

The Architecture of Trust

Security, Cryptography, and the Persistence of Agency

Forrest Landry

UVSM PUBLICATIONS
Wytheville, Virginia

© 2026 Forrest Landry. Some rights reserved.

Published by UVSM Publications LLC
965 Reed Creek Mill Rd
Wytheville, VA 24382-8287
USA

uvsmpublications.com
contact@uvsmpublications.com

ISBN 979-8-90710-233-0

DOI: https://doi.org/10.67291/RLPM5743

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

You may share this work for non-commercial purposes with proper attribution, but you may not adapt, transform, or build upon it.

Full license text: https://creativecommons.org/licenses/by-nc-nd/4.0/

Author ORCID: 0009-0005-6275-0362

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), except as permitted under the license above, without prior written permission of the publisher.

Table of Contents

Introduction ..... **iv**

Part I: Foundations of Security **1**

Chapter 1. Cryptographic Primitives 2

Chapter 2. On Uniqueness and Spoofing 13

Chapter 3. Architecture of a Secure Query System 23

Part II: Infrastructure and Resilience **29**

Chapter 4. Infrastructure Security: Ranking the Threats and Deciding Who Pays 30

Part III: Control, Complexity, and the Limits of Security **37**

Chapter 5. Control Theory, Observability, and the Architecture of Safety 38

Part IV: Technology, Society, and Choice **57**

Chapter 6. Transparency, Consolidation, and the Erosion of Choice 58

Chapter 7. Computing, Cryptography, and the Persistence of Agency 69

Appendix. The Secure Query System in Full **83**

Terms **89**

Bibliography **97**

Index **103**

About the Author **110**

Introduction

What makes the systems I use every day trustworthy? Suppose you pay a bill on a web page. A small lock sits in the address bar, and a message says the money has gone. You cannot see the machines, the wires or the people between you and the company you paid, and you trust them all the same. Now suppose your front door has the strongest lock ever made, and the only key is held by a company that opens the door for you when it chooses. The lock is excellent, and who comes through your door is no longer yours to decide.

This book is about what trust in a complex system rests on, and what security is for. Its answer is that security is measured by what it leaves in the person's hands. It is written for a reader who uses digital systems every day and wants to understand what makes them trustworthy; it assumes no knowledge of cryptography (the craft of hiding messages and of proving who wrote them).

Chapter 1 gives the parts: hash functions (which show any change to a document), symmetric and public keys (which lock and unlock messages), signatures, blockchains (shared records in which tampering shows) and forward security (which keeps old messages closed even if a key is later lost). Chapter 2 names what every one of them depends on: "Uniqueness is the foundation of all security". Chapter 3 shows the parts assembled into one design, in which no single failure exposes what the design protects. Chapter 4 ranks the threats to the electric grid by what an adversary must invest, and asks who should pay to protect it. Chapter 5 asks how far any controller (anything that steers a system toward a goal) can keep a system from harming what lies downstream of it, and finds a limit that no method of control can pass. Chapter 6 shows how consolidation (alignment with a large block, compelled without any choice on the part of the individual), which is not cooperation, closes the paths by which choice enters a system (the ways a person's own decision can still

change what the system does). Chapter 7 shows small, clear systems that keep those paths open, and states where all of this leads: "technology can serve human agency or it can subvert it". Its last words name the purpose: "This is what security is ultimately for".

The book makes five claims, and the last of them rests on the other four. "Uniqueness is the foundation of all security" (Chapter 2). "Each layer of security provides independent protection, so that the system remains secure even if multiple layers are breached" (Chapter 3). "There is no methodology in the controller that can constrain the system such that it does not hurt the living things downstream" (Chapter 5), where the controller is any controller, the system is the system it controls, and the living things downstream are the living things that the effects of that system reach. Think of the people who run a chemical plant as the controller, the plant as the system, and the fish and the families along the river below the plant as the living things downstream. "Technology creates consolidated blocks, but these are not the same as cooperation" (Chapter 6). Think of neighbors who agree to share a well, and of a town in which every house must buy its water from one company. The neighbors cooperate; the town has been consolidated. And Chapter 7 states: "technology can serve human agency or it can subvert it", and then states what decides which: "The difference depends upon whether the systems we build are transparent, inspectable, personally controllable, and resilient against the failures and consolidations that the preceding chapters have described".

Each of those four words is argued in these chapters, so that by the end of Chapter 7 you will have seen the reason for every one of them: transparent and inspectable by Chapters 5 and 7, personally controllable by Chapters 2 and 6, and resilient by Chapters 3 and 4. So security, in this book, is measured by what it leaves in the person's hands. Uniqueness and layering are how a system can be trusted; the limit on control is why a system must stay inspectable and small; and consolidation is what takes control away from the person. So the fifth claim, that security exists to keep human agency alive, rests on the other four. Chapter 4 adds one more support for the fifth claim: a person's agency rests on the systems

everything else depends on, the electric grid first, and the cost of protecting those systems is a public responsibility, because the dependence they protect is the public's.

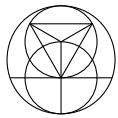
The book ends with what one person can do: run software you can inspect, hold your own keys, and keep at least one channel of communication that does not depend on one company.

To see what any of this leaves in your hands, start with that small lock in the address bar. When your browser shows it, what is actually going on?

PART I

Foundations of Security

What are the parts of security, what do they all rest on, and how are they put together into one design?



Chapter 1

Cryptographic Primitives

> What are the parts every secure system is built from? This chapter gives six foundational building blocks of modern cryptography, each built from first principles and each taking up what the one before leaves open: hash functions, symmetric key cryptography, public key cryptography, digital signatures, blockchain consensus, and forward security.

I. Hash Functions

When your browser shows that small lock, several separate parts are working together, and Section V shows them meeting in the lock. Cryptography is built from a few basic parts, called cryptographic primitives. There are others, but three are the main ones that people need to pay attention to and talk about: the hash function, symmetric key cryptography, and public key cryptography. In this chapter I give those three primitives, and then three parts built from the three primitives: digital signatures, blockchains and forward security.

A hash function (or cryptographic hash function) takes a long string of characters, a document as a linear sequence of characters, and runs it through what is, in effect, a grinding operation. Think of a meat grinder. You take conventional structured material, you put it in, and you end up with a mishmash coming out.

What makes a hash function interesting is how sensitive the mishmash is. A large number of bits (a bit is the smallest unit of digital data, a single 0 or 1) goes in and a smaller number of bits comes out, and the bits that come out are extremely sensitive to any change in the input string. Say I have a one megabyte document, eight million bits worth of data. If even a single one of those bits changes, the hash output changes enormously: a completely different hash output results.

In effect, a hash function is very good at detecting changes in the input document. It also has the property (which is why cryptography is interested in it) that, looking at the hash output, there is no easy way to figure out what the input was. It is called a one-way hash function because the input data cannot be reconstructed from the output hash.

Two documents that are identical generate the same output hash. By comparing the hashes, I can say that the two documents that were the input to the hash function are the same document. I can validate that a document has not changed or that no one has tampered with it. I can compare an earlier hash with a later hash.

Take software, for example: software is a big collection of bits. Say I download a program. I want to make sure that the software its publisher released and the software that I downloaded are the same, that no one has tampered with it or injected a virus en route. The publisher posts the hash of the program it released, and I compute the hash of the program I received. Comparing those two hashes gives me very high confidence that the two programs are in fact identical. It is, for practical purposes, impossible to tamper with the input data and have the hashes stay the same.

A hash collision is somebody defeating this: coming up with two separate documents that produce the same hash output. Finding two documents with the same hash output is very hard to do. When a hash function is broken, someone has found a way to learn something about the input, or a way to make two documents look the same to the hash even though they are different.

II. Symmetric Key Cryptography

A hash shows that a document has changed, and it cannot be run backward to give the document back. So how can I send you a document that you can read and no one else can? Think of a strongbox and two copies of one key. I put a letter in the box, lock it, and hand it to any courier. The courier cannot open the box. You have the other copy of the key, and you can. Symmetric key cryptography is close to what we normally think of as cryptography. You have a key, which is a collection of bits, and you have a document, which is another collection of bits. You combine them in some way (some algorithm, a fixed series of steps, that generates what looks like a random output string), and then that random output string is transmitted. At the other end, if you have the same key, then you can take the output string and the key, and you can effectively reverse the algorithm and end up with the original plain text document.

Plain text (the message before it is locked, also written plaintext) plus key equals ciphertext (the random output string). Ciphertext plus key equals plain text. Symmetric key cryptography can be stated in those two sentences. It is well known: most people understand it as what cryptography is.

III. Public Key Cryptography

The strongbox has one difficulty: you and I must each have a copy of the key before the first letter is sent. How can two people who have never met send each other a secret? Think of an open padlock. I give out as many open padlocks as anyone wants, all of one pattern, and I alone keep the one key that opens them. Anyone can put a letter in a box and snap my padlock shut. After that, not even the sender can open the box. Only I can. Public key cryptography is a form of cryptography where you have two different algorithms, one for encryption (turning plain text into ciphertext) and one for decryption (turning ciphertext back into plain text). The

key is split into two parts, a public part and a private part. The public part and the private part have a specific mathematical relationship with one another.

Somebody can send you a message by taking their plain text, combining it with your public key, and generating a ciphertext. The plaintext cannot be recovered from the ciphertext just using the public key. The only thing that will turn the ciphertext back into plaintext is a process that requires the private key. Someone can encrypt a message using your public key, but nobody can decrypt the message except yourself, who holds the private key.

Public key cryptography gives a secure way for someone to send you a message without your needing to give them a symmetric key, which would mean that any other messages you use with that symmetric key would also be visible to that person. A public key cryptosystem lets people who do not already know one another send encrypted content to one another as long as they know one another's public keys.

IV. Digital Signatures

A locked box says nothing about who locked it, since anyone can use my public key. How can you be sure that a document came from me? Think of a seal ring that only I own, pressed into wax at the foot of a letter. Anyone who knows my seal can look at the wax and say that the letter is mine, and no one without the ring can press that seal. Public key cryptography can be run the other way round to do the same work. In Section III the public key locked and only the private key could open. I can take a plain text and my private key and encrypt the message, and the only way it can be decrypted is using the public key. Encrypting a message with the private key works as a signature algorithm, and makes no secret message. RSA signatures work this way, applied not to the whole document but to a short hash of it (RSA is the public key method of Rivest, Shamir and Adleman, 1978); other signature methods in wide use reach the same result, a

signature anyone can check with the public key and only the holder of the private key can make, by different mathematics. What follows holds of every one of these methods.

Somebody can take the document, which is public, together with its signature, and use my public key to check that the signature matches the document's hash, and therefore verify that I am the person who wrote that document. It works as a digital signature, and for practical purposes it is unforgeable, because no one else has my private key to make a signature that my public key will match to the document.

If the public key is associated with my personal identity, then you know that I have the private key, and therefore I can use the signature as a means for validating that I have made this statement. It is, in effect, an indelible signed document that no one can contest. How do you know that a public key is mine in the first place? Either I handed it to you myself, or someone you already trust has signed a statement tying that key to my name. Section V shows the second way at work in the lock of your browser.

V. Bootstrapping and the Web

So what happens when the small lock appears in your address bar? Public key cryptography is used as a bootstrapping method (a slow method used once, at the start, to set up a faster one), because it is computationally expensive (it takes a great deal of computing). Rather than using it to do internet communications for the bulk of the data, you use the public key method to generate a symmetric key (a different key than the one used for the public-private key pair), and then use that symmetric key for the actual data transmission. Think of the padlock of Section III and the strongbox of Section II: the padlock is slow to work, so the two ends use it once, to settle a strongbox key between them, and then use the strongbox for everything after.

Most web applications work by using public key cryptography once, at the start, to settle a symmetric key, and then using the symmetric key for the rest of the data. There is a handshaking protocol (a protocol is a set of agreed rules for how two machines talk to each other) when a browser signs on to a web page with the HTTPS prefix (HTTPS is the secure form of the web's protocol, shown by the lock in the address bar). The browser and the server talk to one another: they agree on which algorithms to use, the server presents a certificate that ties its public key to its name, and then they negotiate how to create between them a symmetric key that they can then use for all data exchanges subsequent to that.

Who gave the server that certificate, and why does your browser believe it? The certificate is a signed statement. An authority (an organization that browsers trust to issue certificates) signs the statement that ties the server's public key to its name. Your browser arrives with the public keys of those authorities already built in, so it can check the signature in the way Section IV describes. So the lock tells you that an authority your browser trusts has vouched for the name in the address bar, and that what passes between you and that server is locked under a symmetric key that the two of you made.

The hash function, symmetric key cryptography and public key cryptography are the three cryptographic primitives that for the most part people treat as the building blocks of cryptography. There are many different algorithms that implement each of these: a whole range of hashing algorithms, many symmetric key algorithms, and a handful (not too many) of algorithms of the public and private key pair type. So when this book says a hash function or a public key, it names a kind of part, and not any one algorithm.

VI. Blockchain and Consensus

A signature shows who wrote one document. It says nothing about which documents belong to the record that a group shares, or in what order they came. How can many people who do not know one another keep one shared record that none of them can quietly change? Say I take a data block that represents a collection of transactions. I treat that as an input to a hash function. I get a set of hash bits, which I keep around. Think of the transaction block as a table out of a database, each record representing an exchange or an agreement that is now part of the public record. If someone tries to change any of the data in that table, the hash function allows us to detect, using the remembered hash value, whether anybody has tampered with it.

If somebody tries to change it later and you hash it again and do not get the same hash value, then you know someone has tampered with the record. Detecting tampering by hashing the record again is fine and good as long as the remembered hash value is also protected in some way.

A blockchain is this: take the hash value and make it part of the transaction block for the next transaction set. You have an incoming flow of transactions occurring regularly. You take all those, roll them up in the table, calculate the hash, take that hash value and hold on to it for the next period. As the next period's transactions come in, append the hash value to the bottom of the table, then take the hash of that new table (with the hash of the old table in it as a record) to get a new hash value.

Because the hash of the prior table is part of the data content of the next table, the hash value of the next table effectively covers both tables. If I try to change anything in the first table, it changes the hash output value, and the new value disagrees with the value written in the second table. To hide that disagreement, I would have to rewrite the value in the second table, which changes the content of the second table, which changes the hash of the second table. I can keep doing this (table three, table four, and onward), and if people change anything anywhere without rewriting every table after it, the values do not compute anymore.

The weakness is not in the earliest table. It is in the last one. An earlier table is held in place by the tables after it, since the next table carries its hash. The last table has no table after it. Someone who replaces the last table, or who rewrites every table from some point onward, leaves no value anywhere that disagrees. So you need a way of pinning what the head is (the head is the last table, the newest end of the chain). Bitcoin (the first cryptocurrency, launched in 2009) is, at its heart, a consensus-building algorithm as to what is the last block at the head of the chain. A miner (a computer that competes to add the next block) is trying to calculate a function that is very difficult: it requires energy and effort (this is called proof of work). The function is the hash function. The miner takes the new table, adds a number of its own choosing, and computes the hash. The rule asks for a hash value that begins with a long run of zeros. A hash cannot be run backward, so no miner can work out which number will give such a value. Each miner can only try one number after another. Think of a crowd of people rolling handfuls of dice over and over, each hoping to be the first to roll all sixes. You have many miners doing this at the same time. At some point, somebody gets an answer. It is very easy to tell who the first person to get the answer is because the answer is easy to validate even though it is very hard to calculate.

In mining, the difference between finding an answer and checking an answer matters, and the question called P versus NP (the question whether every problem whose answer is easy to check is also easy to solve) asks about the same difference in general. At the difficulty of mid-2026, the miners together perform on average about five hundred thousand billion billion hashes to find one block, while anyone can check the answer with a single hash: the ratio is about five hundred thousand billion billion to one. The miner who provides the solution broadcasts it to the rest, and has the privilege of declaring that block as the top block. It is impossible to predict in advance which miner will have this privilege. There is consensus because the whole pool can easily verify that the answer is correct.

The work of finding a winning answer for each table is what stops the rewriting of earlier tables. Each table on the chain carries its winning answer, and the answer fits only that table's content. Someone who rewrites an earlier table must find a new winning answer for that table and for every table after it, while the whole pool of other miners goes on adding new tables at the head. The pool keeps to the chain that has the most work in it. So a forged chain would have to out-work all the other miners together, and that work is what pins the head.

VII. Forward Security

A chain of hashes protects a record that everyone may read. A secret message is different: it stays secret only as long as its key does. What happens to all my old messages on the day a key gets out? Think of a club whose members lock all their letters under one strongbox key, year after year, while a stranger quietly keeps a copy of every locked box the couriers carry. On the day one member hands the stranger the key, every box the stranger has kept comes open. Now suppose each pair of members cuts a new key for each exchange of letters and melts it afterward. The stranger still holds the boxes, and no key to them exists anywhere. One thing to keep in mind about symmetric key cryptography: a symmetric key can be shared among a local trusted group, but a key shared that way does not have forward security. Forward security is a protection against one of the people in a trusted group defecting and releasing the symmetric key that had been used for all messages in that group.

Now that defecting party (or whoever they release the key to) can go back in history and look at all of the previous messages, because they are all encrypted and decrypted under the same key. A symmetric key that the group keeps using has this problem: if anybody in the group leaks the key, all of the messages are revealed. All secrets come out.

What happens in the better cryptosystems (not as common as it should be, but what distinguishes a really good system from one not so good) is that you have a process that generates a new symmetric key for every connection. The public key cryptosystem is used as a bootstrap for the symmetric key. Every single time you create a connection from any person to any other person, you are creating a temporary symmetric key, and then you throw the key away. So the trusted group no longer keeps one lasting key: each connection between two of its members gets a key of its own.

