OS.FUN

THE SYSTEM LAYER FOR ONCHAIN SOFTWARE

Blockchain as an Operating System

A kernel-level specification for onchain application systems, spanning the EVM and the SVM, with a userland that runs ordinary programming languages

VERSION	DATE	SILICON	ONLINE
1.0	August 2026	EVM · SVM	@osdotfun

Abstract

Onchain computing has excellent silicon and no operating system. Two mature execution substrates — the Ethereum Virtual Machine as deployed on Base, and the Solana Virtual Machine — now settle hundreds of millions of dollars of value per day with sub-second block times and predictable fees. Yet every application built on top of them re-implements the same missing layer: key custody and session authority, balance and asset accounting, permissioning, scheduling, retry and recovery, cross-domain settlement, indexing, and a user interface for all of it. The result is not a platform but an archipelago — each application an island, the human being the integration layer between them.

This document specifies os.fun: an operating system whose hardware is the blockchain. We treat a chain not as a database or a settlement venue but as a *processor* — a deterministic machine with an instruction set, an address space, a scheduler, and a resource budget — and we build the layer that a processor has always needed in order to become a computer.

The specification is organised around six questions. *Why* a system layer is the missing artefact rather than another application. *What* a blockchain operating system is, formally: a kernel that mediates every interaction between a principal and a domain, a syscall interface that is stable across domains, and a userland in which programs are written in ordinary languages. *Who* acts, expressed as a capability calculus over principals, sessions and delegations, with proofs of non-amplification. *Where* execution happens, expressed as a hardware abstraction layer over the EVM's sequential storage machine and the SVM's parallel account machine. *When* it happens, expressed as a partial order over slots, blocks, confirmations and finality, with the scheduler's clock derived from it. *How* it happens, expressed as the syscall ABI, its two reference implementations — a Rust program for the SVM and a Solidity contract suite for the EVM — and the settlement protocol that binds them.

We give the formal model in full: an abstract machine $M_c = (\Sigma_c, \mathcal{T}_c, \delta_c, \Gamma_c, F_c)$ per domain, a product state with explicit observation lag, a unified resource unit that normalises Ethereum gas and Solana compute units into a single schedulable quantity, a conflict-graph treatment of parallel execution with a serializability theorem, a lattice of capabilities with an attenuation-monotonicity theorem, and a cross-domain commit protocol with a safety condition relating timeout to finality.

Between the model and the implementation we place the result we believe to be new. **Finality is an information-flow lattice.** Every value an onchain program computes carries a *stamp* — how permanent the facts behind it are, in each domain — and every effect declares how permanent it needs them to be. Acting above one's evidence, which is the shape of the failures that empty bridges and re-roll lotteries, is then a typing error rather than an incident report. We give the lattice, a typed calculus over it, and a soundness theorem: a well-typed program's high-finality effects are invariant under any reorganisation below that level. Two consequences follow. The compensation obligations that the kernel imposes dynamically become static side conditions discharged by the checker. And the confirmation depth that every production system hardcodes becomes an optimisation with a closed form, whose solution grows only logarithmically in the value at risk — which is why fixed constants are simultaneously too slow for a coffee and too fast for a treasury.

We then specify the userland. os.fun's process VM (*OSVM*) is a deterministic RV64IM machine with a Merkleised memory, a metered cycle budget and a single trap instruction into the kernel's syscall page. Because it is a real, ordinary instruction set, ordinary language toolchains target it: we specify *osPy*, a deterministic Python dialect, its compiler pipeline, its runtime, and the rules that make Python execution bit-reproducible; and we show the same path for TypeScript, Go, C and Rust. Userland results reach the chains through a settlement layer that is optimistic by default — an interactive

bisection game whose one-step proof is verified onchain in $O(1)$ — and succinct where latency requires it.

Every mechanism in this document is given as mathematics and as code: approximately two thousand lines of Rust, Solidity and Python are specified normatively, including the SVM kernel program, the EVM kernel contracts, the OSVM interpreter core, the osPy compiler's lowering rules and the fraud-proof verifier. Appendices give the complete syscall table, error codes, wire formats, the osPy grammar, and the OSVM instruction set.

The claim of this paper is narrow and testable: the reason onchain software feels unfinished is not a missing chain, a missing rollup or a missing bridge. It is a missing operating system, and the interfaces it requires are now specifiable, implementable and verifiable on the substrates that exist today.

