key” becomes “always a key.”

Why redraw rather than patch? A tempting shortcut is to take a too-big number and subtract n from it. Then each of the smallest $2^{256} - n$ keys could be reached two ways — drawn directly, or drawn as itself plus n — and would be twice as likely as every other key. The favored keys would be a sliver too thin to matter in practice, but redrawing makes the bias exactly zero: every key from 1 to $n - 1$ stays equally likely, which is the uniformity Section 7.1 said turns a number into a key.

One draw per wallet. When you create a wallet, this loop runs once — to make a *seed*, not a spending key. Core feeds the seed to a keyed hash (HMAC-SHA512, keyed with the text `Bitcoin seed`) and cuts the 64-byte result in two: the first 32 bytes become a master private key, the last 32 a *chain code*. From that pair a fixed public recipe, BIP 32, derives as many child keys as the wallet will ever need, along four standard paths — one per address type, each split into a receiving branch and a change branch. The recipe’s arithmetic is curve work and belongs to the companion volume, *The Curve That Powers Bitcoin*. What matters here is the entropy accounting: **randomness enters the wallet exactly once**, 256 bits of it, and every later key is deterministic arithmetic on those bits. The wallet’s backup is its descriptors (or the wallet file), not a word list — Bitcoin Core does not use the seed phrases Chapter 11 opens, and Section 11.6 explains why.

It was not always so:

Versions	Wallet	How keys are made
before 0.13	independent keys	every key its own fresh draw; a backup covered only the keys drawn before it was made
0.13 to 22	legacy HD	one seed; BIP 32 path <code>m/0'/0'/k'</code>
23 onward	descriptor (default)	one seed; four descriptor paths (44h, 49h, 84h, 86h), receive and change

The trend is the book’s argument in miniature: the one random draw got more careful, and everything after it became arithmetic.

7.5 Looking Ahead

So the defense rests on a single sentence: *a fairly chosen 256-bit number cannot be guessed*. Chapter 8 turns that sentence from intuition into an invoice: what a search of this space costs in time, in machines, and in raw energy — and why the one loophole that survives (batch attacks, many targets at once) buys the attacker almost nothing. Then Part IV asks the question that actually empties wallets: what happens when the choosing is *not* fair?

Key Takeaways

- **A private key is an integer** from 1 to just under 2^{256} — the exact ceiling n is a secp256k1 parameter, close enough to 2^{256} that the difference only appears in the 39th digit.
- **The space is physics-large.** About 10^{77} keys — a tenth of a percent of the atoms in the observable universe; a billion computers at a billion keys a second for a billion years cover one part in 10^{43} .
- **Possession is ownership.** The key signs; anyone can verify; no institution mediates. The choosing of the number is therefore the entire security story.
- **Fair choosing is what makes it a key.** Uniform generation gives 256 bits of min-entropy: no value is any likelier than any other, so no guess is any cheaper than the average.
- **Bitcoin Core redraws, never patches.** MakeNewKey throws away zero and anything at or above n — about 1 draw in 2.7×10^{38} — so every key stays equally likely. A wallet’s randomness enters once, as a 256-bit seed; every later key is BIP 32 arithmetic on it.

Exercises

Exercise 7.1 (★). *The ceiling.* In Python, compute 2^{256} and the secp256k1 group order

```
int("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141", 16)
```

and confirm $n < 2^{256}$. What is $2^{256} - n$? You should get a 39-digit number:

432,420,386,565,659,656,852,420,866,394,968,145,599.

Exercise 7.2 (★). *Counting the space.* How many digits does 2^{256} have? Confirm with `len(str(2**256))`: exactly 78.

Exercise 7.3 (★★). *The hopeless fraction.* A billion machines (10^9) each test a billion keys per second (10^9) for a billion years (about 3.2×10^{16} seconds). Compute the total keys tested, and the fraction of the 2^{256} space covered. You should get about 3.2×10^{34} keys and a fraction of about 1 in 3.6×10^{42} .

Exercise 7.4 (★★). *Your own keyspace fact.* Compute how many *seconds old* the universe is (13.8 billion years; a year is 31,557,600 seconds), and find the smallest power of 2 that exceeds it. You should get about 4.35×10^{17} seconds, and $2^{59} \approx 5.8 \times 10^{17}$ — the age of the universe in seconds is smaller than a 59-bit space.

Exercise 7.5 (★★★). A skeptic says: “A 256-bit key is too long to remember and too big to mean anything — surely something smaller would do.” Using this chapter and Chapter 5, write the two-paragraph reply: what property the 256-bit size buys (min-entropy of a uniform draw), and what would concretely go wrong at, say, 32 bits. Your reply should quote at least one number from Chapter 6.

Exercise 7.6 (★★★). *The redraw odds.* Using Exercise 7.1’s numbers, compute in Python the number of 32-byte draws MakeNewKey rejects (zero, plus every value at or above n) and the odds against a rejection:

```
2**256 - n + 1    and    2**256 / (2**256 - n + 1)
```

You should get rejects = 432,420,386,565,659,656,852,420,866,394,968,145,600 and odds of about 1 in 2.678×10^{38} . Then explain in two sentences why Core redraws instead of subtracting n from a too-big draw.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 7.5. The 256-bit size buys a min-entropy of 256 for any fairly chosen key: the attacker’s best single guess faces 1 in 2^{256} , and Chapter 8 will price what that means in energy. Shrink to 32 bits and the space collapses to about 4.3 billion keys — and Chapter 6 already priced that space: full enumeration in about 72 minutes at a million tries a second, before lunch. A 32-bit “key” is not a small key; it is no key. The length is not for your memory — the seed phrase of Chapter 11 handles memory — it is for the thief’s arithmetic: every added bit doubles his bill.

Exercise 7.6. Subtracting n would give each of the smallest $2^{256} - n$ keys two ways to be drawn, so those keys would be twice as likely as the rest — a non-uniform draw. The bias is far too thin to exploit, but redrawing removes it completely at a cost of one extra draw in about 2.7×10^{38} : exact uniformity for free.

8

Guessing Games at Scale

► The Story

Suppose you owned every computer on Earth. All of them — every laptop, every server farm, every phone in every pocket — pointed at one job: guessing a single fairly chosen 256-bit key. How long would you need?

This chapter prices that question honestly, with nothing but Part I’s arithmetic and a few numbers from physics. The answer comes in three currencies: time, machines, and energy. And then it answers the attacker’s last resort — “what if I don’t care *which* key, any funded key will do” — and shows that even that discount changes nothing. This is the chapter where “cannot be guessed” stops being a slogan and becomes an invoice.

8.1 Pricing the Search

Assemble the tools. Chapter 3: a systematic search through N possibilities expects to finish halfway through — an average of $N/2$ tries — and succeeds after k distinct tries with chance exactly k/N . Chapter 7: $N = 2^{256}$. Chapter 6’s computer science: tries cost time and energy. Pricing the search is now substitution:

- **Chance after k tries:** $k / 2^{256}$.
- **Expected tries:** $2^{256}/2 = 2^{255}$.
- **Expected cost:** (expected tries) $\times$ (cost per try), in whatever currency you like — seconds, watt-hours, dollars.

The entire question is what a realistic try rate and cost are. So measure with the fastest relevant machine humanity operates.