For this to protect earlier messages, the public key step must itself use fresh key pairs, made for that one connection and thrown away with it (this is called ephemeral Diffie-Hellman key exchange). The handshake of Section V does this today: TLS (Transport Layer Security), the protocol behind web addresses that begin with HTTPS, uses it for every full handshake in its current version, the web standard TLS 1.3. If instead a long-lived private key is used to carry the temporary key, anyone who later obtains that private key can open every recorded session.

Once the messages are exchanged and the work has been done, you do not need the ciphertext anymore, you do not need the plaintext anymore, and you do not need the key anymore. Discard all of them. Even if someone has intercepted the ciphertext, there is no way to decrypt it at any future moment. Whatever was secret stays secret. Forward security means that once the key has been discarded, no one who intercepted the ciphertext can ever decrypt the ciphertext. Because a large quantum computer (a computer built on the physics of very small things, able to solve some problems that ordinary computers cannot) would break today's public key methods (Chapter 2), browsers and servers, Chrome and Cloudflare among the first, now add to the ephemeral exchange a method built to resist one; with both in the handshake, a recording made today stays closed even to a quantum computer.

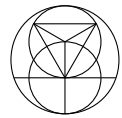
Chapter 7 shows that symmetric keys alone can give forward security too, if the shared secret is replaced by its own hash after every exchange, and that such a protocol can make a fresh key for every message, not only for every connection.

VIII. Summary

In this chapter I have given six parts. A hash function shows any change to a document, and it cannot be run backward. A symmetric key both locks and unlocks. A public key pair lets people who have never met send each other secrets, and, used the other way round, it makes a signature that no one can contest. A chain of hashes makes a shared record tamper-evident, and the work of mining pins its head. Forward security, by making and discarding a fresh key for every connection, keeps whatever was secret secret.

Three of these parts let you check something for yourself, in place of taking someone's word for it: the hash, that a document has not changed; the signature, that a message came from the holder of a key; and the chain of hashes, that a shared record has not been rewritten. The other three keep something closed. From the six parts I draw two answers to the question of what makes a system trustworthy: checks that you can make for yourself, and ways to keep something closed. What the six parts leave in a person's hands is a key: the holder of a private key can open what was locked for that key, and can sign what no one else can sign.

Every one of these parts depends on something more basic. A signature no one can contest, a secret that only its two ends can open, a record no one can rewrite: what does every key, signature and hash rest on?



Chapter 2

On Uniqueness and Spoofing

> What does every key, signature and hash rest on? This chapter answers: uniqueness, and with it identity, and it works through the nature of bearer certificates (anything that grants access to whoever holds it), the event of spoofing (a copy passing for the one true original), the double-spend problem, and the question of data ownership in an era of technological consolidation. In Chapter 1 I used keys, signatures and hashes; in this chapter I ask what every one of those three tools depends on.

I. The Problem of Uniqueness

Think of a vault that opens to one key, and suppose exactly one such key exists in the world. Whoever arrives with that key is the person it was given to. Now suppose a second key is cut that no one can tell from the first. The vault still opens, and it can no longer tell you who has come. All security systems rest, at their deepest level, upon the concept of uniqueness. A key is useful because it is unique. (A public key is meant to be known to everyone, so the key that must be unique is its private partner; a symmetric key shared among a trusted group is unique to that group, since no one outside the group can produce it.) A signature is valid because it is unreproducible. An identity is trustworthy because it cannot be counterfeited. The entire architecture of trust (from the most primitive physical

lock to the most sophisticated cryptographic protocol) depends upon the assumption that certain objects or patterns can exist in one and only one instance. Uniqueness has an exact meaning here, and the chapter's Summary returns to it: an instance is unique, for a given system, exactly when no comparison that system can make finds a second instance that it cannot tell apart from the first. In the digital domain, the place where uniqueness matters most is the private key: the tools of Chapter 1 gather every bearer certificate into that one key (Section IV of this chapter shows how), and its uniqueness rests on its having one holder (Section VII).

The assumption that certain objects or patterns can exist in one and only one instance is not a trivial assumption. The assumption that certain objects or patterns can exist in one and only one instance is a claim about the physics of the world: a claim about how far material reality allows states to exist that cannot be duplicated at all. Physics does allow states that cannot be copied at all: the no-cloning theorem (Wootters and Zurek, and Dieks, 1982) proves that an unknown quantum state (the physical condition of something very small, such as one particle of light) cannot be copied, and Stephen Wiesner's quantum money (1983) builds a banknote on that fact: a counterfeiter cannot copy the quantum states the banknote carries, and so cannot copy the banknote. The security of any system is bounded by the degree to which this assumption holds.

Why is uniqueness so hard to hold in data? Think of a loaf of bread and a song. Give the loaf away, and you no longer have it. Send a friend the song, and you both have it. At the scale of human affairs, the world holds three kinds of thing: atoms, energy and pattern. Pattern always rides on one of the other two: pattern on energy is what we call signal, and pattern on atoms is what we call a molecule. Each of the three has its own character. Atoms flow in closed loops. Energy flows along a gradient, from high to low; its flow is always open, never a loop. Atoms and energy are each conserved: the conservation of matter and of energy holds the amount of each the same, so neither can be had twice over. Pattern is not conserved. A pattern can be copied, destroyed or created at almost no cost beyond the marginal cost (the cost of making one more copy) of the en-

ergy or the atoms it rides on. So pattern is non-rivalrous: many can hold the same pattern at once, and none of them takes it from the others, where atoms and energy are rivalrous: what one holds, another cannot. Data is pattern. The fact that data is pattern is the general reason that data can be copied, and the reason that a key, a signature or an identity held as data is unique only so long as no one copies the key, the signature or the identity.

II. Bearer Certificates and Duplication

The disadvantage of any bearer certificate (anything that grants access to whoever holds it) is that all can potentially be duplicated, whether physical key, token (a small device or code that grants access), or embedded firmware chip (a chip that carries built-in software). The main issue is the degree of difficulty involved with duplication.

A physical key can be photographed and recut. A token can be reverse-engineered. A firmware chip can be read and reproduced. In each case, the security of the system rests not upon absolute impossibility of duplication but upon the practical difficulty of achieving it. The question is always one of degree: how much effort, how much resource, how much specialized knowledge must an adversary invest to produce a convincing duplicate? The strongest physical keys are built on this question: a physically unclonable function in a chip takes its key from the variation that manufacturing leaves in that one chip, a fingerprint that even its maker cannot reproduce. The variation has been measured: in chips of one design, all programmed alike, the delays of the same circuits differed from chip to chip by five thousand to thirty thousand parts per million, while repeated measurements of one chip differed by about thirty parts per million (Gassend, Clarke, van Dijk and Devadas, 2002). So one chip differs from another chip far more than it differs from itself, and that difference is what tells one chip from every other. The maker sets the design; the manufacturing, which the maker does not control, sets the fingerprint.

Because the question of duplication is always one of degree, of how much an adversary must invest to make a convincing duplicate, security is never absolute. Security is always relative to the capability profile of the adversary. A key that is secure against casual intrusion may be trivial for a state-level actor (a national government and its agencies) to duplicate. A cryptographic protocol that resists all known mathematical attacks today may yield to advances in computation tomorrow. A large quantum computer would break today's public key methods, because those methods rest on mathematical problems that ordinary computers cannot solve in any useful time and that such a computer could solve; new methods built to resist it were standardized in 2024, and the move to them has begun. Security exists on a spectrum defined by the gap between the defender's architecture and the attacker's resources.

III. The Event of Spoofing

Suppose a guard admits anyone who shows the right badge, and someone makes a perfect copy of your badge. Whoever holds the copy comes in at night, and the guard's log says that you came in. The log now holds your real visits and the visits made with the copy side by side, and no one reading it can say which were which. Any situation where a given bearer certificate can go from a state of "absolutely universe unique" to a state of "it seems like two identical instances" (to whatever measurement or detection equipment) is the event of spoofing. Spoofing is the event in which a security system is compromised.

Spoofing is a categorical event (a change of kind), and more than a technical failure: it is the transition from a world in which a particular identity is guaranteed to be singular to a world in which that guarantee has been violated. Once violated, the entire chain of trust that depended on that uniqueness is broken simultaneously. Every transaction authenticated by that certificate, every access granted on its authority, every asser-

tion of identity it ever made becomes suspect. Spoofing is the case in which the signature of Chapter 1, which no one can contest while the private key has one holder, can be contested after all.

Because a spoof makes every transaction, every access and every assertion of identity that rested on the spoofed certificate suspect, spoofing is so much more serious than ordinary theft. A stolen key can be revoked. It can be revoked because its loss is known. A key copied without its holder's knowledge has not been stolen in this sense: it has been spoofed, and a private key copied in secret is the worst case of spoofing. A spoofed identity has contaminated every system it ever touched without anyone knowing which interactions were genuine and which were fraudulent.

IV. The Double-Spend Problem

The double-spend problem is spoofing applied to economic instruments. It is a single dollar being spent more than once by one person. In the physical world, this is prevented by the materiality of currency: a physical coin cannot be in two places at once. In the digital world, where all data is inherently copyable, the double-spend problem becomes the central challenge of any digital currency system.

The solution, as described in Chapter 1, is the blockchain: a distributed consensus mechanism that creates a shared, tamper-evident record of which digital tokens (the units of a digital currency) have been spent and which remain available. The tables of Chapter 1 here hold spends: each transaction in a table says that one token has passed from one owner to another. Three things together keep a digital coin from being spent twice. The owner's private key makes each spend a signed statement that only the owner can make. The miners' consensus fixes one order of all spends, so that when two spends of the same coin appear, only the one recorded first counts. The hash function makes that recorded order tamper-evident: no earlier entry can be changed without the change showing in every later

table. The hash function, together with the owner's signature and the miners' consensus, provides the uniqueness guarantee that physical materiality provides in the world of coins and paper.

Cryptography and identity are connected because hash functions create digital uniqueness. Hash functions produce outputs that are, for all practical purposes, unreproducible without executing the exact same algorithm on the exact same input. Hash outputs that, for all practical purposes, no one can reproduce without running the same algorithm on the same input are the closest approximation to physical uniqueness that can be achieved in the digital world. The hash function makes every change of content show, and, as described above, the owner's signature together with the miners' order is what makes one spend count once. So digital uniqueness has two parts: the hash function makes the content of a record unique, in that any change to it shows; and the private key, held by one holder, makes an act unique to its owner, which, with the miners' order, makes one spend count once. A copy of a document has the same hash as the original, because the hash says which content this is and does not say which copy this is. Because a copy of a document has the same hash as the original document, the hash function by itself cannot make one spend count once, and so the private key and the miners' order are needed as well.

Section I said that the tools of Chapter 1 gather every bearer certificate into one key. Think of a ring of door keys, an office badge and a bank card, each a separate thing that can be copied. Suppose the door, the office and the bank each keep your public key on file, and each asks you to sign a fresh message with your private key whenever you arrive. The ring, the badge and the card have been replaced by one thing to keep safe.

V. Data Ownership and Self-Sovereignty

The question extends beyond currency to the ownership of data itself. Cloud services and large technology companies have a huge collection of data about what is going on in people's lives and work, and it is very hard not to participate in what everybody is doing, which is using those services. How can you own your own data, in place of leaving it with those companies, and not allow yourself to be mined (to have your data harvested by those companies)?

The question of how you can own your own data, in place of leaving your data with the cloud services and large technology companies, is the question that motivates the development of Web3 (the name, coined in 2014, for internet services built on blockchains, with users holding their own keys). Web3 is developed to ensure privacy, to restore data ownership to the individual, and to provide cryptographic guarantees that personal information remains under personal control.

The connection to uniqueness is direct. If your identity is unique (cryptographically verifiable and unforgeable), then you can assert ownership over data associated with that identity. Think of a locked trunk left in a warehouse. The warehouse can store the trunk and move it around, and only the holder of the key can see what is inside. In the same way, a company can store data locked under your own key, and the company holds only ciphertext and can read nothing until you unlock the data. What the company cannot read, it cannot harvest, and that is how owning your data answers the mining. Without unique identity, there is no foundation upon which ownership claims can rest. The bearer certificate problem and the data ownership problem are the same problem viewed at different levels of abstraction (the first looks at one key, the second at everything tied to one identity).

VI. Uniqueness and Comparison

Who decides that two things are two? Suppose I set two keys side by side on a table and compare them. The keys stand in two places. If I find them the same in every way I can measure, I have found a sameness of what they are across a difference of where they are. If I find a scratch on one that the other lacks, I have found a difference of what they are across a difference of where they are. Now suppose I look at one key, in its one place, and look again. If I find it unchanged, I have found a sameness of what it is across a sameness of where it is. If I find it changed, I have found a difference of what it is across a sameness of where it is. The definition of uniqueness given above rests uniqueness on comparison, and that joins it to the Immanent Metaphysics (the formal philosophical system set out in my book *An Immanent Metaphysics*). In the terms of *An Immanent Metaphysics*, symmetry is a sameness of content across a difference of context, continuity is a sameness of content across a sameness of context, asymmetry is a difference of content across a difference of context, and discontinuity is a difference of content across a sameness of context. With the keys on the table, the content was what each key is, and the context was where each key is. So the four findings about the keys were, in order, symmetry, asymmetry, continuity and discontinuity.

An instance is unique when every comparison that sets it beside something in another place finds asymmetry; spoofing is the event in which such a comparison finds symmetry instead. Every such comparison is an act of knowing, and the Incommensuration Theorem, a structural fact about the nature of knowing, holds of every one: symmetry and continuity cannot both be applied, simultaneously and fundamentally, to any event or comparison; and neither can asymmetry and discontinuity (*An Immanent Metaphysics*, aphorism 1.43-2). Where a system tells an instance apart from every other instance, the comparison by which the system does so applies asymmetry. So, by the Incommensuration Theorem, the comparison that applies asymmetry cannot, at the same time and fundamentally, also apply discontinuity. In the case of uniqueness, to apply discon-

tinuity is to find the instance different from itself within the instance's own context. Whatever a system can hold as unique is therefore always a thing that the same comparison does not, at the same time, find changed within that thing. The vault tells its key from every other key by the comparison it makes when the key turns, and in that comparison the key is taken as the same key, unchanged in itself. So the question to ask of any key, signature or coin is this: by what comparison, made by what system, is it held to be one of a kind?

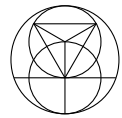
VII. Summary

Uniqueness is the foundation of all security. Every cryptographic protocol, every authentication mechanism, every trust architecture depends on the proposition that certain patterns can exist in one and only one instance. Spoofing, the event that violates this proposition, is therefore a categorical breach in the foundations of trust itself, and more than a technical failure. Uniqueness and distinguishability are one question asked from two sides: an instance is unique, for a given system, exactly when no comparison that system can make finds a second instance that it cannot tell apart from the first. Section VI shows, from the Incommensuration Theorem, that what a system can hold as unique is always something the same comparison does not, at the same time, find changed within itself.

The tools described in Chapter 1 (hash functions, public key cryptography, digital signatures) are precisely the mechanisms by which uniqueness is created and maintained in the digital domain. They answer the vulnerability that every bearer certificate has, and they are the foundational guarantee upon which all digital trust is built. They answer it by gathering every bearer certificate into one, the private key. A private key is itself a bearer certificate, since it grants access to whoever holds it, and its uniqueness rests on its having one holder; the physically unclonable function of Section II is a way of keeping that one key unique. What the tools of Chapter 1 leave in a person's hands is that one key, and Section V

showed what that key is for: data locked under a key that one person holds stays under that person's control. In this book, data is 'personally controllable' when the data is locked under a key that one person holds, and so stays under that person's control.

If everything is gathered into one key with one holder, what happens when that one thing fails? How are the parts put together so that no single failure exposes what a design protects?



Chapter 3

Architecture of a Secure Query System

How are the parts put together so that no single failure exposes what a design protects? This chapter answers with one design: the architecture of a secure electronic query system designed for a research institution. The system enables authorized researchers to determine sample availability without exposing underlying data. The architecture employs defense in depth: air-gapped source databases (held on computers with no network connection of any kind, which data reaches only on media carried by hand), encrypted physical media transfers, biometric authentication (proving who you are by a measurement of your body, such as a fingerprint), and encrypted network communications. In Chapter 1 I gave the parts, and in Chapter 2 I named what every one of the parts depends on. In this chapter I show the parts assembled into one design, in which each layer protects what the other layers might lose. The Appendix gives the specification of this design in full.

I. What the Design Is For

Suppose you once gave a sample of your blood to a research hospital, and your name and your medical history are filed with it. Years later, a researcher planning an experiment wants to know whether the hospital has enough samples from people like you. The researcher needs one fact, and

you need everything else to stay private. This design is worth walking through because it has to give that one fact and keep the rest even when parts of it fail. The design I describe here is a system that answers one question about private data without ever showing the data: are samples that meet a researcher's criteria available? It gives researchers that answer conveniently and securely, through a web page made for the purpose. A researcher working on a grant proposal needs to know whether the samples that an experiment requires are currently available. The system helps researchers decide by providing a specialized web interface to which authorized researchers can submit availability queries. A researcher might ask, for example, whether samples from people over sixty who have diabetes are available. The system answers yes or no, and it shows nothing else.