Contents

PART I · THE SIX QUESTIONS

1 Why: The Missing System Layer 2
1.1 The machine that already works 2
1.2 What a person still experiences 3
1.3 The seven re-implementations 3
1.4 Why an operating system, precisely 4
1.5 The analogy, and where it breaks 4
1.6 Thesis and contributions 6
1.7 How to read this document 6
2 What: A Blockchain Operating System, Defined 8
2.1 Definition 8
2.2 The five resources 8
2.3 The privilege model 9
2.4 Kernel and userland: the placement question 10
2.5 What os.fun is not 10
2.6 The design axioms 10
2.7 The system in one picture 12
3 Who: Principals, Identity and Authority 13
3.1 The principal taxonomy 13
3.2 One account across two address spaces 13
3.3 Sessions: the unit of practical authority 14
3.4 Programs as principals 14
3.5 Recovery: the guardian policy 15
4 Where: The Silicon 16
4.1 The EVM as a machine 16
4.2 The SVM as a machine 18
4.3 The honest comparison 19
4.4 Addressing and naming 20
5 When: Time, Ordering and Finality 21
5.1 Three clocks 21
5.2 Finality is a function, not a flag 21
5.3 The latency budget 22
5.4 Ordering as a resource 23
5.5 The scheduler's clock 23
6 How: One Intent, End to End 24
6.1 The intent 24
6.2 The trace 25
6.3 What each stage costs 25
6.4 The same trace, from a Python program 25
6.5 What follows 26

PART II · FORMAL FOUNDATIONS

7 The Abstract Machine and the Domain Product 28
7.1 The domain machine 28
7.2 The product machine 29
7.3 Syscalls as partial functions 30

7.4 The soundness theorem 30
7.5 Reorganisation and convergence 31
8 Resource Accounting: Gas, Compute Units and the OSU 32
8.1 Why a unified unit is necessary 32
8.2 The EVM fee market as a control loop 32
8.3 The SVM fee market is local 33
8.4 Deriving the syscall cost table 33
8.5 Budgets and admission control 34
8.6 Who pays, and in what 34
8.7 Metering the userland 35
9 Concurrency, Conflict Graphs and Serializability 36
9.1 Conflicts 36
9.2 The EVM as the degenerate case 36
9.3 Batch formation 37
9.4 How long to wait before submitting 38
9.5 Contention and queueing 38
10 The Capability Calculus 39
10.1 Capabilities 39
10.2 Attenuation 39
10.3 Revocation in constant time 40
10.4 Encoding and verification cost 40
10.5 Instantiation on the EVM 41
10.6 Instantiation on the SVM 42
11 Cross-Domain Atomicity 44
11.1 The impossibility, stated plainly 44
11.2 The protocol 44
11.3 Safety 45
11.4 Liveness and the filler's economics 45
11.5 Verifying the fill 46
11.6 Replay, idempotence and compensation 47
11.7 Alignment with ERC-7683 47
12 Finality Is a Lattice 48
12.1 The bug class nobody types 48
12.2 Finality is an integrity label 48
12.3 The lattice 49
12.4 Combination is meet; only waiting joins 50
12.5 The cross-domain corollary 50
12.6 What is new here, and what is not 51
13 The Finality Calculus 52
13.1 The core language 52
13.2 The rules 52
13.3 Operational semantics with retraction 53
13.4 Soundness 53
13.5 Inference: the programmer writes almost nothing 54
13.6 What the programmer sees 56
13.7 Gradual finality 56
13.8 The four failures, revisited 57
14 Choosing the Level 58
14.1 The question nobody computes 58

14.2 The hazard model 58
14.3 The objective 58
14.4 From depth to level 59
14.5 Against a constant 59
14.6 Making the policy auditable 60
14.7 Limits 60