8.2 The Cosmic Rung

The mightiest guessing engine ever built is the Bitcoin network itself: in the mid-2020s it grinds through about 10^{21} hashes per second — a 1 with 21 zeros, every second, worldwide. (Chapter 1’s power notation, scaled up: 10^{21} is a 1 followed by 21 zeros.) Lend the attacker the whole thing — every miner on Earth repurposed from mining to key-searching, at 10^{21} tries per second:

$$\text{expected time} = \frac{2^{255}}{10^{21}} \text{ seconds} \approx 1.8 \times 10^{48} \text{ years.}$$

The universe is 1.38×10^{10} years old. The expected search runs about 10^{38} *universe-lifetimes* long. Even the lucky cases don't save him: to get even a 1-in-a-million shot at the key ($k/N = 10^{-6}$ — a minus sign in the exponent just means “one over,” so 10^{-6} is $1/10^6$, one in a million) he needs $k = 2^{256}/10^6$ tries, which at the same rate is still 10^{50} seconds of trying. There is no schedule here to be lucky against.

8.3 The Floor Under All Floors

Maybe the attacker waits for better computers? Physics answers with a floor that no computer can ever go under. Flipping a bit — the smallest possible step of any computation — costs energy, and the thermodynamic minimum (the *Landauer limit*) at room temperature is about 2.9×10^{-21} joules per flip: 0.000 000 000 000 000 000 002 9 joules. That is the floor; every real machine sits orders of magnitude above it.

Now merely *count* to 2^{256} — not test keys, not hash anything, just count — and each step flips bits at that floor:

$$2^{256} \times 2.9 \times 10^{-21} \text{ joules} \approx 3.4 \times 10^{56} \text{ joules.}$$

The Sun's entire output for a year is about 1.2×10^{34} joules. So counting to 2^{256} at the coldest cost physics permits would burn the Sun's whole annual output about 2.8×10^{22} times — twenty-eight thousand billion billion sun-years, to *count*. Real key-testing machinery, per try, sits far above the floor; and the floor itself only drops a little with colder computers. There is no engineering path from here to there. The number is not hard to search; it is *thermodynamically* unsearchable.

8.4 Batch Attacks Buy Almost Nothing

One loophole is worth taking seriously, because it is the attacker's best card. All the pricing so far targeted *one specific* key. What if the thief is indiscriminate — any funded key will do? If a million keys each guard coins, then every guess has a million chances to hit something: the per-try chance rises from 1 in 2^{256} to 10^6 in 2^{256} . That is the batch advantage, and it is real.

Now price it. The per-try chance becomes about 8.6×10^{-72} . The expected tries to some hit: about 5.8×10^{70} (Chapter 3's halfway rule applied to the reduced space; the exact first-hit mean is twice this — the difference never matters here). At the whole-network rate of 10^{21} tries per second, that is about 1.8×10^{42} years for *one* hit against *any* of a million targets. A million targets, and the invoice still reads in universe-lifetimes beyond counting. The batch advantage multiplies the attacker's power by the number of targets — and even a billion targets multiply a number beyond imagining by a factor that remains imaginable. The space absorbs it.

So brute force against *fairly chosen* keys is over: not hard, not impractical — over. Which concentrates the whole security question onto the one assumption doing the work: *fairly chosen*. What if the key was chosen from a smaller or skewed space? Then the attacker searches *that* space, and the arithmetic turns savage. That is Part IV, and it is where every real theft in Bitcoin's history lives.

8.5 Looking Ahead

Part II measured choice; Part III priced the search. The invoice says: a fairly chosen 256-bit key is beyond guessing — by time, by machines, by thermodynamics, even in batches. So the attacker's profession reduces to a hunt for keys that were *never fairly chosen*: the human-picked phrase, the wounded random number generator, the nonce that rolled the same value twice. Part IV opens the case files — the real thefts, the on-chain evidence, and the mempool sharks that still circle low-entropy keys today.

Key Takeaways

- **The invoice.** Chance after k distinct tries: $k/2^{256}$. Expected tries: 2^{255} . Everything else is the cost of a try.
- **Time.** The entire Bitcoin network’s mid-2020s muscle — 10^{21} tries per second — needs about 1.8×10^{48} years on average: some 10^{38} universe-lifetimes.
- **Energy.** Merely counting to 2^{256} at physics’ floor costs 3.4×10^{56} joules — the Sun’s annual output times 2.8×10^{22} . No engineering reaches it; the floor is thermodynamics.
- **Batches don’t dent it.** A million funded targets raise the per-try chance a million-fold — to 8.6×10^{-72} . The space absorbs multipliers.
- **The whole weight rests on “fairly chosen.”** The instant a key comes from a smaller or skewed process, the search price collapses from cosmic to trivial. Part IV is a museum of exactly that collapse.

Exercises

Exercise 8.1 (★). *The invoice line.* An attacker makes 10^{15} distinct tries at a fairly chosen 256-bit key (a quadrillion — far beyond any real budget). What is his exact chance of success? You should get $10^{15}/2^{256} \approx 8.6 \times 10^{-63}$.

Exercise 8.2 (★). *Network muscle.* At 10^{21} tries per second, how many years does the *average* search of a 256-bit space take? Confirm the chapter’s figure: about 1.8×10^{48} years.

Exercise 8.3 (★★). *The sun-years check.* Using the chapter’s constants — 2.9×10^{-21} joules per bit flip, 1.2×10^{34} joules per Sun-year — verify that counting to 2^{256} costs about 2.8×10^{22} Sun-years. What assumption makes this a *floor* rather than an estimate?

Exercise 8.4 (★★). *The batch discount.* There are roughly fifty million funded Bitcoin addresses. Recompute the attacker’s per-try chance and expected tries-to-first-hit against the whole batch. You should get about 4.3×10^{-70} per try and about 1.2×10^{69} expected tries. Does the discount change the verdict?

Exercise 8.5 (★ ★ ★). “Security through a big keyspace is security by obscurity.” Take a position, in two paragraphs, using this chapter’s invoice and Chapter 5’s metric. Distinguish: a secret that is *one of astronomically many, fairly chosen* versus a secret that is *hidden* — and name the exact property (from Chapter 5) that separates the two.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 8.3. The floor comes from the Landauer principle: flipping a bit dissipates at least about 2.9×10^{-21} joules at room temperature, so 2^{256} flips cost at least 3.4×10^{56} joules no matter what the computer is made of — silicon, light, or physics not yet invented. “Floor” means exactly that: the argument assumes the

cheapest physics allows, so no future machine can beat the number. Real machines sit orders of magnitude above it.

Exercise 8.5. Security by obscurity fails because the hiding place is one of few, and leakers exist: the safe under the bed is among the first places a thief looks — low min-entropy in this book’s terms. A fairly chosen 256-bit key is the opposite situation: it is not hidden in a small space someone can ransack, it is one point in a space no search can traverse, with every point equally likely — maximal min-entropy. The obscurity critique is really the Chapter-5 critique: it bites exactly when the choice process is small or skewed. The remedy is not a cleverer hiding place; it is a fairer die.

Part IV

When Randomness Fails

9

Low-Entropy Keys in the Wild

► The Story

Everything up to this chapter has been the good news: a fairly chosen 256-bit key is beyond guessing, priced in sun-years. Now the autopsies. This chapter opens the case files on keys that were stolen — not because the arithmetic failed, but because the *choosing* did. Each case is real, each emptied actual wallets, and each one maps cleanly onto a tool you now own: a space too small to matter, a distribution too skewed to hide in, a generator that quietly repeated itself.

Watch for the pattern as you read: in every case, the thief's job shrank from “search 2^{256} ” to something a laptop does before lunch. The failures are never in the lock. They are always in the key-cutting.