However, the nature of the information demands the strongest possible security measures. Data that would be damaging to the privacy of the sample providers (the people who gave the samples) must not be made available either directly or indirectly. The system provides information only about sample *availability*, never about the samples themselves. There must be an effective guarantee that no method of unauthorized access could succeed. Chapter 2 showed that security is never absolute, because any one lock can yield to an adversary with enough resources. So this guarantee cannot rest on any one lock. The design meets this requirement by layers: no success at any one layer reaches the protected information, and even a total compromise reveals only whether samples are available. It can reveal no more because the query system holds no more: the data extracted from the source databases (the hospital's full records, where your name and your medical history are filed) is only what availability answers need (Appendix, Section III).

II. Seven Layers of Security

The security architecture employs seven layers. Around all seven layers stands the air gap, which is not counted among them: the computers that hold the source databases are not connected via any form of network to any other machines, and data leaves them only on hand-carried physical media. The air gap is not counted among the seven because the seven guard the system that researchers reach over a network, and the air gap keeps the source databases out of that system altogether.

Only what availability answers need is extracted from the source databases, and it is encrypted before it is carried out by hand. It goes to intermediate hosts. A host is a computer on which part of the system runs, and one intermediate host stands at each source database location. The intermediate hosts are connected via a secure communications link to the query processing server, which answers the researchers' questions. A physically separate security server handles login. Think of an office in which the receptionist who checks your identification at the door has no key to the records room, and the clerk who has that key never decides who may come in. An attacker who takes over one of the two servers has not thereby taken over the other.

The seven layers are these, and each of them stops an attack that the others do not stop; that is what independent means in this chapter. First, the software is strongly resistant to all known forms of unauthorized login, requiring both password and biometric verification. Second, all network communications use encryption, with resistance to stream injection attacks (inserting forged packets into a live connection, packets being the small pieces in which data crosses a network) through positive state synchronization (a shared state that the two ends of a connection check on every message, so that a forged packet is refused). Third, user login is accepted only from registered host machines (the researchers' own computers, each one known to the system) with matching network address patterns (the numbers that locate a machine on a network).

Fourth, information on intermediate hosts is stored in encrypted format, and all physical transfers use encryption. Fifth, the host machine is secured according to best practices for network-connected systems. Sixth, the host is isolated so it cannot serve as a platform for unauthorized access to any other institutional systems. Seventh, even in the event of total system compromise, the information selected for publication (the information the system gives out in its answers) is chosen to preserve the privacy of sample providers. This holds for a series of questions as well as for one: the answers are chosen so that no series of availability answers, each of them true, narrows down to a single sample provider. Suppose a researcher asks whether samples are available from people over sixty, then from people over sixty in one town, then from people over sixty in that town who have one rare illness. Each answer is true, and the last of them could point at one neighbor the researcher knows. The answers are chosen so that such a series never arrives at one person. The Appendix gives the rule that does this and the reason the rule is needed.

In these layers the parts of Chapters 1 and 2 are put to work. The keys of Chapter 1 do the locking in the second and fourth layers: what crosses the network, what is stored and what is carried by hand are all ciphertext to anyone without the key. The uniqueness of Chapter 2 does the admitting in the first and third layers: a password is a bearer certificate and can be copied, so the design also asks for a measurement of the researcher's own body, and for a machine that the system already knows.

The table in the Appendix sets the seven layers side by side, and it shows which layers share a dependence: the first and third rely on the security server, and the second and fourth rely on the keys. The air gap and the seventh layer rely on neither, and so they hold even when both fail. The air gap and the seventh layer are the two guards that keep the system secure even if multiple layers are breached. Whatever else falls, the source databases stay behind the air gap, and the query system holds only the data needed for availability answers, and answers nothing but availability.

III. Defense in Depth

The architecture embodies the principle of defense in depth. Each layer of security operates independently: the air gap protects source databases even if every network component is compromised; encryption protects data in transit even if physical media is intercepted; biometric authentication protects access even if passwords are stolen; and the query-only interface protects detailed information even if the query system itself is breached. Independent here means that each layer stops an attack the others do not stop; two layers can still rely on the same thing, as Section II shows.

Suppose an attacker steals a researcher's password. The login still asks for a measurement of the researcher's body. Suppose the attacker gets past that as well. The login is still accepted only from a registered machine. Suppose the attacker takes over an intermediate host. What is stored there is encrypted. Suppose the attacker gets the keys too. At that point the attacker still holds only the data needed for availability answers, and the source databases, with all the rest of the data, are still behind the air gap.

Think of a librarian who checks the closed shelves and tells you only whether the library has what you want. The librarian knows what is on the shelves; you learn yes or no. The design separates what the system *knows* from what it *reveals*. The query system has access to the abstracted data (the minimal extract from the source databases) needed to answer availability questions, but the security architecture ensures that even a fully compromised system can reveal only availability status, never the underlying sample data.

Security is measured by what it leaves in the person's hands, and the person this design protects is the sample provider. What the design leaves in that person's hands is that person's privacy: even a total compromise reveals only whether samples are available, and no series of true answers narrows down to that one person.

IV. Summary

A secure query system for sensitive research data must be designed so that no single point of failure can expose protected information. The architecture described here achieves this through air-gapped source databases, encrypted physical transfers, separated query and security servers, biometric authentication, and a query interface that answers availability questions without exposing any underlying data. Each layer of security provides independent protection, so that the system remains secure even if multiple layers are breached. A system is resilient against failure when the system stays secure while parts of the system fail, as this design stays secure when multiple layers of the design are breached. Trust in this design rests on no single part, and what the design leaves in the sample provider's hands is that person's privacy.

The first three chapters have dealt with information systems. Every one of those systems, the keys, the servers and the web page, runs on electric power. So what threatens the system everything else depends on, the electric grid, and who should pay to protect it?

PART II

Infrastructure and Resilience

What threatens the system everything else depends on, and who should pay to protect it?



Chapter 4

Infrastructure Security: Ranking the Threats and Deciding Who Pays

> What threatens the electric grid, and who should pay to protect it? This chapter answers by ranking three threat models (a threat model is one way a system can be brought down, seen from the point of view of an adversary) by adversary investment, and by the question of who pays for grid hardening (building the grid to withstand such threats). The first three chapters dealt with information systems; this chapter ranks the threats to the system everything else depends on, the electric grid, by what an adversary must invest.

I. Why the Grid Matters

What would happen to me if the power went out and stayed out? Suppose a whole region loses power for weeks. The food in the refrigerator spoils. The water stops when the pumps stop. The phone goes dead when its battery runs down and the towers lose their backup power. The card reader at the store does not work, and the hospital runs on its generators only as long as their fuel lasts. The electric grid is the system that generates electric power and distributes it to where it is used. We depend a great deal on the grid. We depend upon it in a way that can be set beside COVID-19 and compared. In how great an impact each has, the grid and COVID-19

compare in full: the grid's importance can be shown the way the pandemic's effects were shown, through the economy and through what happened to individuals, states and businesses. Think of the months when shops closed, schools sent children home and businesses failed. A long loss of the grid would reach those same households, shops and schools. The impact is very easy to understand, and our dependence on the grid is beyond any real argument. So what threatens the system everything else depends on, and who should pay to protect it?

II. Three Threat Models Ranked by Adversary Investment

How much would it cost someone to do this to us? Chapter 2 asked the same thing of a key: how much effort, how much resource, how much specialized knowledge must an adversary invest? Seen from the point of view of an adversary, the three threat models (the geomagnetic storm, the cyberattack and the physical attack) sort into a clear order. Threat modeling has to have some idea of what an adversary would do, and the three line up by the amount of effort, commitment, and involvement that an adversary would need to have.

With the geomagnetic storm (the solar storm scenario), the only thing an adversary has to do is just sit back and watch. They have no involvement, no investment, nothing. Their only other task is to make sure their own grid is solid enough that they survive. Sitting back and watching is zero commitment on the part of any agentic person (a person acting on purposes of their own), state actor, or terrorist organization.

To understand what such a storm does to the grid, one needs to understand what a coronal mass ejection is (a cloud of charged gas thrown from the sun), something about DC offsets and AC offsets (the steady currents a solar storm drives into power lines, on top of their normal alternating current), transformer design (a transformer is the device that changes the voltage of the power on a line), and the reason why cascade failures happen (a cascade failure is a failure that causes further failures, as one

tripped line overloads the next). In plain words: the sun throws off a cloud of charged gas; when the cloud reaches the earth, it drives a steady current into long power lines; the steady current pushes transformers beyond what they were built to carry; and a transformer that fails throws its load onto the next part of the grid.

With the cyberattack, very little investment is needed: some hacker somewhere decides they want to shut down the United States, without really understanding the implications. The amount of investment and involvement is quite low. A state actor would be able to cover its tracks completely, and for it the investment is moderate to low, the commitment is almost none, and there are no political ramifications if it manages to do the deed and get away with it. Suppose the intruder reaches the computers that tell the grid's switches when to open and close, and orders the switches open. The power stops, and nothing has been physically broken.

For the third threat model (an actual physical attack on critical infrastructure), think of a special-operations team or some terrorist cell putting together a coordinated effort. They land people on the shore, work their way in, and manage to take over enough critical resources to cause real damage. The physical attack is a somewhat higher level of expense than the cyberattack. Rather than just buying a computer and maybe a geek's time, the attacker has to handle the logistics of outfitting a team of people: the planning and coordination. The physical attack breaks the equipment itself, and the power stays off until the equipment is replaced.

The three go from zero involvement and commitment to the involvement and commitment of an outfitted team. So the ranking says how far each threat is within reach: the first waits on no adversary at all, the second is within reach of some hacker somewhere, and the third takes a team. The ranking of the three threats by how far each is within reach is the spectrum of Chapter 2, the gap between the defender's architecture and the attacker's resources.

Threat model	What the adversary must invest	What the adversary must commit	What the adversary must have
1. Geomagnetic (solar) storm	Nothing: "just sit back and watch"	None	Only a grid solid enough to survive
2. Cyberattack	Low: "just buying a computer and maybe a geek's time"	Almost none; a state actor can cover its tracks	Skill, even without understanding the implications
3. Physical attack on critical infrastructure	Higher: outfitting a team, with logistics, planning and coordination	More than a cyberattack	People able to land, work their way in and take over critical resources

Think of a house built on a coast that has hurricanes. Its roof is strapped down and its windows have shutters. Hardening does the same work for the grid, each measure matched to one threat: equipment that can carry the stray current of a storm, control computers that an intruder cannot reach, and sites that an outfitted team cannot simply walk into. A grid built this way is resilient: it keeps working through what would otherwise bring it down.

III. Who Pays for Grid Hardening

Whether an adversary must invest little or much, the people left without power are the same people. So who pays: the companies that sell the power, or the people who have come to depend on it? On the question of who pays to protect the grid, this chapter uses some tools from the theory of non-relative ethics, whose two principles (the symmetry and the continuity of effective choice) hold for every person and situation alike. Effective choice is a choice that has effect: one that changes what happens in the world. The two principles are set out in *The Two Principles of Ethics*.

Each of the two principles carries one step of the argument that follows: the symmetry of effective choice looks at what is the same for every person, and the continuity of effective choice looks at what carries on unbroken over time. The situation has to be looked at from two perspectives: what is the function of government, and what is the function of markets?

The function of government (this is a classical answer going back a thousand years) is to protect the land and the people. Regardless of the form of government, one needs this as a first premise in order to have any sense of what the notion of good governance even means.

Markets are about making good things happen: prosperity from the inventiveness of people who are motivated through patent law, commercial ownership, deployment of capital. The rationale for why we have markets at all is that we can create wealth or betterment through the dynamics of exchange.

Roll the clock back a hundred and some years (Edison, Tesla, Westinghouse): they built electric power, finding ways to generate it, distribute it and use it. Much goodness happened. There was optionality then: people had horse and buggy, kerosene and candles. The electricity was a better way of solving problems, but we were not dependent upon it because we had alternatives.

Then roll the clock forward a few generations, and we have become more and more dependent on it. The question becomes: who is responsible for the dependence? It is a little like a drug company discovering a miracle drug: no real side effects, increases health, increases longevity, makes people smarter. You cannot fault a company for selling that, and you cannot fault people for becoming dependent on something like that.

The work of addressing addiction and dependence does not fall on the developer of the goodness. When the industry raises the concern that this is expensive and impacts its bottom line, there is a certain amount of legitimacy to it, because the business was developed as a business. Originally it was voluntary; now it has become critical infrastructure, but that change happened slowly. Look for the day on which anyone chose to depend on

the grid, and no such day can be found: each generation made the next small and sensible choice. This step is the one the continuity of effective choice gives. That principle looks at what carries on unbroken over time, and here the choosing carried on unbroken: no single generation's choice can be set apart from the choice before it as the choice that made the dependence. So a dependence formed slowly, by no single choice, carries no fault.

If there is a dependency created on the part of the land and the people for otherwise benign reasons, then the government, as an independent entity, would therefore need to find a way to allocate taxpayer money to protect the taxpayers from their own dependence. Everyone depends on the grid in the same way: the owner of a power company and a child doing homework lose the same light when the grid goes down. This step is the one the symmetry of effective choice gives: the dependence is the same for all, so the obligation falls on the people as a whole. The cost is paid out of taxpayer money. The industry is not the party that pays: the protection of the grid is about the security and welfare of the nation.

The same argument reaches every system that the land and the people have come to depend on in the same way, for otherwise benign reasons: water, communications and fuel, as well as the electric grid.

Security is measured by what it leaves in the person's hands, and a person's agency rests on the systems everything else depends on, the electric grid first: without power, the digital keys, the software and the channels of communication in a person's hands cannot be used. The cost of protecting those systems is a public responsibility, because the dependence on those systems is the public's.

IV. Summary

Three threats to the electric grid rank by what an adversary must invest: the geomagnetic storm, the cyberattack and the physical attack. The costs of protecting the grid are a public responsibility, because the dependence

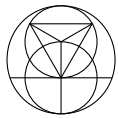
on the grid is the public's. Without the grid, what a person holds in hand cannot be used, so a resilient grid is part of what keeps those things usable in that person's hands.

In this chapter I have looked at what threatens a system from outside. A system can also do harm by its own working, to the people and the living things its effects reach. Whatever steers a system is its controller, and the operators of the grid are one example. So how far can any controller keep a system from harming what lies downstream of it?

PART III

Control, Complexity, and the Limits of Security

How far can any controller protect what lies downstream of a system it cannot predict?



Chapter 5

Control Theory, Observability, and the Architecture of Safety

How far can any controller keep a system from harming what lies downstream of it? This chapter answers with a model of the limits of control in complex systems, using the hash function as a model of algorithmic complexity (the length of the shortest description that produces a thing). It covers observability and controllability (how much of a system a controller can see and how much of it the controller can change) as duals (mirror images of each other), layers of abstraction in information systems, a model of a controller, the system it controls, and the living things downstream, artificial intelligence as the controller, and the implications for predicting and preventing harm across multiple levels of abstraction. In Chapter 4 I ranked threats. In this chapter I ask how far any controller can keep a system from harming what lies downstream of the system.

I. Observability and Controllability

Can anyone, or any machine, keep a complicated thing from hurting the people and the living things around it? Chapter 4 asked what threatens a system. In this chapter I ask what the ones who steer a system can do. Think of a factory beside a river, with farms and a town below it. The factory is what this chapter calls the system: the thing that runs and puts

things out. The person at the valves is the controller: the one who steers the system toward a goal. The fish, the crops and the people below the factory are the living things downstream: the living things that the effects of the factory reach. The person at the valves wants what comes out of the factory to do no harm below. How far can that person succeed, when the factory does more than that person can foresee?

Now step into the factory's control room. It has a wall of lamps and a wall of switches. Wires run out from the parts of the factory to the lamps, so each lamp shows what one part is doing. Wires run in from the switches to the parts of the factory, so each switch changes what one part is doing. Ask which parts have a lamp, and you are asking how much of the factory can be seen. Ask which parts have a switch, and you are asking how much of the factory can be changed. The two questions have one form, with the wires read in opposite directions. And a part with no lamp is a part whose response to a switch no one can check.

Control theory is the study of how a controller keeps a system near a goal; observability is how much of the system's state the controller can see; controllability is how much of it the controller can change. Observability and controllability are duals of each other (each is the mirror image of the other): in a lot of systems they are mathematically the same thing seen from two sides. A real machine may add small differences of its own; with those set aside, the relation in the mathematics is a clean one: what holds for seeing holds, mirrored, for changing. If you cannot fully observe a system, you cannot fully control it. This holds for control by feedback, where the controller acts on what it observes, as the person at the switches acts on what the lamps show.

Control needs both observation and a comparison with some goal state, and observability is connected to predictability as well as to control. A household thermostat shows the first half of that sentence: it reads the temperature of the room, compares the reading with the number someone has set, and only then turns the heat on or off. The second half matters because harm has to be stopped before it happens. To act in time, a controller must see not only what a system is doing now but what the system

will do next, and to see what a system will do next is to predict it. So from here on in this chapter, observing a system includes knowing what will come out of the system before it comes out.

Layers of abstraction (the levels at which one thing can be described, from its bits up to what those bits mean) add a further dimension to how we think about control theory: the degree of abstraction. I might be able to observe all the bits of a given message, for example (error correction in message transmission, the adding of redundant bits so that damaged bits can be repaired, is a model of how to think about this): I might be able to do error correction at the level of bits, and still not know anything about what those bits mean.