PART III · THE KERNEL

15 Kernel Architecture 62
15.1 The subsystems 62
15.2 The kernel's onchain footprint 63
15.3 Kernel invariants 64
15.4 The entry path 64
15.5 What deliberately runs outside 65
16 Processes and the Intent Lifecycle 66
16.1 Intents 66
16.2 The process control block 67
16.3 The lifecycle 67
16.4 Planning 68
16.5 The scheduler 68
16.6 Retry, replacement and idempotence 69
16.7 Isolation between processes 70
17 Memory: Accounts, Storage, Namespaces and Leases 71
17.1 The object model 71
17.2 Allocation and its price 71
17.3 Namespaces: the filesystem 72
17.4 Sharding hot objects 72
17.5 Cold storage and paging 73
17.6 Garbage collection 74
18 The Driver Model 75
18.1 The driver interface 75
18.2 The EVM driver 76
18.3 The SVM driver 77
18.4 Driver conformance 78
18.5 Adding a domain 79
19 The Syscall Interface 80
19.1 Design rules 80
19.2 The eight families 80
19.3 Calling conventions 81
19.4 Wire encoding 82
19.5 Errors 83
19.6 Versioning 83
19.7 Three specifications in full 84
20 Kernel Implementation on the SVM 85
20.1 Program layout 85
20.2 Account layout 85
20.3 A handler in full: `val_transfer` 86
20.4 Compute budget accounting 88
20.5 Program upgrade policy 88

20.6 Testing 88
21 Kernel Implementation on the EVM 89
21.1 The contract map 89
21.2 The capability registry 89
21.3 Storage layout and its price 92
21.4 ERC-4337 integration 92
21.5 Transient storage as the kernel context 93
21.6 Immutability and what it costs 94
21.7 Invariant tests 94

PART IV · USERLAND

22 The Userland Model 96
22.1 What a program is 96
22.2 The manifest 96
22.3 Where a program runs 97
22.4 The process boundary 97
22.5 Drawing 97
23 OSVM: A Deterministic Process Machine 99
23.1 Why an ordinary instruction set 99
23.2 Machine state 99
23.3 Metering 99
23.4 Memory 100
23.5 Determinism 100
23.6 The interpreter core 101
23.7 Execution receipts 104
23.8 Performance 104
24 Python on the OS 105
24.1 The goal, stated precisely 105
24.2 Every place Python is non-deterministic, and what we do 105
24.3 What is in the language 106
24.4 State, typed 107
24.5 Finality in the language 107
24.6 The compiler pipeline 109
24.7 The runtime 109
24.8 The standard library 110
24.9 Metering Python 110
25 A Complete Application, End to End 111
25.1 The application 111
25.2 The manifest 111
25.3 The program 111
25.4 What the compiler produces 116
25.5 One invocation, traced 116
25.6 Settlement of the invocation 117
25.7 The failure path 118
25.8 Total cost of a circle 118
26 Other Languages 119
26.1 The recipe 119
26.2 Rust 119
26.3 TypeScript and JavaScript 120

26.4 The WASM path 123
26.5 The conformance suite 123
27 Verification and Settlement 124
27.1 The claim 124
27.2 Replay: the cheap check everybody can run 124
27.3 The bisection game 124
27.4 The one-step proof 125
27.5 Succinct proofs 126
27.6 Bonds and incentives 127

PART V · SYSTEM SERVICES

28 Identity, Sessions and Key Management 129
28.1 Credentials 129
28.2 Establishing a session 130
28.3 Rotation 130
28.4 Recovery 130
28.5 Threat cases 131
29 Value: Balances, Transfers and the Unified Ledger 132
29.1 The unified balance 132
29.2 Asset classes and resolution 132
29.3 Payments 133
29.4 Swaps and the price band 133
29.5 The ledger 133
30 The Package Manager and App Registry 134
30.1 Publishing 134
30.2 Updating 134
30.3 Distribution and integrity 134
30.4 Revenue 135
30.5 Quarantine, and its limits 135
31 IPC, Events and Notifications 136
31.1 The event model 136
31.2 Inter-app communication 136
31.3 Scheduled and deferred execution 137
31.4 Notifications 137
32 The Filesystem: Journal, Index and Availability 138
32.1 The journal 138
32.2 Indexing 138
32.3 Queries 138
32.4 Privacy 139

PART VI · SECURITY, ECONOMICS, GOVERNANCE

33 Threat Model and Security Analysis 141
33.1 What is being protected 141
33.2 Trust assumptions, enumerated 141
33.3 The attack catalogue 141
33.4 What we do not protect against 144
33.5 Formal properties and where they hold 144
33.6 Verification plan 145
34 Economics of the System Layer 146
34.1 Who pays for what 146