9.1 The Graveyard Shape

Part III established that the only path to a fairly chosen key is brute force, and that brute force is thermodynamically closed. So the working thief never attacks the space. He attacks the *process*: find the keys that were drawn from a corner of it. Three shapes of failure cover nearly every case: the space was small (a human's idea), the distribution was skewed (favorites), or the generator was broken (repeats). The case files in this chapter are filed under those three shapes — the first file spans two of them — and Chapter 5's metric prices every one.

9.2 Brainwallets: The Lyric That Emptied Wallets

A *brainwallet* is a key made from a memorized phrase: take a line of a song, hash it, use the hash as the private key. The appeal is obvious — no hardware, no paper, nothing to lose but your own memory. The arithmetic is merciless.

First, the mechanism, and it is checkable right now:

The Idea in Code: A brainwallet key, made in public

```
>>> import hashlib
>>> hashlib.sha256(b"to be or not to be").hexdigest()
2e9d72a733ae5a6c1c1ebd6802a33dfc92854bfc3439878e5d77dac320995f54
```

That 64-character hex string *could be used as a private key* — and that is exactly the problem. Every bot on Earth computed this same string years ago.

Now price the defense with the book’s tools. How many phrases might a human actually pick? Song lines, quotes, proverbs, jokes, names, passwords from memory — be generous and call it a billion candidates: 10^9 . Chapter 3’s arithmetic: at a million hashes per second (a laptop, no special hardware), the whole candidate space falls in $10^9/10^6 = 1,000$ seconds — about 17 minutes. Chapter 5’s metric: the real space is far worse than a billion uniform candidates, because humans converge on the *same* hundred famous lines — to be or not to be is the 123456 of brainwallets. Min-entropy of a human-picked phrase: not 256 bits, not 30; a handful of bits against a bot that tries the famous ones first.

The historical record matches the arithmetic to the letter. In March 2014 the security researcher who writes as *palkeo* published *How to steal Bitcoins*, describing exactly this: bots that hash every conceivable phrase and watch the resulting addresses. He documented sweeps landing *seconds* after coins arrived — one weak address lost 0.36 BTC two seconds after being funded — nearly ten thousand once-used brainwallet addresses found with a password dictionary, and even a comment left on-chain by one theft bot to another. The address for private key *one* had already seen 4 BTC flow through it. And the phenomenon never died — it industrialized. In 2023, the recovery firm Unciphered publicized its *KeyBleed* research on low-entropy-key sweeps still running at scale; and on today’s network, watchers documented live “scorched-earth fee races” in 2024 and 2025: coins landing on a known-weak address trigger multiple bots to outbid each other’s transactions in the mempool, fighting for the right to steal, within blocks.

A key that a bot is already holding is not a key. It is a tip jar on the sidewalk.

9.3 The Wounded Generator

Second shape: the process had the right space but a broken mixer. In August 2013, the Android operating system’s cryptographic random number generator — the component every Bitcoin wallet app used to make keys and signature nonces — was found to be faulty: on some devices it produced *repeated* or predictable outputs. Bitcoin.org posted a security alert; wallet apps rushed out updates; but coins whose keys had been made or used with the wounded generator were exposed, and thefts followed.

Measure it. A generator that sometimes repeats itself does not draw from 2^{256} equally likely values — it draws from whatever smaller, correlated set its fault allows, and the attacker does not need to know the fault precisely to exploit it: he needs only to *watch for collisions*. Two signatures from one wallet that reuse the same nonce leak the key outright — Chapter 10 does that algebra — and the 2013 failures produced exactly those collisions on-chain, in public, for anyone scanning. The search cost for the thief: not 2^{256} , not a billion — *detection only*. Read the ledger, spot the repeats, compute the key.

Note what did not fail: secp256k1, SHA-256, the peer-to-peer network, the mathematics. What failed was one component’s supply of fresh unpredictability — Chapter 6’s well, run dry on one platform. The vault door held; the key-cutting machine slipped.

9.4 The Mempool Sharks

The cases above might read as history. They are not; they are an ecosystem. Today’s mempool — the public waiting room where transactions queue before confirmation — is patrolled by bots that maintain maps of weak keys: every brainwallet phrase ever published, every key from a documented broken generator, every pattern humans love. The instant coins move to a weak address, a sweep transaction appears; if two bots spot the same prize, they fight by fee — each replacing its own bid with a higher-fee copy until one wins — which is the “scorched-earth fee race” observers logged in 2024–2025.

For the owner of a weak key, the practical sentence is this: *any coin that touches a weak address is gone within blocks* — not eventually, not probably. The bots’ economics are Chapter 3’s campaign arithmetic running in your disfavor: they play millions of raffles at once, and their expected wins are your expected losses.

If a human invented it, a bot has already tried it

There is no clever phrase. No obscure quote, no private joke, no backwards spelling, no keyboard pattern. The candidate lists are public, hashed, and watched around the clock, and they have been for over a decade. The only safe key is one no human chose — Chapter 11 shows how such keys are actually made and kept.

9.5 Looking Ahead

One case file remains, and it is the subtlest: a failure that needs no weak key at all. Bitcoin signatures carry a second secret — a per-signature random number called the nonce — and if that number is ever repeated or leaked, the key behind it falls out of the arithmetic, even when the key itself was chosen perfectly. The 2013 Android wounds exploited exactly this, Sony’s PlayStation 3 signing key died of it, and the fix — removing fresh randomness from signing altogether — is one of the quiet triumphs of modern Bitcoin engineering. Chapter 10 does the algebra, with toy numbers you can check by hand.

Key Takeaways

- **The failures are never in the lock.** Every real key theft traces to the choosing: a small space (human phrases), a skewed distribution (favorites), or a broken generator (repeats) — never to the mathematics of secp256k1.
- **Brainwallets are tip jars.** A generous billion-phrase space falls in about 17 minutes at laptop speed; the famous-phrase corner falls instantly. Bots computed to be or not to be years ago.
- **A wounded generator needs no search.** Android 2013: repeated outputs meant the thief only had to *watch the ledger for collisions*. Detection, not search.
- **The mempool is patrolled.** Weak-address maps are maintained live; coins landing on them are swept within blocks, with bots fee-fighting over the prize. 2014’s warning is 2025’s industry.
- **Only one defense exists.** Keys no human chose, from a generator that works — Part V’s practice.

Exercises

Exercise 9.1 (★). *The seventeen-minute wall.* A botnet checks a million candidate phrases per second. How long to exhaust a generous billion-phrase brainwallet space? You should get 1,000 seconds — about 17 minutes.

Exercise 9.2 (★). *Make the key no one should use.* Reproduce the chapter’s specimen:

```
hashlib.sha256(b"to be or not to be").hexdigest()
```

It should print exactly 2e9d72a7...20995f54. Then change one letter and confirm the whole key changes — but remember Chapter 4’s lesson: the string tells you nothing about the entropy of the *process*.

Exercise 9.3 (★★). *Pricing the skew.* Suppose the single most famous phrase is the choice of 1% of all brainwallet users. Using Chapter 5’s metric: what is the min-entropy of that choice process, at most? You should get the flip-count of 100 — about 6.6 bits. Compare with 132.

Exercise 9.4 (★★). *The collision watcher.* In 2013 the exploit needed no brute force. Explain, in a few sentences, what the attacker scanned the blockchain for, why two signatures sharing a nonce value are a smoking gun (Chapter 10 has the algebra — your answer here needs only the shape), and why this failure mode is *cheaper* than any guessing.