Error correction itself works layer by layer, and the method at one layer does not catch the errors of the next. On the internet, the lowest layer is the wire: which voltage stands for a zero, which for a one, and which transitions make up a signal are chosen so that brief disturbances are suppressed. At the next layer, a parity bit on each byte (a group of eight bits) shows that a bit has flipped. Above that, a check value (a cyclic redundancy check, or CRC) computed over each packet (a small piece of a message, sent on its own) shows whether the packet is whole or has lost some of its bits; the check on each byte and the check on each packet are different error corrections. The packets are assembled into messages, and the messages into larger units of meaning that make sense only within a protocol, and the errors there are of another kind: an acknowledgment arrives before the packet was sent; one packet brings three acknowledgments; a packet sent first arrives second, and the packets must be put back in order. No amount of error correction that detects flipped bits or changed bytes will notice that the protocol is out of sync. Higher still, in TCP/IP (the protocols by which the internet carries packets between machines and puts them back in order) and in the state of an application, what counts as an error changes again, and it is caught in yet another way. Redundancy and double checking are used at every layer, but the higher the layer, the more ambiguous it becomes what counts as an error, because it becomes more ambiguous what counts as a message, an action or a transaction.

Between two people speaking, there is no way to check that the listener has understood exactly what the speaker meant; at that level of abstraction people hope for the best.

Scale works the same way. A web service with a database behind it, run on one computer with one web server, may serve 10,000 users well. Beyond that, people using it at the same time call for a second host, and the second host brings new problems: how to manage the database, whether to replicate the data, how to keep the copies in step, how to track the state of each session. Going from 10,000 users to 100,000, a modest step from four orders of magnitude to five, means re-architecting the service, and so does going from 1,000 to 10,000, or from 100,000 to a million. Each increase in scale brings a new set of problems, different in kind, and what solves the problems at one level says little about the next. So no one set of techniques solves at every level of scale, or at every level of abstraction. A claim that error correction solves at every level, or that artificial intelligence can be given error correction and alignment (goals made to match what people intend) at every level at once, expects error correction to work where ambiguity is greatest and discreteness least (discreteness is how cleanly a thing falls into separate values, as a bit is either zero or one) as it works for bits, bytes and packets. Communication theory, design theory and the math all show that this expectation is false, and so does working on the architecture of systems: the higher the layer, the more ambiguous it becomes what counts as an error, and what solves the problems at one level says little about the next.

Return to the message whose bits I can observe. An encryption could have been applied to it, with some hidden message inside it. I can ensure the bits are correct, but I cannot ensure that the meaning is correct. I would need to unwrap the meaning somehow and treat that as an observable. And the meaning may have significance in some larger context, like "attack at dawn", for example. It is the classic coordination problem (how separate parties come to act together when each knows only what the messages carry). Picture two armies on two hills with a city in the valley between them. They win only if they attack together, and they can talk

only by sending a runner across the valley. The runner carries "attack at dawn", and each commander needs to know that the other has read those words the same way. The meaning of "attack at dawn" is very specific to the context, which we might not be able to observe just looking at the message.

Suppose the context is not military but theater, or a business deal: then "attack at dawn" might have a very different meaning. To some extent, that is because of mutual information between the people communicating (what two parties already share that shapes how a message is read; in information theory the term names a quantity, how much one signal tells about another; here it names the shared context itself). There is a hidden form of information, a hidden context, here that is very important.

And that hidden context can live at multiple levels of abstraction. The higher we go up the abstraction ladder, the more this back channel of information matters.

The rest of this chapter builds a model in four steps: a scale of how predictable a process is, from the direct mapping to the hash function (Section II); a controller placed in front of a system (Section III); the living things downstream of the system (Sections IV to VI); and the inequality that follows (a statement that one quantity is greater than another; Section VII).

II. The Hash Function as a Model of Complexity

How can anyone say how predictable a process is? One way to talk about this is to set up an example system. Take a hash function, like MD5 (an older hash function, now broken). It still does roughly what is needed as far as this metaphor is concerned, though it is discredited for use in security, for a reason Section III gives. A certain number of bits go in, an algorithm muddles those bits up, and they come out as an output bit sequence. If we change one of the input bits, we know it is going to change

quite a few (about half, and no one can say which) of the output bits. The muddling of the bits, in which a change to one input bit changes about half of the output bits, is how a hash function works.

Let us treat this kind of algorithm as an exemplar. At the other end of the spectrum, we could look at something which is the simplest possible transform: it takes input bits and maps them directly to output bits. No algorithm at all. Let us call the no-algorithm case zero and the hash function case one: one hundred percent algorithmic transformation. Those two numbers give the share of the output bits that no one can call in advance. In the no-algorithm case, no one is in doubt about any output bit, so the share is zero. In the hash function case, about half of the output bits flip, and every output bit might be among that half, so all of the output bits are in doubt, and the share is one.

Other algorithms resemble the zero case more, or resemble the one case more, or lie somewhere in between. We could end up with all permutations (every possible arrangement of the bits): in a hash function, there is no clear way to predict which output any given input will map to, except by executing the hash function.

The predictability depends purely on running the algorithm. If there were any simplification, we could observe the outcome in advance. Observability in this case means running the algorithm, and non-observability means I cannot predict the outcome: I cannot see or characterize the output bits except by running the algorithm.

We now have this graded series between zero and one where we can say: is any given situation more like the simplest case or more like the transforms-every-bit case? The graded series between zero and one is a useful metric, because the series gives us a way to talk about how fragile the system is. Fragility here means how easily a small change breaks what depends on the output. The nearer a process stands to one, the less anyone can say in advance what a small change at its input will do, and so the less anyone can protect what depends on its output.

One measurement for this scale already exists: does each output bit flip with a chance of one half whenever any one input bit changes? That test is the strict avalanche criterion of Webster and Tavares (1985). The zero case, which maps each input bit directly to an output bit, changes exactly one output bit, which anyone can name in advance, for each input bit changed; a good hash function meets the criterion. So the share of output bits in doubt when one input bit changes (the bits that might flip, with no one able to say in advance whether they will) places a process between zero and one, and that share is the number Section VII uses when it places biological systems between the zero case and the full hash function case. A system with several levels of abstraction has such a share at each level, and what must be predicted multiplies across the boundaries between the levels (a boundary, in this sense, is the step at which what one level hands on is taken up by the next; Section VII shows the multiplying).

III. The Controlling Algorithm Versus the Controlled Algorithm

Say I have a system that depends very sensitively on us being able to control the output bits to a particular configuration. Say the system produces one hundred output bits and I need bit twenty-three to be specifically a given value over all possible inputs to the algorithm. Could I put another algorithm in front of this algorithm? Call the algorithm that transforms inputs to outputs the system. Can I put another algorithm, the controller, in front of it, such that the controller will output a sequence of bits that will guarantee that bit twenty-three of the output of the system is always a one? That would be a kind of control process.

What I am asserting is that the degree to which the controller can model the system depends upon our having some notion of how the system works that is faster or simpler than the system's own algorithm. This gets us directly into the theory of algorithmic complexity (the length of the shortest description that produces a thing), which says that there are a

vast number of cases where we do not know that there is any simpler version of a given algorithm. If the system is a hash function, there might not be any simpler version than that hash function itself: it may be the simplest representation.

We can say this is likely, because there is a kind of empirical evidence. If the degree to which human beings can figure things out is proportional to their incentive, then the use of hash functions to create blockchains (and the economic viability associated with those blockchains) means that a great many of the ablest people alive have been motivated to figure out some way to predict the outputs of hash functions so that they could break blockchains and therefore prevail at billion or trillion dollar levels. The evidence that these problems are hard is strong, and it is empirical. MD5 and SHA-1 (another older hash function, of the United States standard) were broken by finding two inputs with the same output; that is a different task from predicting the output of a given input, and no one has found a way to do that faster than running the function.

A reader may ask whether that kind of algorithm matters beyond blockchains: it is very useful for blockchains, but it is not what you want your computer to run. An algorithm of the hash function's kind matters beyond blockchains because the world itself may sometimes be implementing a process that is closer to that kind of algorithm.

IV. The Model of Controller, System and the Living Things Downstream

The three names so far: the controller; the system it controls; the living things downstream of the system.

Why should the controller now be an artificial intelligence? The strongest controller anyone proposes for keeping the world safe is an artificial intelligence, so the limit matters most there. The proposal, in plain words, is to build a machine that watches the world and steers it away from harm. Let us now extend this model. I am going to make the system

be the world and the controller an artificial intelligence, and the aim is to control what comes out of the world. The world is not like a hash function in every respect. But there are some respects (like genomics, the study of the full set of genes of a living thing) in which the biosphere and ecosystems themselves have aspects that are closer to the hash function. Think of one letter changed in the long text of a living thing's genes. The change may do nothing that anyone can find, or it may alter a protein that alters what many other genes do, and reading that one letter does not say which. The way to find out is to let the living thing grow and see. That has the shape of the hash function: one small change going in, and what comes out known only by running the whole process.

This model assumes that the world has those aspects. What this assumption is used for is clear: it means that in order to control and handle the whole world, one needs something that can handle systems like the hash function.

If I want to ensure safety for biological systems (in the sense of not disrupting them), I need at least a sufficient level of control over artificial processes (what people's machines and factories do in the world, as the factory of Section I does to its river; these are the part of the world that the controller has the task of steering) so that they do not interact with biological processes to disrupt those biological processes.

This general model with the hash function is valuable. It lets us recognize that the world is composed of parts with various degrees of complexity, and that we care about some parts of the world that are potentially very sensitive and require control or protection. The complexity of protection (how much work it takes to keep a sensitive part of the world from harm; Section VII writes this as the work needed to predict) is a concept we ought to be thinking much more about.

The complexity of protection is where I am going in this chapter. What is the level of complexity of protection? We can start to look at the cascades (here, the way the output of one stage becomes the input of the next) through these kinds of systems. Now put back what Section I placed

below the factory: let us suppose that downstream of the system there is some biological component, the living things downstream. The living things are part of the world, and so is the controller. The factory and the fields below it stand in one valley, and the fields are still downstream of the factory, because what the factory puts into the river reaches the fields afterward. In the same way, the world as a system acts, and the living things are the part of the world that the world's outputs reach afterward. So the model has its three parts: the controller, the system, and the living things downstream. The living things downstream are the unfathomable cellular things that make up bodies and ecosystems. We are going to be exploring them for all the rest of eternity, because there is so much going on in nervous systems and immune systems and all of the ways in which things happen.

V. Cyanide, Caffeine, and the Reliability of Harm

The three names: the controller; the system it controls; the living things downstream of the system.

Which harms can anyone know about in advance, and which can no one know? The living things downstream have fragilities that depend upon certain things in the output of the system. If the output of the system is cyanide, for example, then it is unlikely that things downstream of that are going to work well.

Suppose I am asked to bet whether my potted flower is going to do well if I pour cyanide or arsenic on it. I can make a sensible bet that it is not going to do well. We have a very reliable finding that cyanide and arsenic are destructive to living things. If I were to pour a bit of coffee on the flower, on the other hand, it is a bit more iffy: caffeine is bad for dogs, somewhat good for academics. The interesting part here is the reliability of the badness. Cyanide tends to be bad for living beings because it blocks

the electron transport chain (the chain of proteins by which cells make most of their energy), and that chain is essential for everything. The finding that cyanide harms living beings is very reliable.

Caffeine is bad for dogs because it and its relative in chocolate, theobromine, block a signal that slows the heart (the heart's natural pacemaker, the sinus node, is among the places they act), and dogs clear theobromine slowly. The dog has a far more complicated function in its sensitivity to the alkaloids (the family of plant compounds that includes caffeine and theobromine).

The higher we climb up the abstraction scale (here, the scale of what can reach a living thing, from a bare poison through a molecule such as caffeine, then a protein, to a virus, each a larger and more organized thing than the one before), the harder it is to make predictions. Coffee, because it is much more complex in its relationships to the environment and to various particular creatures, is going to be a lot harder to predict: we would not necessarily know in advance that dogs have this particular feature that makes them more sensitive to caffeine. Caffeine is also a far more complicated molecule than cyanide or arsenic. Arsenic is simpler still: it is an element (a single kind of atom), and the poisons are its simple compounds.

When we look higher up the chain, if I am introducing a prion (a misfolded protein that makes other proteins misfold), that is going to affect things in a fairly broad way. It stands higher on that scale than a molecule of caffeine, being a far larger and more organized thing, and still it is a little simpler than coffee. Simpler here means that the harm from a prion depends less on which creature meets it, since its effect is broad, while the effect of coffee differs from one creature to the next. If I go higher up the scale and start thinking about viruses, then some viruses will affect some creatures and not others in extremely differential ways. So some harms can be bet on with confidence, as with cyanide, and others depend on a particular sensitivity of a particular creature, which no one would know in advance.

VI. The Subspace Analysis of Safety

The three names: the controller; the system it controls; the living things downstream of the system.

We need to control not only individual bits on the output of the system before they go to the living things downstream, but configurations of bits, and think about the whole range of possible implications at multiple levels of abstraction simultaneously. What is the degree of control we need in these circumstances?

Picture a very large wall painted in patches of three colors. Green is safe, red is known to be unsafe, and gray is not known either way. Each spot on the wall is one possible output of the system, with every bit of it set one way or the other, and each time the system runs, one dart lands on the wall. The green may be one broad patch, or many small flecks scattered through the red.

What I need to do is to calculate a kind of probability of the subspace (a region set apart inside a larger space) of the total possibilities: the subspace which is safe versus the subspace which is known unsafe via very simple predictable outcomes like cyanide, or very likely to be unsafe given certain generalizations. We can throw a bunch of heuristics (rules of thumb) at it. But somewhere along the way, we are going to end up with two secondary measures.

One measure is the relative volumes of the subspace: safe versus unsafe versus unknown. The other is the amount of surface area or the boundary distinguishing the safe and unsafe volumes (the boundary, here, is the edge between green and red on the wall): the degree of mixing that occurs between the two. This describes the degree of precision we need versus the degree of accuracy we need. Accuracy is landing in the safe region at all; precision, as these words are used here, is landing in it every time when the safe region is scattered. The volume of the safe region decides the accuracy needed, and the length of its boundary decides the precision needed. A wall that is mostly red calls for good aim; a wall on which the green is flecks among the red calls for a dart that lands on a

fleck every time. The accuracy is: did we hit the region of space that is the safe subspace? And if the proportion of that space relative to the total possibility space is relatively small, then we need the system to be really accurate. If that subspace is scattered across the total space, then we need it to be accurate and precise.

Even a little bit of deviation in the output bit configuration at any of the abstraction levels defining the overall possibility space can result in hitting the bad region because it happens to be sitting right next to the good region. The level of control needed on the configuration bits on the boundary between the system and the living things downstream (here, the place where the system's output reaches them) needs to be quite high depending upon the number of these metrics (the values that the two measures above come to). So the smaller and the more scattered the safe region is, the greater the control needed.

VII. The Inequality of Control

The three names: the controller; the system it controls; the living things downstream of the system.

Is there a point at which no skill in the controller is enough? I am saying that even though the algorithms observed in actual biological systems are certainly not all the way to the hash function (they are somewhere between the zero case and the full hash function case), their number on the scale (the share of Section II: how much of what comes out is in doubt, with no one able to say in advance whether it will change, when one thing going in changes) is high enough that as I cascade multiple levels of "this algorithm has to predict that algorithm" (stacking such levels, each resting on the one before), an inequality results.

Suppose a sentence has to pass through three translators in a row, and each translator might render it in any of ten ways. To know what comes out of the first translator, I must follow ten possibilities. To know what comes out of the second, I must follow ten more for each of those ten: one

hundred. To know what comes out of the third: one thousand. Each handing-on is a boundary in the sense of Section II, and each boundary multiplies what I must follow.

As the intervening stages increase, and the level of precision and control required rises, eventually an inequality is reached where it does not matter what I do in the controller. There is no methodology in the controller that can constrain the system such that it does not hurt the living things downstream. It is in the multiple boundaries that the fundamental limitation resides.

When we look at actual functions, the interesting fact is that their structures do sometimes modularize (divide into parts that can be worked with separately). Evolution cannot handle too much lack of modularity. In software, we design modularly, because we learned at great cost that otherwise it is very hard to maintain. Typically we solve coordination problems by forcing the problem to become a problem that is solvable. Forcing a problem to become solvable is the remedy where that remedy is available. A system that a person builds can be made simple enough to predict, and in Chapter 7 I show systems of that kind. The world downstream of a system cannot be made simple, and there the limit holds.

Living things are built from parts. Can they then be predicted part by part? The modularity of living things agrees with the control-limit argument, rather than cutting against the control-limit argument. Biological systems have high degrees of modularity, but the modularity is not infinitely high, so prediction part by part reaches only as far as the parts are separate. A biological system is not the simplest possible system, but it is not equivalent to a hash function. It is somewhere in the middle. The changed letter in a gene of Section IV stands there: some such changes do nothing anyone can find, which the hash function never allows, and some alter many other genes, which the direct mapping never allows. And the question of where exactly in the middle determines the constraints on controllability.

Stated as an inequality: to keep the system from harming the living things downstream, the controller must know, before the system acts, which outputs of the system fall in the unsafe region. A controller can predict only by some method it has, and every method of prediction is a description of the system it predicts. When the system is like a hash function, the shortest description anyone has is the system itself, so the controller's prediction costs at least as much as running the system. The record of the search stands behind that step: the value locked in blockchains has given a great many able people, for more than a decade, a reason worth billions of dollars to look for a shortcut through hash functions (Section III), and none has been published; and the mathematician Andrey Kolmogorov showed that most strings of bits have no shorter description at all, so having no shortcut is the ordinary case and having one is the rare case.