Exercise 9.5 (★ ★ ★). A friend proposes a “strong brainwallet”: a sixteen-word passphrase they invented themselves, interleaved with their birthday and spelled backwards. Write the response, using this chapter’s ecosystem facts (candidate lists, live mempool bots, fee races) and Chapter 5’s metric. Your answer should state clearly what would have to be true for their scheme to survive — and why human invention cannot supply it.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 9.4. The attacker scans for two signatures made by the same key that carry the same nonce value — visible in the signature data itself, no guessing involved. A repeat means the signer ran the same secret number twice, and Chapter 10’s toy algebra shows that two such equations solve for the key with school arithmetic. It is cheaper than guessing because there is no search at all: the ledger itself announces which keys are compromised. The 2013 wounds mattered precisely because a faulty generator manufactures those repeats.

Exercise 9.5. For the scheme to survive, the phrase would have to sit in a space no bot has enumerated, with no favorites the bots try first — in this book’s terms, high min-entropy against the public candidate lists. Human invention cannot supply it: the lists already contain every pattern a person finds clever (birthdays, backwards spellings, substitutions, interleavings — these are generation rules, and rules are enumerable), and the mempool bots keep the lists current. The gap is not cleverness but kind: a fair die produces outcomes no list contains because no list can contain “the physical world’s noise.” That is why the only safe brainwallet is no brainwallet.

10

The Nonce Catastrophe

► The Story

In December 2010, at the 27th Chaos Communication Congress in Berlin, a hacker collective called fail0verflow walked on stage and showed the world Sony’s most secret secret: the private key that signed every PlayStation 3 game. Sony’s cryptography was correctly implemented. The curve was fine, the keys were fine, the signatures checked out. The company had made exactly one mistake, and it fit in a single number: they used the *same* random number to sign everything.

That number is called the nonce, and its catastrophic sensitivity is not a Sony bug or a 2010 bug — it is a property of how ECDSA signatures work, and it has emptied Bitcoin wallets too. This chapter shows you the algebra with numbers small enough to check by hand, and then the elegant fix that modern wallets quietly use: don’t roll the nonce at all.

10.1 One Random Number Per Signature

A Bitcoin signature is not made from the private key alone. Each signature also uses a fresh secret number, chosen at signing time, used once, and thrown away — the **nonce** (“number used once”). Call it k . The signature is a pair of numbers computed from k , the private key d , and the message’s hash z , and the defining equation has this shape:

$$s = \frac{z + r d}{k}$$

— where r is a public part of the signature everyone sees. One honesty note: real signatures run this arithmetic on a clock-face of n values, where “division” means the modular cousin taught in the companion volume; this chapter uses ordinary division, and every step of the trick keeps its shape. The nonce k hides d inside that equation. That hiding is k ’s only job — and it holds only while k is secret and used exactly once.

Two failure modes, then, both fatal. If k ever *leaks*, the equation solves for d in one step. And if k is ever *reused* on two different messages, the two public signatures — sitting on the blockchain for anyone to read — combine to expose both k and d . No brute force, no search: school algebra.

10.2 Same k Twice = Game Over

Here is the whole catastrophe with toy numbers. Use ordinary division in place of the clock-arithmetic (the shape is identical; the companion volume does the real thing parameter by parameter). Let the secret key be $d = 13$, the reused nonce be $k = 7$, and the public $r = 5$. Two signatures on two messages with hashes $z_1 = 61$ and $z_2 = 33$:

$$s_1 = \frac{61 + 5 \times 13}{7} = \frac{126}{7} = 18 \quad s_2 = \frac{33 + 5 \times 13}{7} = \frac{98}{7} = 14$$

The attacker sees $(r, s_1) = (5, 18)$ and $(r, s_2) = (5, 14)$ on-chain. The r values *match* — that is the alarm bell, since r comes from k alone. Now subtract the two equations: the $r d$ terms cancel, and

$$s_1 - s_2 = \frac{z_1 - z_2}{k} \implies k = \frac{z_1 - z_2}{s_1 - s_2} = \frac{61 - 33}{18 - 14} = \frac{28}{4} = 7.$$

The nonce is out. And with k known, either equation gives the key:

$$d = \frac{s_1 k - z_1}{r} = \frac{18 \times 7 - 61}{5} = \frac{65}{5} = 13.$$

Check that with nothing but a pencil: it is exact. The real attack is the same trick wearing modular arithmetic, run against the public ledger by anyone who can spot matching r values.

One reuse, total loss

This is the sharpest edge in all of Bitcoin’s cryptography. A private key can be generated perfectly — full 256 bits of fair entropy — and still fall to a signature-layer failure that has nothing to do with how well the key was chosen. The nonce is a second secret, made fresh per signature, and it tolerates zero mistakes.

10.3 The PlayStation and the Phone

Sony’s mistake was exactly the toy one: the same nonce, every signature. Once fail0verflow collected two signed firmware files, the private signing key came out with the algebra of Section 10.2 — the master key to the console, recovered from public data, presented on stage.

Bitcoin’s version arrived in August 2013, on Android. The platform’s random number generator — the well from Chapter 6 — was faulty, and among its failures was this one: wallet apps signing transactions sometimes drew the *same* k twice. Matching r values began appearing on the public blockchain, and anyone scanning for them could run the algebra and sweep the coins. Bitcoin.org’s alert pushed wallet updates; the coins already exposed were gone. Note the chain of causation: not a broken curve, not a broken hash — a dry well, turning into repeated nonces, turning into open keys.

10.4 The Fix: Deterministic Nonces

Engineers being engineers, the response was not “be more careful.” It was: *remove the dice from the room*. Modern wallets compute the nonce *deterministically* from the private key and the message — a construction standardized as RFC 6979: k is produced by keyed hashing of the message with the key, so the same message always yields the same k (harmless — identical inputs produce an identical signature, which leaks nothing new) and different messages always yield different, unpredictable k ’s (because the key is in the mix). No fresh randomness is consumed at signing time at all.

Read what that accomplishes in this book’s vocabulary: the entire entropy budget of custody moves to one moment — key generation. Chapter 6’s well, drawn on once, properly, and then never needed again. The catastrophic per-signature dependency on live randomness is engineered out. This is why hardware wallets are safe to operate on compromised-by-entropy laptops: the nonce does not come from the laptop.

10.5 Looking Ahead

The case files are closed. Part IV’s lesson, stated once: a Bitcoin key’s security is *entropy at generation, entropy never again* — and history’s thefts all bought their tickets at one of those two moments. Part V turns from autopsy to practice: how seed phrases carry 128 or 256 bits of fair entropy in a form a human can hold (Chapter 11 shows how those sit inside the 132 you counted), what alphabetizing the words costs (Chapter 1’s counts, cashed in by Chapter 11), and the operational checklist that keeps all of this from being theoretical.

Key Takeaways

- **Every signature hides a second secret.** The nonce k must be fresh, secret, and single-use. Leak it or repeat it, and the key behind it falls out of public data.
- **The reuse algebra is school-level.** Two signatures, one nonce: subtract, solve for k , solve for d . The toy version in this chapter is checkable with a pencil; the real one runs on the ledger.
- **Sony 2010, Android 2013.** One static nonce, one wounded generator — same failure shape, total key loss, no brute force anywhere.
- **The fix removes the dice.** Deterministic nonces (RFC 6979) derive k from key and message, so signing consumes no fresh entropy. Custody’s randomness budget collapses to a single moment: generation.

Exercises

Exercise 10.1 (★). *Pencil check.* Using the chapter’s toy values — $d = 13$, $k = 7$, $r = 5$, $z_1 = 61$, $z_2 = 33$ — verify the two signatures $s_1 = 18$ and $s_2 = 14$, then carry out the attacker’s two steps and recover k and d yourself.

Exercise 10.2 (★). *The alarm bell.* Two signatures on the blockchain show the same r value. What does a matching r tell the observer, and why? (Your answer needs one sentence and the fact that r comes from k alone.)

Exercise 10.3 (★★). *A second toy.* Let $d = 9$, $k = 5$, $r = 3$, with message hashes $z_1 = 23$ and $z_2 = 8$. Compute s_1 and s_2 , then recover k and d using Section 10.2’s steps. You should get $s_1 = 10$, $s_2 = 7$, and back $k = 5$, $d = 9$.

Exercise 10.4 (★★). *Why deterministic is safe here.* With RFC 6979, signing the *same* message twice with the same key yields the *same* k — seemingly the reuse this chapter just condemned. Explain why it is not condemned: what would an attacker need two *different* of, for the algebra to work?

Exercise 10.5 (★★★). “The nonce catastrophe shows that Bitcoin signatures are fragile.” Respond, using this chapter and Chapter 6: locate the fragility precisely (what is consumed, when, by whom), explain how RFC 6979 relocates it, and state what property of the *generation moment* everything then rests on.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 10.2. A matching r means a matching k : r comes from the nonce alone, so two signatures showing the same r were signed with the same nonce — the reuse this chapter prices is visible to anyone reading the chain.

Exercise 10.4. The reuse attack needs two *different messages* signed with the same nonce — that is what makes the subtraction eliminate the key term and solve for k . Same message plus same nonce produces the same signature twice: two copies of one equation, and two copies of one equation are still one equation — nothing new to subtract, nothing new learned. RFC 6979 guarantees different messages get different nonces, so the dangerous pair can never be produced.

Exercise 10.5. The fragility is a consumption point: every signature used to drink fresh randomness, live, at signing time — a second secret with zero tolerance for error, drawn from whatever well the signing machine happened to have. Sony’s well was a constant; Android 2013’s was intermittently dry; both failures were fatal not because the key was weak but because the per-signature draw was. RFC 6979 removes the consumption point: the nonce is derived from the key and message by hashing, so signing needs no randomness at all. Everything then rests on the generation moment — 256 fair bits from Chapter 6’s well, once — which is exactly the moment this whole book is about getting right, and exactly why Part V’s practice is disciplined about it.

Part V

Protecting the Key

11

Seed Phrases

► The Story

The card on the table has been with us since Chapter 1. Twelve words, written by hand, guarding everything. You have counted it, priced it, and watched its imitators die in the case files. Now, at last, the machinery: what the twelve words *are*, where they come from, and why their design is one of the quiet masterpieces of Bitcoin’s usability engineering.

The standard is called BIP 39, and its trick is this: a seed phrase is not a password made of words. It is a 128-bit (or 256-bit) random number, *wearing words as a costume* — a number you can hold in your head’s hands, write on paper, read aloud, and type without ever touching a hex digit. This chapter opens the costume, and then prices the one shortcut people take with it that quietly burns most of its value.

11.1 The Card, Revisited

Back to Chapter 1’s counts, now with names attached. The list of 2,048 words is the BIP 39 English wordlist — public, fixed, identical in every wallet on Earth. Twelve words from it, order mattering: $2,048^{12} = 2^{132}$ phrases, as Chapter 1 counted. But the machinery below makes a sharper statement: the phrase carries exactly **128 bits of fresh randomness** plus 4 bits of something else — and $128 + 4 = 132$ is no coincidence. Watch where the 4 comes from.

11.2 Entropy to Words: The Mechanics

The generation recipe, step by step:

1. **Draw the entropy.** 128 fair bits from Chapter 6’s well (16 bytes; for a 24-word phrase, 256 bits — 32 bytes).
2. **Append a checksum.** Hash the entropy bytes with SHA-256 and append the first few bits of the digest — 4 bits for 128-bit entropy, 8 for 256. (The rule: one checksum bit per 32 entropy bits.) Now the bundle holds $128 + 4 = 132$ bits.
3. **Chop into words.** Split the bundle into eleven-bit groups: $132/11 = 12$ groups. Each group is a number from 0 to 2,047 — an index into the wordlist. Read off the words. Done: twelve words.

Run it on the standard’s own test vector — 128 zero bits, used here purely because its every step is reproducible (and because no one would be mad enough to use all-zeros as real entropy):

The Idea in Code: BIP 39, end to end, on the test vector

```
>>> import hashlib
>>> ent = b'\x00' * 16          # 128 zero bits (test vector!)
>>> digest = hashlib.sha256(ent).digest()
>>> digest.hex()
'374708fff7719dd5979ec875d56cd2286f6d3cf7ec317a3b25632aab28ec37bb'
>>> bits = bin(int.from_bytes(ent, 'big'))[2:].zfill(128)
>>> cs = bin(digest[0])[2:].zfill(8)[:4]    # first 4 bits
>>> cs
'0011'
>>> full = bits + cs              # 132 bits
>>> groups = [full[i:i+11] for i in range(0, 132, 11)]
>>> [int(g, 2) for g in groups]      # wordlist indices
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3]
```

Indices 0 through 3 on the wordlist are abandon, ability, able, about — so the phrase is eleven abandons and an about. Check any BIP 39 reference: the all-zeros test vector maps to exactly that phrase.

Read the pipeline once more, slowly, because it is the whole chapter: *the randomness comes first, the words are just a printout*. The words hold no magic; the 128 bits under them are the key, and every wallet on Earth can reconstruct them from the phrase.

11.3 What the Checksum Buys

The 4 appended bits earn their keep the day you mis-copy a word. Because the last four bits are a function of all the rest, a phrase with a typo almost never passes the checksum: with 4 checksum bits, a random error validates about 1 time in $2^4 = 16$ — your wallet can say “*this phrase is wrong*” instead of silently opening an empty wallet at the wrong address. (24-word phrases carry 8 checksum bits: 1 in 256 bad phrases slip through.)

The wordlist helps too, by design: every English BIP 39 word is uniquely determined by its first four letters, so smudged handwriting and half-legible words remain recoverable — aban can only be abandon. Small mercies, engineered on purpose.

11.4 What Alphabetizing Costs You

Now the shortcut that burns the value. Some people, told to back up their twelve words, get clever: “*I’ll write them in alphabetical order — easier to remember, and I’ll know the rule.*” Count what the rule costs, with Chapter 1’s tools.

An attacker who suspects alphabetizing no longer faces the ordered arrangements. He faces only the *unordered sets* — and Chapter 1 counted both:

$$\text{ordered: } 2,048 \times 2,047 \times \cdots \times 2,037 \qquad \text{sorted: } \binom{2,048}{12}$$

The sorted space is smaller by a factor of exactly $12! = 479,001,600$ against the distinct-word count in the display — because every set of twelve distinct words had $12!$ orderings, and alphabetizing throws the ordering away. (Against Section 11.1’s full repeat-allowing space the factor is about 4.95×10^8 — a ~3% corner of phrases have repeats, and it doesn’t change the price.) In bits: $\log_2(12!) \approx 28.8$ bits of entropy, burned by a filing habit. Your “132-bit” phrase now offers about $132 - 28.8 \approx 103$ bits (against Section 11.1’s 128 fresh bits, the burn is about 24.9 — same story, same moral) — and a publicly discussed one, too: this exact shortcut and its price were debated on Bitcoin Stack Exchange in 2022, where the half-billion-fold speedup for 12 words was worked out in the open. (The same question for 24 words divides by $24! \approx 6.2 \times 10^{23}$, about 79 bits — which is why that discussion judged 24 words safe from the trick *for now* and 12 words getting close to practical reach.)