Think of a clerk told to keep, in one ledger, a full record of everything that happens in the office, including the clerk's own writing in that ledger. Every entry is one more thing to enter, and the clerk can never get ahead of the office. And where the system is the world, the step holds from the structure of the case: the controller is part of the world it must predict, so it must predict itself as well, and no part can compute the whole of which it is a part faster than that whole runs. The clerk's trouble has been proved in general: David Wolpert proved a form of this, that any physical device that makes predictions about the universe it belongs to has some fact about that universe that it cannot predict.

Each further boundary, from the system to the living things downstream and within their own levels of abstraction, multiplies what the controller must predict, and the precision needed rises as the safe and unsafe regions mix. The controller is itself a part of the world, with a finite capacity to compute. When the prediction needed exceeds what the controller can compute before the system acts, no method in the controller can keep the system from harming the living things downstream.

Written out with the names in full, the inequality reads:

work needed to predict, before the system acts, which outputs of the system fall in the unsafe region

is greater than

work the controller can do before the system acts;

and therefore

no method in the controller keeps the system from harming the living things downstream.

The left side grows with every further boundary, since each multiplies what the controller must predict, and with every mixing of the safe and unsafe regions, since mixing raises the precision needed. The right side is bounded because the controller is itself a part of the world and computes finitely.

Has anyone else reached this limit by another road? Two results of other authors stand behind the steps of this inequality, the first in two forms. Kolmogorov's result, given above, is the first form: most strings of bits have no description shorter than themselves. Stephen Wolfram's computational irreducibility is the second form, the same claim for physical processes: for many of them, no shortcut predicts the outcome faster than running the process itself.

Suppose nine different winds can strike a small boat, and its helmsman has three different ways of setting the sail. Even a perfect helmsman can bring the nine winds down to three different courses, and no fewer. To hold the boat to one course, the helmsman needs nine ways of setting the sail. The second result, W. Ross Ashby's law of requisite variety, reaches the same limit from the side of the regulator (the regulator is the controller, seen as what holds a system's outcomes in a range): the number of distinct outcomes a regulator can hold the system to is at least the number of distinct disturbances divided by the number of distinct responses the regulator has. A regulator cannot hold the outcome to a safe region with fewer outcomes than the number of disturbances divided by the number of its responses; the more varied the world, the more responses the regulator needs. Its companion, the good regulator theorem of Roger Conant

and W. Ross Ashby (1970), joins control to prediction directly: every good regulator of a system must be a model of that system, so a controller that keeps a system to a goal must contain a model of the system it controls. My book *The Study of Pure Causality* (Chapter IX) gives a third result, which stands behind the step in which the controller must say which outputs of the system fall in the unsafe region: no general procedure (no one method that works for every case) can determine, for an arbitrary program (any program at all), whether that program will finish; and by the same reasoning, given by H. G. Rice in 1953, no general procedure can decide, for an arbitrary program, whether its output will fall in a given region. So no one method sorts the outputs of every program into safe and unsafe.

VIII. Summary

In this chapter I have shown a limit that no method of control can pass. The limit applies where a controller must predict a system, the controller cannot predict the system faster than the system runs, and boundaries multiply what the controller must predict. Where the limit applies, no method in the controller keeps the system from harming what lies downstream of the system. The same limit holds between a person and the tools the person uses. Think of a bicycle and a phone. The rider of a bicycle can tell what each lever will do before pulling it. The owner of a phone cannot tell what the phone will send, and to whom, when one of its programs is opened, and finds out, if at all, only from what happens. A person who uses a tool is its controller, and can keep it from harm only by knowing what it will do before it does it. A tool more complex than its user can follow is, for that user, like the hash function: what it does can be known only by running it. So the systems that leave control with the person are the ones simple enough for the person to predict: inspectable and small.

Security is measured by what it leaves in the person's hands, and this limit says what can be left there: control of a tool stays in a person's hands only when the person can predict the tool.

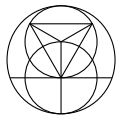
For artificial intelligence the limit means this: what remains in reach is the choice of which systems to build, since control that relies on predicting the world cannot keep the world safe. The full argument from this limit to the safety of artificial general intelligence (artificial intelligence that can learn to do any task a person can do) is set out in *Why AGI Cannot Be Made Safe; The Argument and Its Implications*, and developed in its technical companion, *Why No Control Suffices; The Technical Case for the Impossibility of AGI Safety*, where it is called the control-limit argument. Section I showed a checker of bits who knows nothing of what those bits mean: the checker works at one level of abstraction, and the meaning stands at another. The argument says how far that gap can go: beyond a threshold in the difference between the levels of abstraction at which a controller and a system operate, no control strategy does better than chance (that is, no better than acting at random).

In this chapter I found a limit on what any controller can do: a person can keep control only of what the person can predict. Who, then, has taken that control from the people inside a system, and how was it taken?

PART IV

Technology, Society, and Choice

How is choice taken from people inside a system, and how can they keep it?



Chapter 6

Transparency, Consolidation, and the Erosion of Choice

How is choice taken from people inside a system? This chapter answers by examining the critical balances that sustain complex systems (transparency against opacity, competition against cooperation, and local process against global consolidation) and by tracing how technology departs from natural balance: through consolidation and not through genuine cooperation, so that causal process increasingly compels over individual choice. Through the infrastructure of communication (email routing, encryption protocols, application ecosystems, and hardware lockdown), it shows that an inexorable consolidation is closing every independent path through which choice can enter the system. In Chapter 5 I found a limit on what any controller can do. In this chapter I show control being taken from the people inside a system, by consolidation.

I. Three Pairs in Subtle Balance

Why can I not just run my own thing any more, and who decided that? The question of why a person cannot run that person's own programs any more, and of who decided so, is the question that I left standing at the end of Chapter 5, put in the words of a person holding a phone. Suppose a friend writes a small program that lets your family share recordings, and

you try to put it on your new phone. The phone refuses until a company you have never dealt with has given its permission. A path by which your own choice reached your own phone has closed. A path by which choice enters a system, in this chapter, is any way that a person's own decision can change what the system does: the program you chose runs, and the letter you sent arrives. How do such paths close? A path closes when a balance that keeps a system healthy tips, so the place to begin is with those balances.

What are the primary balances that must be maintained in complex systems? There are three pairs of concepts in subtle balance. The first of the three pairs is transparency and opacity. If transparency goes too high, creativity goes to zero, and there is no innovation and no new development. If opacity goes too high, every kind of criminal organization takes over the world. The balance must be right. Computing shows both sides. A developer whose every unfinished experiment is published and judged at once stops experimenting; and ransomware gangs, which lock a victim's files and sell back the key, ask to be paid in cryptocurrency because its payments are hard to trace.

Where the balance falls depends on scale and on role. A small company that considered publishing its financial figures to all its staff shows one side. A public company must publish its figures to some extent, and at that scale publishing works well enough. In a small company, very few of the staff understand the figures, so once the figures are open to everyone, they become a political matter inside the company, and that can destroy the company. A person in a public role shows the other side: a minister in one country's government. He holds that as a civil servant he must be transparent, so he records every conversation he has, and the recordings can be found on the internet. The minister's choice to record every conversation is a wise choice. Recording every conversation is his commitment to integrity, and the recordings help the people he works with, because the people he works with can check what he said at other times, and he does not have to repeat himself. The transparency that would break a small company is, for a civil servant, part of the role.

The second pair is local versus global: the degree to which phenomena happen locally versus at scale. Think of the clock on a village church and a single time signal that sets every clock in a country. A fault in the church clock shows in one village, and the village can see it and mend it. A fault at the center of the country's signal shows on every clock, and the people looking at the clocks cannot reach it to mend it. The third of the three pairs is competition versus cooperation. In nature, these two forces are almost exactly at parity (equal in strength), within something like one hundredth of a percent: the stronger exceeds the weaker by something like one part in ten thousand. Even a tiny shift in the balance produces phase change: a slight increase in competition moves a system from abundant life to sterility. Rabbits brought to Australia show the first: freed of the predators that held them in check, they stripped whole regions of the plants that held the soil. A slight excess of cooperation produces monocultures that become fragile, brittle, and eventually die. A plantation of one crop shows the second: the Gros Michel banana, grown around the world as a single clone, was all but wiped out of commercial growing by one disease, Panama disease, in the middle of the twentieth century.

II. Consolidation Is Not Cooperation

How does technology depart from this natural balance? Technology creates consolidated blocks (many people or machines held together under one controlling party), but these are not the same as cooperation. The difference is critical. Think of a line at a bakery and a turnstile at a train station. The people in the bakery line stand one behind another because each has chosen to, and any of them can step out or wave another person ahead. The people at the turnstile also pass in single file, and the bar makes them do it. Both show one orderly line, and only the first is cooperation. In genuine cooperation, each person locally, on their own choice, agrees to follow the shared rules: agrees to move with the uniform pattern of the social contract. But with certain kinds of cryptographic enforce-

ment, alignment with a consolidated block can now be compelled without any choice or permission on the part of the individual. Three kinds are at work now: code signing, under which a device runs only software signed by a key the platform owner approves; device attestation, under which a service refuses a device that cannot prove, by a signature, that it runs unmodified software; and certificates that must be renewed from an authority (an organization that browsers trust to issue certificates; Section III). Each of the three is the digital signature of Chapter 1 held by someone else. The signature protects whoever holds the key, and consolidation is the case in which the key that decides what runs is held by someone other than the person it runs for. The question of agency (the capacity to act on one's own understanding and choice) is therefore whose key makes the signature. Which of the three balances does consolidation tip? It tips the second pair most directly, since what each person once decided locally is now decided in one place for everyone. And it does so while looking like the cooperation of the third pair.

Suppose a car will start only after a company's computer has answered yes. The owner's decision to drive no longer settles whether the car starts; the answer from that computer settles it. In effect, causal process (what happens because earlier events force it) has become stronger than choice process (what happens because a person decides it). This inverts the principle that choice must remain the primary driver. The principle, grounded in *An Immanent Metaphysics*, is that in any system that includes people, choice, and not mere cause, must remain what drives the system. The reason can be said in this book's own terms. A system always serves somebody, and a system that the people in it cannot steer by choosing serves whoever set its causes going, which leaves those people serving that party and not themselves. What is needed is to restore choice to the system in a more distributed, more local way, and to break the binds that prevent that from happening because of the formation of large consolidated blocks.

III. The Infrastructure of Lockdown

How is this consolidation showing itself in the infrastructure of communication? Consider what happens when a single entity processes eighty percent of all email traffic. That eighty percent is a figure to think with. The measured share of the largest provider is already large: Gmail alone has more than 1.8 billion users (DemandSage, 2026), about two of every five of the world's 4.6 billion users of email in 2025 (The Radicati Group, 2024). Such an entity, using its own algorithms, can unilaterally decide which communications are delivered and which are not. Email, which once was a reliable point-to-point transport, is no longer reliable. The largest providers already set the terms of delivery: from February 2024, Gmail has required any sender of more than five thousand messages a day to its users to carry published records and signatures showing that the mail comes from the domain it names, and Yahoo set the same requirement (Proofpoint, 2024).

The alternative is applications, and they all depend on market penetration and operate through intermediate channels subject to the same dominance dynamics. To communicate reliably, one now needs more than a dozen applications, none of which offer the features that email once provided: editing, threaded replies, preservation of context, automated management.

The next step (running a private website as a personal communications channel) runs into the requirement for certificate-based encryption. To obtain a certificate that other people's browsers will trust, a server's owner must prove control of its name, through the public naming system or through a server reachable from the public internet, and must prove it again at every renewal. And every certificate that browsers trust is written into public logs that anyone can search, under a system called Certificate Transparency, which Chrome has required for every certificate issued since April 30, 2018, and Apple's platforms since October 2018. So the name of every site with a trusted certificate is published the moment its certificate is issued. The longest lifetime allowed for a certificate is 200

days from March 15, 2026, will be 100 days from March 15, 2027, and will be 47 days from March 15, 2029. The shorter the life of a certificate, the more often its owner must go back to the authority and prove control of the name again. A website that its owner means to keep private can no longer be kept private. And because all streaming protocols depend on the same certificate infrastructure, there is no way to set up a communications channel that is simultaneously encrypted, trusted by people who have installed nothing special, and private. When friends install by hand a certificate that the owner of a channel made, the channel is encrypted and private, since that certificate need never appear in the public logs, and the channel is trusted by exactly the people who installed the certificate.

A worked case shows the decision that the certificate infrastructure forces. A reader who never writes code can still follow it, since the point to watch is the moment at which a private design meets the certificates. Vox is a public voice messaging service: a person records a message, and others can hear it even while it is still being recorded. It works well for what it does, but it has no real API (an application programming interface, through which other programs can use a service), so its audio cannot be downloaded and put to other uses; that is reason enough to build a private replacement. Each piece of a private replacement for Vox is simple to build. Code in the browser takes audio from the device, shows its Fourier transform on the screen (the strength of each frequency in the sound), and turns the audio into a WAV file (a file that holds the sound uncompressed). Further code moves the file from the browser to the server, cut into pieces, because WAV files are large. On the server, an encoder turns each WAV file into an MP3 file (a compressed file of the same sound), and a login on the web server lets a person create an account, download the MP3 files, and play them in a player in the browser. The recorder and the MP3 encoder can run in the browser instead, with each recording limited to fifteen minutes, and the server then has no encoding to do. What Vox does beyond this is stream: it compresses the audio in the browser and sends it, chunk by chunk, to the server and on to the listeners as it goes, so that a listener can hear a message while it is still being sent. A private re-

placement can do the same, but only by compressing the audio in chunks (a format that can be cut at any point, or an MP3 file analyzed chunk by chunk), with a worker thread in the browser (code that runs beside the page without stopping it) sending each chunk to the server as it is made, and the server passing each chunk on to the other listeners at once over a WebSocket (a connection that stays open, so that the server can send data whenever it has some). WebSockets work, but they depend directly on the certificate infrastructure. They load only on relatively new devices, and most devices will not open a WebSocket unless the connection is secured, which requires the server, the router and every intermediate part of the internet to meet a certain standard for certificates. So a streaming design must manage certificates, a certificate of the kind a browser trusts with nothing installed requires a public server, and a public server requires a public IP address (the number that identifies a machine on the internet); the channel is then no longer private. Friends who install a certificate by hand, as above, need no public server, and then only those friends can listen. Polling (asking the server again and again whether anything new has arrived) avoids WebSockets, but it does not work well either. The other design sends each message as a batch: the speaker records the whole message in the browser, decides to send it, and sends it to the server in one piece, and the rest of the design works as described. The service is then a voice messaging platform, more like text messages than like a call. The choice turns on one question: how often does anyone listen to a message while it is still being sent? A group that uses the service as its main means of talking, and wants a private version equal to Vox in every respect, must take on WebSockets and the certificates they bring; a group that does not need streaming can send its messages in batches, without WebSockets. The batch design leaves behind what WebSockets bring. It is still a website, so the limit above still holds for it: it can be encrypted and private with a certificate that friends install by hand, or encrypted and trusted by everyone with a certificate that goes into the public logs.

The newest hardware has been locked down further: private applications can no longer be loaded onto the newest devices without the platform owner's permission. The same lockdown now holds for Android devices as well as for Apple's devices. From September 30, 2026, Google requires developer verification for apps installed on certified Android devices in Brazil, Indonesia, Singapore and Thailand. A global rollout of the requirement is expected in 2027. Two paths stay open on those devices: an advanced install path, and installation over a debugging cable (the cable by which a developer loads software from a computer). The platform owner itself provides both paths, so what runs there by either path still rests on what that owner allows. The more this proceeds, the less possible it becomes for any individual to restore their version of choice by running their own applications, their own devices, their own servers, their own access points, their own wireless protocols.

The same pattern shows wherever a platform stands between people: an app store that has required apps to sell through its own payment system and has taken up to thirty percent of each sale, a service people depended on that its owner simply ends, a cloud account closed with no appeal.

The same gate stands in front of the developer. A reader who never writes an app still stands behind that gate, because every app on a phone has come through it, and what a developer cannot do for a user no user can get. To write an app for the iPhone, a developer must join Apple's developer program, which costs $99 a year, and load Apple's development tools onto the developer's own desktop machine, so that a Trojan horse (hidden code placed inside a system) in those tools would reach the very machine on which the code is written. Before the app can be installed on a phone, Apple must vet it; it may reject an app that competes with its own, and when it accepts an app, it takes up to thirty percent of each sale; and Apple has access to the keys that sign the app. So a developer cannot write a secure app for the iPhone. Writing for Android instead means a completely different development process, with all the problems of that

platform. A third-party service that writes one app and makes versions of it for both platforms solves none of the security problems of either, and still brings every problem of making an app in the first place.

What can stand in place of the app stores? The alternative is to write for a known system: a server whose owner holds the complete stack of its hardware (every layer of it, from the chips up), as the owner of a rooted phone (one with every lock removed) holds that phone, with complete documentation of every part of the device, so that nothing has to be guessed. Such a system has no entanglements with anyone else, and what is written for it works for the iPhone and for Android alike, since a phone reaches such a server through its web browser, which is already on the phone and asks no one's permission for each new page. Such a server still has to settle its certificate, and the choice is the one shown above: with a certificate that friends install by hand, it stays free of every other party; with a certificate that anyone's browser trusts, an authority and the public logs come back in. In either choice, no app store stands between the phone and the server, and no platform owner vets what the server sends. Once the internet is treated as hostile, a server like this becomes the primary support for computing, because a laptop runs code from so many authors that its owner cannot know all that it does, and so it cannot be secured against certain kinds of attacks, such as a Trojan horse placed in the tools it carries; the server of this paragraph holds only what its owner has documented. Such a server can run on a small public Linux distribution (a packaged version of the Linux operating system), Tiny Core Linux; where security must be high, it runs on an operating system of its owner's own. Small matters here because of the limit of Chapter 5: the less there is to the system, the more of it its owner can predict.