Every rule you add is a rule the attacker gets free

Alphabetizing, birthday substitutions, every-other-word, your own spelling system — each “personal rule” is public knowledge the moment it’s written anywhere, and each one collapses the search space by a Chapter-1 count. The phrase’s defense is that there is *no* rule: words from fair dice, in the order they fell.

11.5 Passphrases: The Optional Thirteenth Word

BIP 39 allows an optional extra secret — the *passphrase* — that combines with the phrase to produce a different wallet entirely. What it buys: a second factor that lives in your head, so a photographed card alone is not enough; and *plausible deniability* — different passphrases open different wallets, so a decoy can hold a small balance while the real one stays invisible. What it costs: the passphrase is a human-borne secret, and Chapter 5’s law applies without mercy — a passphrase you invented is a low-entropy secret strapped to a high-entropy one. If you use one, make it *generated* (dice words, not invention), and understand exactly what is at stake: lose the passphrase and the phrase alone opens an empty wallet. No recovery, no appeal.

11.6 Why Bitcoin Core Doesn’t Use It

Here is an oddity. The twelve words are the most widely used backup format in Bitcoin, and the reference software does not speak them: Section 7.4 showed Bitcoin Core drawing its own 256-bit seed and backing up descriptors, and as of version 31.0 its source contains no BIP 39 code at all. That is not an oversight. Core’s developers have argued about it for years in the project’s issue tracker (issues #16393, #17748, #19151) and its chat channel, and the objections come down to two — the standard’s own text now concedes the first.

The words do not say what they unlock. The phrase carries 128 fair bits and a 4-bit seal, and nothing else. Which address types? Which derivation paths? Nothing in the twelve words says. BIP 39 itself now lists “No versioning scheme” among its known problems — “there was no way to distinguish the address format that should be used” — and adds that this “is now largely mitigated by use of descriptor wallets.” Pieter Wuille, who designed the BIP 32 key tree, put the same gap another way: “there isn’t really a watch-only version of them.” A phrase alone is a key without a label, and Core’s backup is the label, written out.

The words go to the seed, but never back. BIP 39 does not use the 128 bits directly. It feeds the *sentence itself* — the words as typed text — into a deliberately slow hash: PBKDF2 with HMAC-SHA512, 2,048 rounds, with “mnemonic” plus any passphrase mixed in, and a 512-bit seed comes out. Two consequences follow. First, the seed depends on the spelling, not just the bits: the standard says plainly that the “generated seed depends on the wordlist that was used,” so the same 128 bits printed from a different language’s list open a different wallet. Second, the hash runs one way. A wallet holding a seed can never show you twelve words for it, so every existing Core wallet — each born from a 256-bit draw, not from a sentence — has no

phrase at all. Greg Maxwell, then a Core developer, said in the project’s chat in 2017 what a word scheme should do instead: “words are fine, but should be a scheme that can encode versioning and bidirectionally convert.”

The door is not shut. In 2025 Wuille wrote that he did not “see a problem with, or difficulty implementing, support for bip39 in descriptor parsing” — reading a phrase in as one more descriptor key — while another long-time contributor, Sjors Provoost, summed up the history: “Every attempt to do this has been met with concept NACKs or lack of review.”

Notice what none of it touches. No one in the dispute says the 128 bits are weak: a fairly drawn phrase holds exactly the entropy this chapter counted. The quarrel is over what is wrapped around the bits — a missing label, and a one-way door.

11.7 Looking Ahead

The phrase is now fully understood: 128 fair bits, a 4-bit integrity seal, eleven-bits-per-word onto a public list. What remains is everything around the phrase — the practice. Where it is written, what never touches it, how it is tested before it is trusted, and how it outlives you. Chapter 12 closes the book with the operational checklist.

Key Takeaways

- **A seed phrase is a number in a costume.** 128 (or 256) fair bits, drawn first; the words are a printout of the bits, never a human composition.
- **The pipeline is checkable.** Entropy bytes → SHA-256 checksum bits (1 per 32) → eleven-bit groups → wordlist indices. The all-zeros test vector reproduces `abandon×11 + about` by hand.
- **The checksum catches your typos.** A mis-copied 12-word phrase validates by accident about 1 time in 16; the wallet can refuse instead of opening the wrong wallet silently.
- **Personal rules burn entropy.** Alphabetizing a 12-word phrase divides the attacker’s work by exactly 12! — about 28.8 bits gone, a half-billion-fold gift. The defense is that there is no rule.
- **A passphrase is a second secret, not a decoration.** Generated, it adds real protection and deniability; invented, it is Chapter 9 waiting to happen; lost, it is total.
- **Bitcoin Core doesn’t speak BIP 39.** The phrase carries no label for what it unlocks, and its slow hash runs from words to seed but never back — so Core backs up descriptors instead. Neither objection touches the 128 bits.

Exercises

Exercise 11.1 (★). *The 132, accounted.* A 12-word phrase holds 128 entropy bits and 4 checksum bits. Verify the accounting twice: $128 + 4 = 132$ bits, and $132/11 = 12$ words. How many bits and words for the 24-word phrase? You should get $256 + 8 = 264$ and $264/11 = 24$.

Exercise 11.2 (★). *The test vector, by hand.* Run Section 11.2’s code box exactly as printed. Confirm the checksum bits are 0011, the indices are eleven 0s and a 3, and the phrase is `abandon` eleven times followed by `about`.

Exercise 11.3 (★★). *The alphabetizing price, exact.* In Python, run

```
math.factorial(12) and math.log2(math.factorial(12))
```

Confirm 479,001,600 and about 28.84 bits. Then compute the same for 24 words and confirm the “about 79 bits” figure.

Exercise 11.4 (★★). *The checksum’s mercy.* A 12-word phrase has 4 checksum bits. If you mistype one word (so the phrase is effectively random with respect to the checksum), about how often does the bad phrase still validate? You should get about 1 time in 16. What does the wallet do with the other 15?

Exercise 11.5 (★★★). A friend says: “I’ll pick the twelve words myself, from the official list, in an order only I understand — the list is public, so it’s all above board.” Dismantle this in two paragraphs using every relevant chapter: what Chapter 1 says the space should be, what Chapter 5 says about human favorites, what Chapter 9’s bots are already holding, and what Section 11.4 says personal rules cost.

Exercise 11.6 (★★★). *The one-way door.* Section 11.6’s seed step is one line of Python. Run it on the all-zeros test phrase with BIP 39’s published test passphrase TREZOR:

```
hashlib.pbkdf2_hmac('sha512', phrase.encode(), b'mnemonicTREZOR', 2048).hex()
```

where phrase is abandon eleven times and about, separated by single spaces. You should get a 128-hex-digit seed beginning c55257c360c07c72 and ending 8e7463b04 — the standard’s own test vector. Then explain, in two sentences, why no software can run this step backward to show the words for a seed it already holds.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 11.4. A mistyped phrase is essentially a random 132-bit bundle against the 4-bit seal, so it passes about 1 time in $2^4 = 16$. The other 15 times the checksum *fails* and the wallet says so — which is the mercy: without the seal, every typo would silently open some other, empty wallet, and you would learn about the error only when the coins were nowhere to be found. The checksum converts silent catastrophe into a loud refusal.