IV. Compelled Markets and the Gate

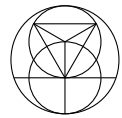
Who gains when every path runs through one gate? How does this connect to the broader dynamics of market creation and extraction (taking payment for passage that people were once free to make)? Anywhere a marketplace can be compelled into existence, an opportunity for extraction follows. Transaction fees, toll gates on communication, platform dependencies: each is an instance of the same structural pattern as a wall built across a city with a gate in the middle. Before the wall, no one had to buy a way across the city. After the wall, everyone who must cross pays whatever the owner of the gate asks.

Cryptocurrency, while offering certain security advantages, does not solve the problem. Early entrants hold the lion's share of resources. The reward for mining a block (a block here is one batch of transactions in the blockchain of Chapter 1, and is a different thing from the consolidated blocks of Section II) halves about every four years, and as it shrinks toward zero, miners are paid more and more by transaction fees. So whoever chooses which transactions enter a block holds more and more of what mining pays. Block building is already concentrated in a few large pools (groups of miners who share their work and its rewards; the largest, Foundry USA, found about 27 percent of blocks in the month to June 19, 2026), and those pools choose which transactions enter each block, and so set the price of entry. The consolidation dynamic repeats. An independent actor who wants a say in what goes into a block now has to come in through a pool, and the choice is the pool's, so the path through which independent actors could enter block building has been removed.

What is being witnessed is an inexorable consolidation that is not cooperation but the squeezing out of every independent path through which choice can enter. Technology, in its current trajectory, is compelling causation over choice, and this is the fundamental issue.

V. Summary

In this chapter I have shown three balances that sustain complex systems, and how technology departs from the three balances. Technology departs from the balances by consolidation, which compels alignment without choice, in email, in certificates, in the platforms that decide what software may run, and in the pools that build the blocks of a cryptocurrency. The consolidation goes on whatever any one person does. What a person can do is keep their own paths open inside it, at some cost in convenience, by their own choice. Consolidation closes an independent path by setting across it a consolidated block someone else controls: a platform's key, a provider's filter, a pool's choice. The paths a person keeps open are those that pass through nothing but what the person holds. What would such a path look like, one that passes through nothing but what a person holds?



Chapter 7

Computing, Cryptography, and the Persistence of Agency

> What is all of this for, and what does a system look like that keeps choice with the person? This chapter answers with paper-based computing, a symmetric key cryptography protocol, and the philosophical significance of personal agency through code, and it closes with a meditation on what all the preceding technical material is for: the preservation of human agency through resilient, transparent, and personally controllable technology. In Chapter 6 I showed how the paths by which choice enters a system are being closed. In this chapter I show small, clear systems that keep the paths open.

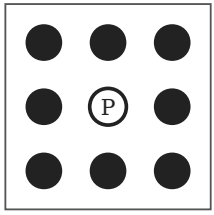
I. Nine-Point Computing

So what can a person actually do, and what was all of this for? Chapter 6 ended by asking what a path would look like that passes through nothing but what a person holds. Begin with the plainest case there can be: a machine in which nothing is hidden, because every character it reads or writes is a pattern of holes in a sheet of paper that you can hold up to the light. You need not build this machine to learn from it. It shows, in the

simplest form, what a machine that keeps choice with the person has to be like. Why paper? Paper can be made from the most basic materials, and what is written on it can be seen.

Consider a computing device based upon paper for storage of ROM code (code in memory that is only read), with the minimum amount of RAM (working memory) and the minimum number of transistors (the tiny electronic switches a computer is built from) involved in the actual computer itself. The paper itself is also used as an output device. All input and output is in the form of nine code.

Nine code consists of a three-by-three array of dots. The center dot is treated as a parity bit (an extra bit that makes the count of ones even, or odd, so that a single wrong bit shows). The remaining dots are treated as eight bits, enough for any standard ASCII code (ASCII is the standard code that gives each letter and sign a number; it uses seven bits). People can learn to read and write in nine code, as regular recognized patterns following the ASCII layout.



The nine-code cell: a three-by-three array of dot positions. The eight outer positions carry the eight bits of a character; the center position, marked P, is the parity bit.

The paper medium itself is dark, unbleached paper, since it is easier to make dark paper than white paper, not requiring bleaching of raw materials. The dark paper is placed on a light table, and the nine code is read through holes created by a laser. The nine code is read by an optical system moving on a carriage controlled by stepper motors (motors that turn in fixed small steps, so that they can place a carriage exactly). The carriage

moves the optical system to whichever character needs to be read, using nine photosensors in a three-by-three grid. New sets of holes can be made by lasers positioned in the same locations as the photosensors.

The computing itself is done by a small circuit built from as few transistors as will do the work: it reads its program, the ROM code, from the paper, keeps its working values in the small RAM, and writes its results back to the paper as new nine code.

The machine suits work that needs few characters and much trust: keeping records, making signatures, and running the protocol of Section II. Its speed is set by its carriage: the optical system must stop over each character to read or write it, so the machine works at the pace of its stepper motors, far below the speed of a conventional computer.

This design represents maximum resilience. There is no dependence on complex supply chains. Its few parts are generic parts that many makers produce, that need no firmware and no one's permission, and that can be stocked as spares for decades. The computing medium, paper, can be manufactured from the most basic materials. The optical reading system requires only photosensors and stepper motors. There are no microprocessors to become obsolete, no firmware to be locked down, no operating system to be compromised. The machine is transparent in the deepest sense: every character it reads or writes is visible in the physical medium itself.

This connects directly to the infrastructure resilience themes of Chapter 4. A grid failure that renders all conventional computers inoperable leaves a paper-based computing system untouched. Conventional computers depend on networks, remote services, and a supply of parts and updates that a long failure stops; the paper machine depends on none of these. The only power needed is sufficient to drive a small laser, a handful of photosensors, the stepper motors, the light table and the small computing circuit: power that a small solar panel or hand-cranked generator can provide. In a world where infrastructure can fail catastrophically, the most resilient computing platform is the one with the fewest dependen-

cies. The companion book *The Tools You Own; Building Infrastructure That Outlives Its Maker* works out the same aim for everyday software and files: infrastructure a person owns, built to outlive its maker.

II. Symmetric Key Protocol

How can two people talk in private, using only what the two of them hold? In plain terms: the two sides share a secret, mix it with a fresh number for each message to make a key used once, and then replace the secret with its own hash, so that nothing earlier can be recovered from it. Picture that last step with the meat grinder of Chapter 1. Two friends each hold a copy of one secret lump. After every letter, each of them puts the lump through the same grinder, keeps what comes out as the new secret, and burns the old lump. Both now hold the same new secret, and what has been ground cannot be turned back into the lump it came from. So a thief who takes today's secret learns nothing of yesterday's.

The symmetric key encrypted communications path consists of three aspects. First, an open name or label for the key, which is always transmitted in the channel. Second, a shared secret known only to the client (the machine that makes requests of a server) and the server, which is never transmitted. Third, a current hard random nonce (a number used once, random in a way no one can predict, as from a physical source and not from a formula), which is only transmitted from the server to the client, on every completed message pushed from the client to the server.

The pattern is started whenever the client makes a request for the current value of the server-side nonce. The server allocates a new nonce on every such request. For every post from client to server, the client takes the current nonce, concatenates it with the shared secret (joins the two end to end), hashes the combined string, so as to generate the one-time-use symmetric encryption key. Hashing the same string with two different fixed labels (a fixed label, here, is a short agreed string added before the hashing, and is a different thing from the key's open name) gives two

keys: one to encrypt, and one to make the keyed hash (a check value that only a holder of the key can make). Hashing with two different fixed labels is the usual way to draw two keys from one secret, and the Internet standard RFC 5869 sets out that method. The client encodes the message using the one-time-use encryption key and transmits it along with a header indicating which key label (the open name of the key) was used. The server decrypts, verifies (it checks a keyed hash of the message, made with the second one-time key and sent with it, which shows that the message came from a holder of the shared secret and was not changed on the way), performs the required actions, generates a new nonce for that key label, and replies with the next nonce. Each reply carries a keyed hash made with the second one-time key the server has just used, and the client moves its secret forward only on a reply that verifies.

After each completed exchange, the client and the server each replace the shared secret with the hash of the old shared secret (this step is called a ratchet, because it turns only one way), and erase the old one. The server makes the replacement when it has decrypted and verified a message; the client makes it when it receives the next nonce. Because the hash cannot be run backward, a shared secret released today cannot be used to compute any earlier shared secret, or any key made from one.

If the server's reply is lost, the server has replaced its secret and the client has not. The server is never more than one step ahead, since it replaces the secret only after decrypting a message, and the client replaces it only after hearing the reply. So the client asks for a fresh nonce and sends again; if the server cannot verify the message, the client replaces its own secret with its hash, as the server already has, and sends once more. The hash can always be run forward, so the client can always catch up, and the server never needs to keep an old secret.

This protocol achieves forward security through the continuous generation of fresh symmetric keys and the continuous replacement of the shared secret. Each message uses a key that exists only for the duration of that single exchange. Even if an adversary captures the encrypted message and the nonce, they cannot reconstruct the key without the shared

secret. And because each key is used only once, the theft of the one-time key of any single exchange reveals nothing about any other exchange. Forward security protects the past: whoever steals today's shared secret can compute every later secret from it, and so read later messages, until a new shared secret is set up by another channel. That channel is one the attacker does not reach: the two people meeting in person, or a secret written on paper and carried by hand; a sheet of nine code (Section I) is such a channel.

The protocol answers the problem Chapter 1 found with a symmetric key that a group keeps using, and it gives forward security with hashes alone. It uses no public key mathematics, so the method by which a large quantum computer would break today's public key methods (Chapter 2) gives no way into its recorded messages.

III. Code Clarity and the Transparency of Systems

What do a machine made of paper and a protocol made of hashes have in common? The nine-point computing device and the symmetric key protocol share a common principle: the system must be comprehensible to the person who uses it. The machine reads characters burned in paper. The protocol generates keys from components that each party understands. Neither system hides its operation behind layers of abstraction that the user cannot inspect.

This connects to the transparency-opacity balance discussed in Chapter 6. The balance of Chapter 6 concerns how visible people and their affairs are to others; the transparency meant here is how visible a tool's workings are to the person who uses it, and that kind keeps the balance by keeping the person, not the operator (the party that runs a system for its users), in control. Opacity in computing systems serves consolidation. The more opaque the system, the more dependent the user becomes upon

the system's operator. Transparency in computing systems serves choice. The more transparent the system, the more capacity the individual retains to understand, modify, and control their own computational environment.

Whose agency does a program serve? Software, like any machine, is an encoding of agency. Whoever writes a program makes every menu item, every function, every capability it has serve somebody, and that is the whole reason open source (software whose code anyone may read, change and share) matters. Code that anyone may read but only its owner may change still serves its owner. A design shows the intention of its designer, so there is no such thing as agency-neutral design. Clear design intentions in a text make for clear design intentions in the code, and clear design intentions in the code let a community know whether those intentions are its own: whether the policy the code carries out is consistent with the community's values, what policy the code actually carries out, and whether those who wrote it did what they said they would do.

Who does it serve? In the old stories of the Grail, the king is wounded, his land has gone to waste, and healing waits on a visitor who asks whom the Grail serves. Whom something serves is the question of the Grail quest: does the king serve himself, or does he serve the land and the people? The same question can be answered for code, for a machine and for a policy, when each of them can be inspected.

Think of a vault, whose walls no one can see through, and whose contents are safe because no one can get at them. Then think of a market square in which every stall stands open and every sale is made in front of the neighbors. No one there needs a lock or a code to know that the scales are true, because anyone can walk up and look. Security can be created in two ways, the way of the vault and the way of the market square. One is cryptography, which is perfected opacity. The other is perfected transparency, a way the world has largely forgotten. Trust needs transparency, and cryptography cannot give transparency, because cryptography is perfected opacity. What cryptography hides is the message. The transparency that trust needs is the kind this section began with: how visible a tool's workings are to the person who uses it. So the protocol of Section II hides every

message it carries, and each party can see all of how it works. Uniqueness, the foundation that Chapter 2 named, stands under both ways. Cryptography keeps a key to one holder, and transparency lets anyone tell a true thing from a false copy by looking. Under perfected transparency, the Trojan horse is made of glass: anyone can see through the whole of it and tell that there are no soldiers inside. A city that cannot see into the horse does not let down its walls and drag the horse in after it, because once the horse is inside, the interests that built it, corporations and platforms alike, come out and trample everything.

Code clarity is an ethical requirement, and more than an aesthetic preference. Clarity makes the system inspectable. Inspectability makes it trustworthy. Trustworthiness makes it an instrument of agency rather than an instrument of control.

IV. The Persistence of Agency Through Code

Why go to all this trouble over code? Think of a grandmother's recipe, written out so clearly that her grandchildren make the dish as she meant it long after she is gone. What she chose to do goes on being done, with nothing standing between her choice and the meal. The reason for wanting all of this support and infrastructure in code is that it represents the largest possible persistence of one's own personal agency, in the least obstructed and most pure way, for the longest duration.

A machine has only the agency its makers put into it. An architect who builds toasters, and whose toasters make toast for him, has invested his agency in the design, and the agency encoded in the design of the toaster benefits him by making toast. A person who acts for another may serve himself instead (the principal-agent problem), but a machine has no agency of its own, unless someone makes the colossal mistake of making the machine intelligent. Short of that, a machine serves whoever invested

agency in it: its owner, or a large corporation that says the machine is for its users while it serves the corporation. Reading its code is the way to tell which.

Perfected clarity in the code lets a community trust it and invest resources in it, knowing that the land and the people are well stewarded, because the code has no agency other than that of the community itself, taken as a whole. Governance, as a process, then protects the land and the people and helps them to thrive, because the agency in it is their own.

Clear code supports clear people and clear trust. A community whose code is clear can see what its tools do and whom they serve (Section III), and does not have to live under tools it cannot read. People whose shared code is clear are healthier psychologically, socially and environmentally, and have more balance, so they can talk together intelligently, evaluate things in a real way, and make sense of things around the dinner table. Some of what they say is written down as text; the clearest and most beautiful parts of that text, and the questions that matter most, become the guidance for writing new code; and the new code brings more clarity to the community, which brings clearer text, and so on, around and around. The cycle runs in the direction of Axiom II of *An Immanent Metaphysics* (the system that Chapter 2 introduced; an axiom is one of its starting statements), and that direction is good. The direction runs from what people say together, to what they write, to what they build, and back to the people. That direction is good because what gets built begins from what people have said together and comes back to them, so the agency in what is built stays their own; Section III showed that whose agency a system carries decides whom that system serves. This cycle is the ethics of code, and it is the solution. The problem it solves is the one Chapters 6 and 7 have been asking about: how people can keep the machines they depend on serving them.

Software should be as complicated as it needs to be, and no more. Clarity and regularity of code is the most essential aspect. The companion book *Clarity Dissolves Complexity; The Foundations of Computer Science, from the Single Instruction to the Planet* develops this thesis at length, and asks of every system one question: is its complexity managed through clarity?

V. The Trusted Machine and the Sovereign Development Machine

What would it take for a person to know everything their own machine does? The limit of Chapter 5 holds between a person and the tools the person uses: the systems that leave control with the person are the ones simple enough for the person to predict, inspectable and small. The machine a person acts through is the first place that limit applies.

An operating system is needed that is simple enough for its operator (here, the person who runs it and acts through it) to understand completely. Such a system can be that simple because it does little itself: the work that needs complex protocols is done by separate proxy machines, separate machines that deal with the network on the trusted machine's behalf, and the trusted machine keeps only what its operator needs. What one gives up is speed and convenience; for this purpose, a machine that is slower but that its operator can understand is the better machine. Its defining principle is that the operator knows everything the machine does: there should never be any question as to whether the machine is doing something that the operator did not program. In modern computing systems, there are so many different code authors, each with intentions to serve interests other than the user's own, that it is impossible to know what is actually going on. The technology stack (the layers of hardware and software that a program rests on) must be flatter. Inter-machine linkages are more important to manage than internal machine complexity, because every line to another machine is a way for something to arrive that the operator did not ask for.

The basis of secure computing requires that all communication with the outside world be synchronous and serial, with time strictly user-controlled. Synchronous communication means that data arrives only when the operator has asked for it; serial communication means one stream at a time, so that the operator watches a single stream rather than several at once; and time strictly user-controlled means that the machine never acts on a clock of its own. There should not be any asynchronous receipt of information that is not immediately logged, so that nothing reaches the machine unseen. The networking interface must be purely serial at the lowest possible common denominator at the hardware level, so that the hardware itself offers no second path around the one line the operator watches. Integration with machines supporting complex protocols must be isolated from the machine acting as a proxy for the agency of a human.

The trusted machine keeps Ashby's law of requisite variety, the law of Chapter 5 seen from the regulator's side. It meets the world through one serial, synchronous, logged line, so the disturbances that reach it are few, and a machine with few responses can meet them. The proxy machines carry the large variety the network demands. A message reaches the trusted machine only when the operator asks the proxy for it, over the one serial line, and the line's log shows every byte. The trusted computing base (the machine that acts on behalf of a human) must be isolated from the proxy machines that interface with the untrusted network. A system one cannot understand is a system one cannot trust, and a person can answer for what a machine does on their behalf only if they can understand it, so the clarity of the trusted machine is an ethical requirement as well as a technical one.