Exercise 11.5. The list being public is not the defense — the *fair draw from* the list is. Chapter 1 counted the space at 2^{132} *for uniformly chosen words*; Chapter 5 measured what human favorites do to that figure (the most likely values price the secret); Chapter 9 showed the bots already hold every human pattern’s outputs. And the “order only I understand” is Section 11.4’s trap exactly: any rule he can state collapses the space by a Chapter-1 count, and any rule he can’t state is a rule he can’t reliably reproduce. The phrase’s strength is that there is no rule — words as the dice fell. His plan replaces that with a composition, and compositions are what the bots eat.

Exercise 11.6. PBKDF2 is built from a hash, and a hash gives no route from output back to input; the only way to find a sentence for a given seed is to guess sentences and hash each one, which is the full 2^{132} -sized search of Section 11.1 — and a seed that never came from a sentence has no sentence to find. That is Maxwell’s “bidirectionally convert” in reverse: BIP 39 chose a one-way door, so the words must exist first and the seed after, never the other way round.

12

The Operational Checklist

► The Story

Everything in this book now cashes out into practice. You know what the twelve words are (Chapter 11), what they cost a thief when they are made fairly (Chapter 8), and exactly how they die when they are not (Chapters 9 and 10). What remains is the smallest chapter and the one you will actually use: the checklist. Short, because the principles did the work; absolute, because the case files show what happens to “mostly right.”

12.1 Generation

- Generate on a hardware wallet from a reputable vendor, or with audited open-source software on a machine that is offline. Never on a website. Never in a browser tab.
- Let the device draw from the well (Chapter 6). Do not supply your own “random” anything.
- The paranoid option, worth it for serious amounts: dice. About a hundred rolls for a full 256-bit draw — fifty suffice for a 12-word phrase (Chapter 6) — entered into open-source tooling, *offline*. Physics you can watch.
- Never generate on a clock: no “quick wallet” made in a hurry on someone else’s machine.

12.2 Storage

- Write the twelve (or twenty-four) words on paper — or better, steel — by hand, in order, numbered. Checksums only work if the order survives (Chapter 11).
- No photos. No cloud notes. No password manager, no email, no screenshots. Any networked copy of the phrase is a copy an attacker eventually reads.
- Two copies, in two physically separate places, protected from fire and water. One flood should not be able to take your coins.
- Never type the phrase into any device except the wallet itself during recovery. No “verification” website, no support agent, no form. *Anyone who asks for the words is stealing them.*

12.3 Testing and Use

- Before funding: restore the phrase on a fresh device and confirm the same addresses appear. A phrase you have never restored is a guess, not a backup.
- Fund it small first. Move a little, spend a little, learn the motions while the stakes are trivial.
- If you use a passphrase (Section 11.5): generated, not invented; memorized or stored apart from the phrase; and understood — lose it and the phrase alone opens an empty wallet.
- Recheck the storage once a year. Paper fades, houses flood, memory edits itself.

12.4 Inheritance

Coins that only you can reach are coins that die with you. Decide deliberately who can recover, and how, and test *that* too: a sealed letter with instructions, a lawyer or trusted kin who knows the plan exists, a recovery drill run while you are watching. The checklist for this is short because the principle covers it: *someone you trust should be able to follow your instructions to the coins without you in the room*. If they can't, neither can your estate.

12.5 The Checklist

The whole chapter, pocket-sized:

1. Fair generation: hardware wallet or offline audited software; dice for the serious.
2. Never self-made entropy: no brainwallets, no invented passphrases, no personal rules on the words.
3. Handwritten, in order, on paper or steel — two copies, two locations.
4. Nothing networked ever sees the words: no photos, no cloud, no typing except at recovery.
5. Test the restore before you fund. Fund small first.
6. Passphrase only if generated, stored apart, and its loss understood as total.
7. A recovery plan someone else can execute. Tested.

12.6 Looking Back

The arc of the book, end to end. Counting told you how many; probability, how likely; expectation, how long the thief waits. Entropy measured the surprise; min-entropy measured his best guess; Chapter 6 found the one true source. Then the key: 256 fair bits, priced in sun-years and thermodynamic floors. The case files showed the price of every shortcut. And the phrase — the card on the table — turned out to be the fairest 128 bits your device can draw, wearing twelve words so a human can hold them.

The vaults of Bitcoin are not guarded by institutions, regulations, or obscurity. They are guarded by counting — by numbers so large the universe itself cannot count to them — and by the discipline to let physics, not habit, choose them. You now own both the arithmetic and the discipline. As always in this book: check it, don't take it.

Key Takeaways

- **Generate fair:** hardware wallet, or offline audited software, or a hundred honest dice. Never a website, never your own invention.
- **Store analog:** hand-written, in order, paper or steel, two copies in two places. Nothing networked ever touches the phrase.
- **Test before trusting:** restore on a fresh device, confirm the addresses, fund small.
- **Passphrases are generated or absent.** Invented ones are Chapter 9; lost ones are final.
- **Plan for your absence.** Someone you trust must be able to recover — with instructions tested while you watch.

Exercises

Exercise 12.1 (★). *The audit.* Take this chapter’s seven-point checklist and audit your own (or a hypothetical) setup against it, item by item. Which items pass? Which fail silently today?

Exercise 12.2 (★). *The support call.* A friendly “wallet support” agent emails asking you to “verify” your twelve words after a suspicious-login warning. Which rule does this trip, and what is the correct response?

Exercise 12.3 (★★). *The dice plan.* You decide to generate a 256-bit key by hand for a long-term vault. Using Chapter 6: how many die rolls do you need, about how long will it take at a roll every ten seconds, and why does the process being offline matter? (You should get about 99–100 rolls and about 17 minutes of rolling.)

Exercise 12.4 (★★). *The two-location rule.* Explain why two copies in the *same* house fail this chapter’s standard even though “backups exist.” Which Chapter-2 concept makes the reasoning precise?

Exercise 12.5 (★★★). *Design the inheritance.* Write a one-page recovery plan for a hypothetical holder: what gets stored where, who knows what, what the recovery drill looks like, and how each of this chapter’s seven items is satisfied. Then attack your own plan: find the single point of failure, and fix it.

Solution Sketches

Sketches, not model answers — enough to check your reasoning against. The computational exercises embed their expected outputs in the exercise text.

Exercise 12.2. It trips the storage rule: anyone who asks for the words is stealing them — no legitimate support channel ever needs the phrase, because the phrase *is* the money. The correct response is to not reply, not comply, and report the attempt as the theft attempt it is. The deeper point: possession of the number is ownership (Chapter 7), so handing over the words is not “verification,” it is transfer.

Exercise 12.4. Same house means the two copies share fate: one fire, one flood, one burglary takes both. In Chapter 2’s language, the two copies are not *independent* — their losses are the same event, so their protection does not multiply. Two locations make the losses independent, and independence is what turns “one disaster” into “two simultaneous disasters,” a chance that multiplies toward zero.



Notation, Brick by Brick

Every symbol in this book, in the order it is built, with the place it is built. Nothing here is assumed; everything is constructed on the page listed.

- **Multiplication** (3×4 ; this book always writes $\times$): the ordinary product. Choices multiply (Chapter 1, Definition 1.1).
- **Powers** (10^4 , 2^n): the little number counts how many copies are multiplied. $10^4 = 10 \times 10 \times 10 \times 10$; 2^n is n twos. Built in section 1.2 and section 1.3.
- **The doubling ladder**: the table $2^1 = 2$ through $2^{20} = 1,048,576$ (section 1.3); extended landmarks in Appendix B.
- **Factorial** ($n!$): the product of all whole numbers from n down to 1; counts arrangements of n distinct things. Built in definition 1.2.
- **Falling product** ($n \times (n-1) \times \cdots \times (n-k+1) = \frac{n!}{(n-k)!}$): ordered arrangements of k from n . Built in section 1.4.
- **Choose** ($\binom{n}{k}$, “ n choose k ”): ways to pick k from n when order doesn’t matter; the falling product divided by $k!$. Built in definition 1.3.
- **The under-brace** ($\underbrace{\quad}_{\text{label}}$): a brace under a repeated pattern, the label saying how many — “twelve copies” in section 1.6.
- **Probability as a fraction** ($1/2$, $1/N$): favorable over possible when outcomes are equally likely; said aloud as “1 in 2,” “1 in N .” Built in definition 2.1.
- **Percent** and **approximately** ($\approx$): ordinary share-of-a-hundred, and “close enough that the difference doesn’t change the point.”
- **Scientific notation** (10^{21} ; 10^{-6}): a 1 followed by that many zeros; a minus sign means “one over” — 10^{-6} is $1/10^6$. Glossed where first cosmic (section 8.2).
- **Flip-count** of a number: how many doublings reach it — what power of 2 it is. Named for the record as the *base-2 logarithm*; computed as `math.log2`. Built in section 4.2.

-
- **Bits of entropy:** the flip-count of the space a choice is drawn from (uniform case, definition 4.1); the expected surprise (general case, definition 4.2); the flip-count of 1 over the likeliest outcome (min-entropy, definition 5.1).
 - **The arrow** ($\rightarrow$): “carries to” — $1/2 \rightarrow 1$ bit reads “a one-half chance carries to one bit” (section 4.3).
 - $p_{\max}$: the chance of the most likely outcome (definition 5.1).
 - **The signature cast** (d, k, r, s, z): private key, nonce, public part of the signature, signature value, message hash — ordinary letters for a toy equation, $s = (z + r d)/k$. Used in Chapter 10 only, with ordinary division standing in for the clock-face kind.
 - **The double arrow** ($\Rightarrow$): “therefore the next line follows” (Chapter 10).

B

Tables: Powers of Two and Probability

Landmark powers of two

Continuing Chapter 1's ladder (2^1 through 2^{20}) to the rungs this book uses:

n	2^n
32	4,294,967,296
64	18,446,744,073,709,551,616
128	340,282,366,920,938,463,463,374,607,431,768,211,456
132	5,444,517,870,735,015,415,413,993,718,908,291,383,296
256	115,792,089,237,316,195,423,570,985,008,687,907,853,269,984, 665,640,564,039,457,584,007,913,129,639,936

The card, counted (Chapter 1)

Question	Exact count
12 words of 2,048, repeats allowed	2^{132} (40 digits)
12 distinct words, order mattering	5,271,537,971,301,488,476,000,309,317,528,177,868,800
12 distinct words, order ignored	$\binom{2048}{12} = 11,005,261,717,918,037,175,659,349,191,168$
Alphabetizing price (12 words)	division by $12! = 479,001,600$ (≈ 28.8 bits)

Physical and network constants used in this book

Quantity	Value used
Seconds in a year	31,557,600
Age of the universe	≈ 13.8 billion years $\approx 4.35 \times 10^{17}$ s
Landauer limit per bit flip (300 K)	$\approx 2.9 \times 10^{-21}$ joules
Sun's output per year	$\approx 1.2 \times 10^{34}$ joules
Bitcoin network rate (mid-2020s)	$\approx 10^{21}$ hashes/second
secp256k1 group order n	1157920892373161954235709850086879 07852837564279074904382605163141518161494337

All values are the ones the chapters compute with; every one appears in a runnable specimen or an exercise's printed answer somewhere in the book.



Specimens and References

Every real-world anchor this book leans on, with the chapter where it appears. Items marked • are primary sources.

- **Claude Shannon**, *A Mathematical Theory of Communication* (Bell System Technical Journal, 1948) — entropy and the bit. (Chapter 4)
- **John Tukey** coined “bit” (1947); **Christian Kramp** introduced $n!$ in *Éléments d’arithmétique universelle* (1808); the **Chevalier de Méré**’s dice question to Pascal (1654) began probability theory. (Chapters 1, 2, 4)
- • **Satoshi Nakamoto**, on the Bitcoin client’s random seed, BitcoinTalk forum (2010): performance counters, the OS entropy collector, mouse and disk timing. (Chapter 6)
- **Cloudflare**, “LavaRand in Production: The Nitty-Gritty Technical Details,” The Cloudflare Blog (November 2017) — the wall of lava lamps in the San Francisco lobby, filmed for seed material. (Chapter 6)
- **R. Landauer**, “Irreversibility and Heat Generation in the Computing Process” (1961) — the per-bit energy floor. (Chapter 8)
- **mempool.space** network hashrate observations (2026) — the mid-2020s network grinding around 10^{21} hashes per second (one zettahash). (Chapter 8)
- • **palkeo**, “How to steal Bitcoins” (March 2014), palkeo.com/code/stealing-bitcoin.html (now offline; archived at the Wayback Machine) — brainwallet sweeps seconds after funding; nearly 10,000 once-used phrase addresses; the private-key-1 address; a theft bot’s on-chain comment. (Chapter 9)
- • **Bitcoin.org**, “Android Security Vulnerability” alert (11 August 2013) — the wounded SecureRandom, the affected wallets, the key-rotation response. (Chapters 9, 10)
- • **fail0verflow**, “Console Hacking 2010,” 27th Chaos Communication Congress (December 2010) — Sony’s static nonce and the recovered signing key. (Chapter 10)
- **RFC 6979**, T. Pornin, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)* (August 2013) — the deterministic-nonce fix. (Chapter 10)
- **BIP 39**, M. Palatinus, P. Rusnak, A. Voisine, S. Bowe, *Mnemonic code for generating deterministic keys* (2013) — the wordlist, the checksum, the test vectors. (Chapter 11)

-
- **The RockYou breach** (December 2009) and the Imperva analysis *Consumer Password Worst Practices* (2010) — 32 million passwords, 123456 on top. (Chapter 5)
 - • **Unciphered’s KeyBleed research** (2023) — industrial-scale low-entropy key sweeps. (Chapter 9)
 - • **Bitcoin Stack Exchange #113432**, “How much entropy is lost alphabetising your mnemonics?” (2022) — the half-billion-fold speedup for 12 sorted words; the 24-word margin. (Chapter 11)
 - • **Mempool fee-race observations** (2024–2025), documented by watcher @mononautical and discussed in BitDevs Socratic Seminar notes — bots outbidding each other for coins landing on weak addresses. (Chapter 9)
 - **Companion volumes:** *The Curve That Powers Bitcoin* (secp256k1, key derivation, ECDSA and Schnorr in full) and *On Hashing* (SHA-256, the birthday problem, the avalanche) — the two neighbor subjects this book deliberately touches only at cross-reference depth. (Chapter 7; the curve volume returns in Chapter 10)