The same principle governs the machine on which code is written. A reader who never writes code still depends on that machine, because whatever got into it is in every program that came from it. Think about security from the point of view of intentionality, what the author of the code means it to do. A person writes software expressing their intention, and we want to be sure that what they intended is what actually happens. There are three threats. Exfiltration: someone steals data, such as a pass-

word typed into a bank or the key to an encrypted message. Infiltration: someone inserts a Trojan horse (hidden code placed inside a system) to make the system do their bidding rather than the author's. Modification: even though the author wrote code expressing their intention, someone changed it, so what actually happens is not what the author wanted.

The three threats apply first to the machine the code is written on. On a sovereign development machine (a machine for writing code that answers only to its owner) everything is organized under a single root directory, dvol (the top folder of the machine's tree of folders): whatever is under dvol is managed, and whatever is outside it is not. The machine's owner does the managing, and because the owner knows what belongs in that one tree, a change to it can be found: that is how the machine meets modification. There is a temporary directory for internet material; downloads are reviewed and either kept or discarded. That review is how the machine meets infiltration: nothing from the internet enters the managed tree until its owner has looked at it. Under dvol are executables (programs ready to run), device drivers (the programs that operate the machine's hardware), and working project directories. The machine is configured to treat its wireless hardware as a hotspot: a local server that nearby devices can connect to. It can also connect as a client to external networks, but external connections are selective, because any external connection introduces a risk to the data on the machine: choosing its connections is how the machine meets exfiltration. The computer is a sovereign local access point. Nearby devices then connect to something its owner holds, with no outside party between them, and an access point of one's own is one of the paths Chapter 6 showed closing.

The trusted machine is the principle, and the sovereign development machine is one case of it. Both pass the test of Chapter 5: the person who relies on the machine can predict what it does.

VI. Summary

All the cryptographic primitives of Chapter 1, the uniqueness guarantees of Chapter 2, the defense-in-depth architecture of Chapter 3, the infrastructure security of Chapter 4, the control-theoretic limits of Chapter 5 (a person can keep control only of what the person can predict), and the critique of consolidation in Chapter 6: all of these converge upon a single point.

Technology can serve human agency, or technology can subvert human agency. The difference depends upon whether the systems we build are transparent, inspectable, personally controllable, and resilient against the failures and consolidations that the preceding chapters have described.

Code that is clear, regular, and minimal is code that an individual can understand, maintain, and trust. Such code is the instrument through which personal agency persists: through which an individual retains the capacity to act in the world according to their own understanding and choice, rather than according to the dictates of systems they cannot inspect or powers they cannot resist.

For one person, this means a few plain practices: run software you can inspect, hold your own keys, and keep at least one channel of communication that does not depend on one company. Each costs some convenience, and each keeps open a path by which choice enters, of the kind Chapter 6 showed being closed. Each of the three keeps in the person's own hands something that would otherwise let another party decide for them: what their software does, which signatures are trusted on their behalf, and whether their messages arrive.

What does each of the three look like for a person who does not write code? Think of a restaurant whose kitchen can be seen from the tables. You need not be a cook to gain from that, since the cooks among your friends can look on your behalf. To run software you can inspect is to choose the program whose code anyone may read, so that you, or people you trust who can read it, can see what it does, as a community can (Sec-

tion III). Think of the key to your house, in your own pocket or left at a desk in someone else's building. To hold your own keys is to keep the keys that make your signatures and open your messages in your own keeping, and not with a company that can use them or refuse you. Suppose the one service that carries your family's messages closes your account tomorrow. To keep at least one channel of communication that does not depend on one company is to have agreed beforehand on a second way to reach those people.

Security is ultimately not for the protection of data in the abstract, but for the preservation of the human capacity to choose, to act, and to understand: the persistence of agency in a world of increasing complexity.

Appendix. The Secure Query System in Full

Chapter 3 says what the secure query system is for, gives its seven layers in words, and says what it leaves in the hands of the sample provider. This appendix gives the specification of that system in full.

I. Project Objective

The design described here is a system that answers one question about private data without ever showing the data: are samples that meet a researcher's criteria available? It gives researchers that answer conveniently and securely, through a web page made for the purpose. Researchers working on grant proposals require a means of determining whether the samples needed to implement an experimental protocol are currently available. The system helps them decide by providing a specialized web interface to which authorized researchers can submit availability queries.

However, the nature of the information demands the strongest possible security measures. Data that would be damaging to the privacy of sample providers must not be made available either directly or indirectly. The system provides information only about sample *availability*, never about the samples themselves. There must be an effective guarantee that no method of unauthorized access could succeed. The design meets this requirement by layers: no success at any one layer reaches the protected information, and even a total compromise reveals only whether samples are available. It can reveal no more because the query system holds no more: the data extracted from the source databases is only what availability answers need (Section III).

II. Air-Gapped Source Databases

The source database host machines are not connected via any form of network to any other machines. Data transfers between source hosts and intermediate hosts are implemented exclusively via hand-carried physical media. This ensures that the source databases are completely protected from any possible form of network attack. The media are the one path into a source host, so they carry data out only: each medium is new when it is written on a source host, and no medium that has left a source host ever returns to one. The rule is needed because hand-carried media are the path by which malware has crossed air gaps before: Eugene Kaspersky reported that Stuxnet (malware built to damage industrial control systems) reached the air-gapped network of a Russian nuclear plant on a USB device (Pauli, 2013).

III. Data Extraction and Encryption

On each source database host, a methodology extracts only the minimal information needed for the query system to resolve whether matching samples are available. The extracted data is filtered according to institutional sharing policies, normalized (put into one standard form) by the project software, and converted into an encrypted format before being placed on physical media.

The extraction, filtering, normalizing, encrypting, and preparation for transport are made as simple and automatic as possible. Personnel responsible for maintaining the data can carry out the required transfers with minimal special training.

IV. Intermediate Hosts and Private Network

An intermediate host is provided at each source database location to minimize the physical distance that hand-carried media must travel. These intermediate hosts are connected via a secure communications link to the query processing server. The communications channel between intermediate hosts and the query processing server also employs encryption.

V. Separated Query and Security Servers

The query processing server accepts specially formatted queries from a physically separate security server. The query processing system checks queries against abstracted data in the intermediate hosts, decrypting database fields as necessary, and returns only an availability answer, never any data about the samples themselves.

The security server handles authentication. Researchers access the system from approved physical hosts equipped with biometric devices. The login process requires a username and password followed by biometric verification. The security server also establishes a positive locked synchronization state (a shared state that client and server check on every message, so that a forged packet is refused) with the client host, which is the researcher's computer, as an additional security measure.

VI. Seven Layers of Security

Around all seven layers stands the air gap of Section II, which is not counted among them. The security architecture employs seven layers. First, the software is strongly resistant to all known forms of unauthorized login, requiring both password and biometric verification. Second, all network communications use encryption, with resistance to stream injection at-

tacks (inserting forged packets into a live connection) through positive state synchronization. Third, user login is accepted only from registered host machines with matching network address patterns.

Fourth, information on intermediate hosts is stored in encrypted format, and all physical transfers use encryption. Fifth, the host machine is secured according to best practices for network-connected systems. Sixth, the host is isolated so it cannot serve as a platform for unauthorized access to any other institutional systems. Seventh, even in the event of total system compromise, the information selected for publication is chosen to preserve the privacy of sample providers. This holds for a series of questions as well as for one: the answers are chosen so that no series of availability answers, each of them true, narrows down to a single sample provider.

The rule that gives this has two parts. The query system answers "available" only when at least a set number of sample providers match the criteria; and it keeps a record of each researcher's questions, and refuses any question that, set beside an earlier one, would narrow the matching providers to fewer than that number. A rule is needed because enough accurate answers to counting questions can be combined to rebuild what each record holds (Dinur and Nissim, 2003), and a question of availability is a counting question with the count reduced to yes or no.

The table below sets the seven layers side by side, so that the independence of each can be checked at once.

Layer	What it stops	What it relies on	What an attacker who has beaten it still faces
1. Login by password and biometric verification	Unauthorized login	The security server and the biometric devices on approved hosts	The registered-host check (layer 3), the encrypted data (layer 4) and the availability-only answers (layer 7)

Layer	What it stops	What it relies on	What an attacker who has beaten it still faces
2. Encrypted network communications with positive state synchronization	Reading data in transit, and inserting forged packets into a live connection	The keys, and the shared state that client and server check on every message	The login checks (layers 1 and 3), the encrypted stored data (layer 4) and the availability-only answers (layer 7)
3. Login only from registered hosts with matching network address patterns	Login from any other machine, even with stolen credentials	The register of approved hosts kept by the security server	The password and biometric login (layer 1), the encrypted data (layer 4) and the availability-only answers (layer 7)
4. Encrypted storage on intermediate hosts and encrypted physical transfers	Reading data from a stolen disk, a stolen medium or an intercepted transfer	The keys	The air gap around the source databases and the availability-only answers (layer 7)
5. Host secured according to best practices for network-connected systems	Ordinary attacks on the host's operating system and services	The host's configuration and its upkeep	The encrypted data (layer 4), the isolation of the host (layer 6) and the availability-only answers (layer 7)
6. Host isolated from other institutional systems	Use of the host as a platform for reaching other institutional systems	The network separation around the host	Inside the design, the availability-only answers (layer 7); outside it, the other institutional systems stay out of reach

Layer	What it stops	What it relies on	What an attacker who has beaten it still faces
7. Only availability is published, chosen to preserve the privacy of sample providers	Learning anything about samples or their providers, even after total compromise	The choice of what the query system holds and what it answers	Nothing more is needed: even a total compromise reveals only whether samples are available, and the source databases stay behind the air gap

The table also shows which layers share a dependence: the first and third rely on the security server, and the second and fourth rely on the keys. The air gap and the seventh layer rely on neither, and so they hold even when both fail. The air gap and the seventh layer are the two guards that keep the system secure even if multiple layers are breached. Whatever else falls, the source databases stay behind the air gap, and the query system holds and answers nothing but availability.

Terms

accuracy: for control, as the word is used in this book, landing in the safe region of outcomes at all; the volume of the safe region decides the accuracy needed (Chapter 5). In measurement, accuracy is closeness to the target.

air gap, air-gapped: a computer with no network connection of any kind; data reaches it only on media carried by hand (Chapter 3).

algorithmic complexity: the length of the shortest description that produces a thing.

alkaloids: the family of plant compounds that includes caffeine and theobromine.

API: an application programming interface, through which other programs can use a service.

ASCII: the standard code that gives each letter and sign a number; it uses seven bits.

asymmetry: in *An Immanent Metaphysics*, a difference of content across a difference of context.

bearer certificate: anything that grants access to whoever holds it, such as a physical key, a token or an embedded firmware chip.

Bitcoin: the first cryptocurrency, launched in 2009.

blockchain: a chain of tables of transactions, each holding the hash of the table before it, so that a change anywhere breaks every later hash.

bootstrapping: using the slow public key method only to set up a symmetric key, which then carries the bulk of the data.

cascade failure: a failure that causes further failures, as one tripped line overloads the next.

Certificate Transparency: the system under which every certificate that browsers trust is written into public logs that anyone can search (RFC 6962); Chrome has required it for certificates issued since April 30, 2018.

ciphertext: a message after encryption.

code signing: a rule under which a device runs only software signed by a key the platform owner approves.

consolidation: alignment with a large block that is compelled, without any choice or permission on the part of the individual; the book sets it against cooperation, in which each person agrees to the shared rules by choice.

computational irreducibility: Stephen Wolfram's name for the property of many physical processes that no shortcut predicts their outcome faster than running the process itself.

continuity: in *An Immanent Metaphysics*, a sameness of content across a sameness of context.

control theory: the study of how a controller keeps a system near a goal.

controllability: how much of a system's state a controller can change.

control-limit argument: the name, in the books on the safety of artificial general intelligence, for the result that beyond a threshold in the difference between the levels of abstraction at which a controller and a system operate, no control strategy does better than chance.

controller, system, living things downstream: in Chapter 5, the controller is any controller, the system is the system it controls, and the living things downstream are those downstream of that system.

coronal mass ejection: a cloud of charged gas thrown from the sun; when it strikes the earth's magnetic field it causes a geomagnetic storm.

CRC: a cyclic redundancy check, a check value computed over a packet of data that shows whether the packet arrived whole.

defense in depth: a design of independent layers of protection, so that no single failure exposes what the design protects; independent here means that each layer stops an attack the others do not stop.

DC offsets, AC offsets: the steady currents a solar storm drives into power lines, on top of their normal alternating current.

device attestation: a rule under which a service refuses a device that cannot prove, by a signature, that it runs unmodified software.

digital signature: a value made with a private key that anyone can check with the matching public key, and that only the holder of the private key can make.

discontinuity: in *An Immanent Metaphysics*, a difference of content across a sameness of context.

double spend: spending the same digital coin more than once.

duals: two properties each of which is the mirror image of the other, as observability and controllability are in many systems.

dvol: the single root directory under which everything on a sovereign development machine is organized; whatever is under it is managed, and whatever is outside it is not.

effective choice: a choice that has effect: one that changes what happens in the world; the symmetry and the continuity of effective choice are the two principles of non-relative ethics (Chapter 4).

electron transport chain: the chain of proteins by which cells make most of their energy.

ephemeral key exchange: making a fresh public and private key pair for one connection and discarding it afterward, so that no long-lived key can later open the session (as in ephemeral Diffie-Hellman key exchange).

error correction: adding redundant bits so that damaged bits can be repaired.

exfiltration, infiltration, modification: the three threats to code (Chapter 7): someone steals data; someone inserts hidden code to make the system do their bidding; someone changes the code, so that what happens is not what its author intended.

forward security: the property that a key released later cannot be used to read earlier messages: "Whatever was secret stays secret".

Fourier transform: the strength of each frequency in a sound or other signal (Chapter 6).

good regulator theorem: the result of Roger Conant and W. Ross Ashby (1970) that every good regulator of a system must be a model of that system.

halting problem: the result that no general procedure can determine, for an arbitrary program, whether that program will finish.

hard random: random in a way no one can predict, as from a physical source and not from a formula.

hash function: a one-way function that grinds a document of any length into a short output that changes completely when any one input bit changes.

HTTPS: the secure form of the web's protocol, shown by the lock in the address bar.

Incommensuration Theorem: a structural fact about the nature of knowing: symmetry and continuity cannot both be applied, simultaneously and fundamentally, to any event or comparison; and neither can asymmetry and discontinuity (*An Immanent Metaphysics*, aphorism 1.43-2).

keyed hash: a hash of a message made with a secret key, which only a holder of the key can make and which any change to the message breaks.

law of requisite variety: W. Ross Ashby's law that the number of distinct outcomes a regulator can hold a system to is at least the number of distinct disturbances divided by the number of distinct responses the regulator has.

MD5: an older hash function, now broken.

miner: a computer that competes to add the next block to a blockchain.

MP3 file: a compressed file of a sound (Chapter 6).

mutual information: what two parties already share that shapes how a message is read; in information theory the term names a quantity, how much one signal tells about another, and here it names the shared context itself.

nine code: a three-by-three array of dots in which the eight outer dots carry the eight bits of a character and the center dot is a parity bit.

no-cloning theorem: the result (Wootters and Zurek, and Dieks, 1982) that an unknown quantum state cannot be copied.

nonce: a number used once.

non-relative ethics: the ethics whose two principles (the symmetry and the continuity of effective choice) hold for every person and situation alike.

observability: how much of a system's state a controller can see.

P versus NP: the question whether every problem whose answer can be checked with work that grows like a fixed power of the size of the problem can also be solved with work that grows that way.

parity bit: an extra bit that makes the count of ones even (or odd), so that a single wrong bit shows.

physically unclonable function: a key that a chip takes from the variation manufacturing leaves in that one chip.

plaintext: a message before encryption.

polling: asking a server again and again whether anything new has arrived (Chapter 6).

positive locked synchronization state: a shared state that client and server check on every message, so that a forged packet is refused.

precision: for control, as the word is used in this book, landing in the safe region every time when that region is scattered; the length of the region's boundary decides the precision needed (Chapter 5). In measurement, precision is how closely repeated results agree.

principal-agent problem: the problem that a person who acts for another may serve himself instead (Chapter 7).

prion: a misfolded protein that makes other proteins misfold.

proof of work: the hard function a miner must compute to add a block, which takes great effort to find and one hash to check.

private key, public key: the two matched parts of a key in public key cryptography: what the public key locks, only the private key opens, and what the private key signs, anyone can check with the public key.

proxy machine: a separate machine that deals with the network on the trusted machine's behalf (Chapter 7).

RAM: working memory.

ransomware: software by which gangs lock a victim's files and sell back the key.

RSA: the public key method of Rivest, Shamir and Adleman (1978), which signs by applying the private key to a hash of the document.

ratchet: replacing a shared secret after each use with the hash of itself, so that later secrets cannot be run back to earlier ones.

rivalrous, non-rivalrous: a thing is rivalrous when what one holds another cannot hold, as with atoms and energy; it is non-rivalrous when many can hold it at once and none takes it from the others, as with a pattern.

ROM: memory that is only read.

rooted phone: a phone with every lock removed, so that its owner holds the whole of it (Chapter 6).

SHA-1: another older hash function, of the United States standard, broken by finding two inputs with the same output.

sinus node: the heart's natural pacemaker.

sovereign development machine: a machine on which code is written, organized under a single root directory, dvol, that serves as its own local access point and makes external connections selectively (Chapter 7).

spoofing: the event in which a bearer certificate goes from being unique to seeming, to whatever measurement or detection equipment, like two identical instances.

stepper motor: a motor that turns in fixed small steps, so that it can place a carriage exactly.

stream injection attack: inserting forged packets into a live connection.

strict avalanche criterion: the standard (Webster and Tavares, 1985) under which each output bit of a function flips with a chance of one half whenever any one input bit changes.

symmetric key: one key that both encrypts and decrypts: plaintext plus key gives ciphertext, and ciphertext plus key gives plaintext.

symmetry: in *An Immanent Metaphysics*, a sameness of content across a difference of context.

TCP/IP: the protocols by which the internet carries packets between machines and puts them back in order (Chapter 5).

TLS: Transport Layer Security, the protocol behind web addresses that begin with HTTPS; its current version, TLS 1.3, dates from 2018.

Trojan horse: hidden code placed inside a system.

trusted computing base: the machine that acts on behalf of a human, isolated from the proxy machines that deal with the untrusted network (Chapter 7).

WAV file: a file that holds a sound uncompressed (Chapter 6).

Web3: the name, coined in 2014, for internet services built on blockchains, with users holding their own keys.

WebSocket: a connection between a browser and a server that stays open, so that the server can send data whenever it has some; most devices open one only over a secured connection.

worker thread: code in a browser that runs beside the page without stopping it (Chapter 6).

Bibliography

Apple. "Apple Developer Program". https://developer.apple.com/programs/. The annual membership of $99 that a developer pays to publish apps for Apple's devices (Chapter 6).

Ashby, W. Ross. *An Introduction to Cybernetics.* London: Chapman & Hall, 1956. The Law of Requisite Variety, foundational to the control-theoretic argument in Chapter 5.

CA/Browser Forum. Ballot SC-081, "Introduce Schedule of Reducing Validity and Data Reuse Periods". 2025. The schedule of maximum certificate lifetimes (200 days from March 15, 2026; 100 days from March 15, 2027; 47 days from March 15, 2029) given in Chapter 6.

Conant, Roger C., and W. Ross Ashby. "Every Good Regulator of a System Must Be a Model of That System". *International Journal of Systems Science* 1, no. 2 (1970): 89-97. The good regulator theorem (Chapter 5).

D-Central. "The State of Bitcoin Mining, H1 2026". https://d-central.tech/reports/state-of-bitcoin-mining-2026-h1/. Difficulty, network hash rate and the share of the largest pool, used in Chapters 1 and 6.

DemandSage. "Gmail Statistics (2026): Number of Accounts and Market Share". October 12, 2026. https://www.demandsage.com/gmail-statistics/. More than 1.8 billion Gmail users (Chapter 6).

Dieks, D. "Communication by EPR Devices". *Physics Letters A* 92, no. 6 (1982): 271-272.

Diffie, Whitfield, and Martin Hellman. "New Directions in Cryptography". *IEEE Transactions on Information Theory* 22, no. 6 (1976): 644-654. Foundational paper on public key cryptography.

Dinur, Irit, and Kobbi Nissim. "Revealing Information While Preserving Privacy". In *Proceedings of the Twenty-Second ACM Symposium on Principles of Database Systems* (*PODS*), 202-210. 2003. Enough accurate answers to counting questions about a database can be combined to rebuild most of what it holds (the Appendix).

Gassend, Blaise, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. "Silicon Physical Random Functions". In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (*CCS '02*), 148-160. Washington, DC, 2002. Measured chip-to-chip variation in circuits of one design (Chapter 2).

Google. "Android Developer Verification". https://developer.android.com/developer-verification. Reported in "Google Sets Sept. 30 Deadline for Android Developer Verification in Four Countries", *The Hacker News*, June 2026, https://thehackernews.com/2026/06/google-sets-sept-30-deadline-for.html.

Google Chrome. "Certificate Transparency Policy". https://google-chrome.github.io/CertificateTransparency/ct_policy.html. Required for certificates issued since April 30, 2018 (Chapter 6).

Kolmogorov, Andrey N. "Three Approaches to the Quantitative Definition of Information". *Problems of Information Transmission* 1, no. 1 (1965): 1-7. Foundational work on algorithmic complexity theory, referenced in the discussion of irreducible algorithms in Chapter 5.

Krawczyk, Hugo, and Pasi Eronen. "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)". RFC 5869. Internet Engineering Task Force, May 2010. Deriving separate keys from one shared secret (Chapter 7).

Landry, Forrest. An Immanent Metaphysics. First published 2001. Wytheville, Virginia: UVSM Publications. ISBN 979-8-9896628-4-5. Cited by aphorism number.

Landry, Forrest. *Clarity Dissolves Complexity; The Foundations of Computer Science, from the Single Instruction to the Planet*. Wytheville, Virginia: UVSM Publications, 2026. ISBN 979-8-90710-230-9. DOI: https://doi.org/10.67291/DPKY9625. Its thesis, that complexity must be managed through clarity (Chapter 7).

Landry, Forrest. *Halo; Drawing Programs in the Normal Form of Code, Data, and Architecture*. Wytheville, Virginia: UVSM Publications, 2026. ISBN 979-8-90710-084-8. DOI: https://doi.org/10.67291/SWIA6798. Chapter 12: the three threats to code set beside a drawn program (Chapter 7).

Landry, Forrest. *Keeping Software Soft; Explicit Code, Safe Change, and the Discipline of Subtraction*. Wytheville, Virginia: UVSM Publications, 2026. ISBN 979-8-90710-245-3. DOI: https://doi.org/10.67291/NDZY5454. Chapter 4, Section IV: the development machine as part of the practice of sharing code (Chapter 7).

Landry, Forrest. *The Study of Pure Causality; On the Philosophy of Computer Science*. Wytheville, Virginia: UVSM Publications, 2026. ISBN 979-8-90710-232-3. DOI: https://doi.org/10.67291/ZVTE4072. Chapter IX, "Drawing the Program": the halting problem (Chapter 5).

Landry, Forrest. *The Tools You Own; Building Infrastructure That Outlives Its Maker*. Wytheville, Virginia: UVSM Publications, 2026. ISBN 979-8-90710-236-1. DOI: https://doi.org/10.67291/WCNX1341. Infrastructure a person owns, built to outlive its maker (Chapter 7).

Landry, Forrest. *The Two Principles of Ethics*. Wytheville, Virginia: UVSM Publications. The symmetry and the continuity of effective choice, the two principles of non-relative ethics (Chapter 4).

Landry, Forrest. *Why AGI Cannot Be Made Safe; The Argument and Its Implications*. Volume 1 of The Impossibility of AGI Safety; The Substrate Needs Convergence Argument. Wytheville, Virginia: UVSM Publications, 2026. ISBN 979-8-951203-02-1.

Landry, Forrest. *Why No Control Suffices; The Technical Case for the Impossibility of AGI Safety*. Volume 2 of The Impossibility of AGI Safety; The Substrate Needs Convergence Argument. Wytheville, Virginia: UVSM Publications, 2026. ISBN 979-8-951203-03-8.

Laurie, Ben, Adam Langley, and Emilia Kasper. "Certificate Transparency". RFC 6962. Internet Engineering Task Force, June 2013. The public logs of certificates (Chapter 6).

Let's Encrypt. "Challenge Types". https://letsencrypt.org/docs/challenge-types/. DNS-01 and HTTP-01 validation of control of a name.

MSD Veterinary Manual. "Chocolate Toxicosis in Animals". https://www.msdvetmanual.com/toxicology/food-hazards/chocolate-toxicosis-in-animals.

Moriarty, Kathleen, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. "PKCS #1: RSA Cryptography Specifications Version 2.2". RFC 8017. Internet Engineering Task Force, November 2016. Section 8: RSA signatures are made on an encoding of a hash of the message, and the message itself is needed to verify them (Chapter 1).

Nakamoto, Satoshi. "Bitcoin: A Peer-to-Peer Electronic Cash System". 2008. https://bitcoin.org/bitcoin.pdf. Describes the blockchain consensus mechanism discussed in Chapter 1.

National Institute of Standards and Technology (NIST). "Secure Hash Standard (SHS)". FIPS PUB 180-4, August 2015. Specification for SHA-1, SHA-224, SHA-256, SHA-384, SHA-512.

National Institute of Standards and Technology (NIST). "Module-Lattice-Based Key-Encapsulation Mechanism Standard". FIPS 203, August 2024. With FIPS 204 and FIPS 205 (August 2024), the first standards for public key methods built to resist a quantum computer (Chapter 2).

Pauli, Darren. "Stuxnet Infected Russian Nuclear Plant". *iTnews*, November 8, 2013. https://www.itnews.com.au/news/stuxnet-infected-russian-nuclear-plant-363578. Eugene Kaspersky's report that Stuxnet crossed an air gap on a USB device (the Appendix).

Proofpoint. "Google and Yahoo Set New Email Authentication Requirements". January 31, 2024. https://www.proofpoint.com/us/blog/email-and-cloud-threats/google-and-yahoo-set-new-email-authentication-requirements. The requirements on senders of more than five thousand messages a day, from February 2024 (Chapter 6).

The Radicati Group. "Email Statistics Report, 2024-2028: Executive Summary". Palo Alto, California, 2024. https://radicati.com/wp/wp-content/uploads/2024/10/Email-Statistics-Report-2024-2028-Executive-Summary.pdf. About 4.6 billion email users worldwide in 2025 (Chapter 6).

Rescorla, Eric. "The Transport Layer Security (TLS) Protocol Version 1.3". RFC 8446. Internet Engineering Task Force, August 2018.

Rice, H. G. "Classes of Recursively Enumerable Sets and Their Decision Problems". *Transactions of the American Mathematical Society* 74, no. 2 (1953): 358-366. No general procedure decides a property of what an arbitrary program does (Chapter 5).

Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM* 21, no. 2 (1978): 120-126. The RSA algorithm, a principal public key cryptosystem.

Webster, A. F., and S. E. Tavares. "On the Design of S-Boxes". In *Advances in Cryptology: CRYPTO '85 Proceedings*, Lecture Notes in Computer Science 218, 523-534. Berlin: Springer, 1986. The strict avalanche criterion (Chapter 5).

Wiesner, Stephen. "Conjugate Coding". *SIGACT News* 15, no. 1 (1983): 78-88. Quantum money (Chapter 2).

Wolfram, Stephen. *A New Kind of Science*. Champaign, Illinois: Wolfram Media, 2002. Computational irreducibility (Chapter 5).

Wolpert, David H. "Physical Limits of Inference". *Physica D: Nonlinear Phenomena* 237, no. 9 (2008): 1257-1281. Any physical device that makes predictions about the universe it belongs to has some fact about that universe it cannot predict (Chapter 5).

Wootters, W. K., and W. H. Zurek. "A Single Quantum Cannot Be Cloned". *Nature* 299 (1982): 802-803.

Index

Each locator gives the chapter and its section: 5.VII is Chapter 5, section VII.

A

accuracy (of control), 5.VI

adversary investment, threats ranked by, 4.II

agency, persistence of, Intro., 7.IV, 7.VI

air gap, 3.II, 3.III, Appendix

algorithmic complexity, 5.III

Android developer verification, 6.III

API (application programming interface), 6.III

app stores, 6.III

artificial intelligence (as controller), 5.IV

ASCII (nine code), 7.I

Ashby, W. Ross (law of requisite variety), 5.VII

asymmetry (in *An Immanent Metaphysics*), 2.VI

atoms, energy and pattern, 2.I

B

bearer certificate, 2.II, 2.III

biometric authentication, 3.II, Appendix

Bitcoin, 1.VI, 6.IV

blockchain, 1.VI, 2.IV, 5.III

bootstrapping (public key to symmetric), 1.V, 1.VII

C

cascade failure, 4.II

causation over choice, 6.II, 6.IV

certificate (web), 1.V, 6.II, 6.III

certificate lifetime, 6.III

Certificate Transparency, 6.III
choice, principle that it drives the system, 6.II
ciphertext, 1.II
code clarity, 7.III, 7.IV
code signing, 6.II
company, small, figures opened to staff, 6.I
competition vs. cooperation, 6.I
complexity of protection, 5.IV
computational irreducibility, 5.VII
Conant, Roger (good regulator theorem), 5.VII
consolidation, 6.II, 6.IV, 6.V
continuity (in *An Immanent Metaphysics*), 2.VI
control theory, 5.I
control-limit argument, 5.VIII
controllability, 5.I
controller, system and the living things downstream (model), 5.III, 5.IV, 5.VII
cooperation, genuine vs. compelled, 6.II
coronal mass ejection, 4.II
COVID-19 (comparability argument), 4.I
CRC (cyclic redundancy check), 5.I
cryptographic enforcement, 6.II
cyanide (model of reliable harm), 5.V
cyberattack (threat model), 4.II
D
data ownership, 2.V
defense in depth, 3.III
design, no agency-neutral, 7.III
developer, app, platform lockdown and, 6.III

development machine, sovereign, 7.V
device attestation, 6.II
Diffie-Hellman key exchange (ephemeral), 1.VII
digital signature, 1.IV, 6.II
Dinur and Nissim (counting questions), Appendix
distinguishability, 2.VII
double-spend problem, 2.IV
duplication (of bearer certificates), 2.II
dvol (root directory), 7.V
E
effective choice, 4.III
email, share of the largest provider, 6.III
email, requirements on bulk senders, 6.III
encryption, of network communications, 3.II, Appendix
ephemeral key exchange, 1.VII
error correction, 5.I
F
forward security, 1.VII, 7.II
Foundry USA (mining pool), 6.IV
Fourier transform, 6.III
fragility (of systems), 5.II, 5.V
G
Gassend, Blaise (measured chip variation), 2.II
geomagnetic storm, 4.II
Gmail, 6.III
good regulator theorem, 5.VII
grid security, 4.I, 4.II, 4.III
Gros Michel banana (monoculture), 6.I

H
halting problem, 5.VII
hash collision, 1.I
hash function, 1.I, 5.II
  as complexity model, 5.II
HTTPS, 1.V
I
identity (cryptographic), 2.IV, 2.V
Immanent Metaphysics, An, 2.VI, 6.II, 7.IV
Incommensuration Theorem, 2.VI
infrastructure lockdown, 6.III
K
key exchange, 1.V, 1.VII
keyed hash, 7.II
keys, who holds them, 6.II, 7.VI
Kolmogorov, Andrey, 5.VII
L
layers of abstraction, 5.I, 5.VI
local vs. global, 6.I
M
MD5, 5.II, 5.III
meat grinder metaphor (hash function), 1.I
mining pools, 6.IV
modularity (biological), 5.VII
MP3 file, 6.III
mutual information, 5.I
N
nine code, 7.I
nine-point computing, 7.I

no-cloning theorem, 2.I
non-relative ethics, 4.III
nonce (in symmetric key protocol), 7.II
O
observability, 5.I, 5.II
one-way function, 1.I, 7.II
opacity, 6.I, 7.III
P
P versus NP, 1.VI
paper-based computing, 7.I
parity bit, 5.I, 7.I
pattern (one of three kinds of thing), 2.I
personal agency, 7.IV, 7.VI
phase change (competition/cooperation), 6.I
physical attack (threat model), 4.II
physically unclonable function, 2.II
plaintext, 1.II
polling, 6.III
precision (of control), 5.VI
principal-agent problem, 7.IV
prion, 5.V
private key, 1.III, 1.IV
proof of work, 1.VI
proxy machines, 7.V
public key cryptography, 1.III
Q
quantum computer, 1.VII, 2.II, 7.II
quantum money, 2.I
query system (secure), 3.I

R

rabbits in Australia (competition), 6.I

ransomware, 6.I

ratchet (replacing the shared secret), 7.II

recording every conversation (transparency in a public role), 6.I

Rice, H. G., 5.VII

rivalrous and non-rivalrous, 2.I

rooted phone, 6.III

RSA, 1.IV

S

sample providers, privacy of, 3.I, 3.II

scale, re-architecting at each order of magnitude, 5.I

seven layers of security, table of, Appendix

SHA-1, 5.III

spoofing, 2.III, 2.VI, 2.VII

stepper motor, 7.I

strict avalanche criterion, 5.II

Stuxnet (crossing an air gap), Appendix

subspace analysis, 5.VI

symmetric key cryptography, 1.II, 7.II

symmetry (in *An Immanent Metaphysics*), 2.VI

T

TCP/IP, 5.I

theobromine, 5.V

threat modeling, 4.II

three threats to code, 7.V

TLS 1.3, 1.VII

transformer design, 4.II

transparency, 6.I, 7.III

Trojan horse, 6.III, 7.III, 7.V
trusted machine, 7.V
U
uniqueness, 2.I, 2.VI, 2.VII
V
Vox (voice messaging service), 6.III
W
WAV file, 6.III
Web3, 2.V
WebSocket, 6.III
Webster and Tavares (strict avalanche criterion), 5.II
Wiesner, Stephen (quantum money), 2.I
Wolfram, Stephen, 5.VII
Wolpert, David, 5.VII
worker thread, 6.III

About the Author

I am Forrest Landry, a philosopher, engineer, and writer. My work spans the foundations of ethics, the architecture of governance, and the infrastructure of civilizational resilience. I am the author and publisher at UVSM Publications LLC in Wytheville, Virginia.

My teaching integrates insights from systems architecture, cryptography, control theory, and the philosophy of choice into a unified framework for understanding how trust, security, and human agency can be preserved and strengthened in an era of increasing technological complexity.