34.2 The fee schedule ... 146
34.3 Unit economics ... 147
34.4 Incentive compatibility ... 147
34.5 On tokens ... 148
35 Governance, Versioning and ABI Stability ... 149
35.1 The immutability map ... 149
35.2 ABI stability rules ... 149
35.3 Parameter governance ... 149
35.4 Upgrade mechanics ... 150
35.5 Emergency response without a pause button ... 150
35.6 Standards ... 150

PART VII · EVALUATION AND HORIZON

36 Performance Model and Evaluation ... 153
36.1 Method and honesty about it ... 153
36.2 Latency ... 153
36.3 Throughput ... 154
36.4 Cost ... 154
36.5 What breaks first ... 155
36.6 Sensitivity ... 156
37 Conformance and the Reference Implementation ... 157
37.1 Status ... 157
37.2 The conformance suite ... 157
37.3 Reference implementation layout ... 158
38 Roadmap: The Boot Sequence ... 159
39 Related Work ... 160
39.1 Capability operating systems ... 160
39.2 Exokernels ... 160
39.3 Onchain execution environments ... 160
39.4 Account abstraction and intents ... 160
39.5 Verifiable off-chain execution ... 161
39.6 Information flow, and what we did to it ... 161
39.7 Deterministic language runtimes ... 161
40 Limitations, Open Problems, and Conclusion ... 163
40.1 Limitations ... 163
40.2 Open problems ... 163
40.3 Conclusion ... 164

PART VIII · APPENDICES

A Syscall Reference ... 166
A.1 Reading these entries ... 166
A.2 Family `cap` — authority ... 166
A.3 Family `proc` — processes ... 167
A.4 Family `mem` — objects and storage ... 167
A.5 Family `val` — value ... 168
A.6 Family `xdom` — cross-domain ... 168
A.7 Family `ns` — naming ... 169
A.8 Family `evt` — events ... 169
A.9 Family `sys` — system ... 169
B Error Codes ... 170

C Wire Formats 171
C.1 TLV tags 171
C.2 Digest preimages 171
C.3 Constraint encoding 171
D osPy Grammar and Determinism Rules 172
D.1 Grammar 172
D.2 Finality annotations 173
D.3 Determinism rules 173
E OSVM Instruction Set 174
E.1 RV64I — integer base 174
E.2 M extension — multiply and divide 175
F Notation and Glossary 176
F.1 Symbols 176
F.2 Glossary 177
G Deployed Contract Interfaces 179
H References 181

PART I

The Six Questions

Before a specification comes a set of questions. This part answers six of them — why a system layer is the missing artefact, what such a layer is, who acts within it, where its instructions execute, when their effects become real, and how a single user intention traverses all of it. Each question is answered informally here and formally in the parts that follow.

CHAPTER 1

Why: The Missing System Layer

The silicon shipped years ago. Nobody wrote the operating system.

1.1 The machine that already works

It has become unfashionable to say so, but the execution problem is largely solved. Two production substrates now offer the properties that a system designer would have described, five years ago, as prerequisites for consumer software: sub-second inclusion, fees measured in fractions of a cent, programmable accounts, and settlement guarantees strong enough to move real value. They are the Ethereum Virtual Machine as deployed on Base, and the Solana Virtual Machine.

Property	Base (EVM domain)	Solana (SVM domain)
Block / slot period	2 s, with sub-block preconfirmations on the order of 200 ms	400 ms target slot
Inclusion cost, simple transfer	10^{-4} – 10^{-2} USD	10^{-4} – 10^{-3} USD
Execution model	sequential, single-threaded per block	parallel over non-conflicting accounts
Metering unit	gas	compute units (CU)
Block compute budget	30 M gas target / 60 M cap (configurable)	on the order of 5×10^7 CU
State model	contract-owned storage tries, 32-byte slots	flat account array, programs own accounts, data up to 10 MiB
Account authority	ECDSA secp256k1 [1], optionally delegated code (EIP-7702)	Ed25519, plus program-derived addresses (PDAs)
Settlement	L2 blocks settle to Ethereum; fraud-proof window	single-layer consensus, optimistic confirmation then finality

None of these numbers are physics. They are configuration, and they have been moving in one direction for six years. The relevant observation is not that they are impressive but that they crossed a threshold: the substrate is no longer the reason onchain software feels unfinished.

> **ON NUMBERS IN THIS DOCUMENT**
> Protocol parameters cited here — compute limits, fee constants, slot times — are those observed at the time of writing and are subject to change by governance in both ecosystems — by the EIP process on the EVM side and the SIMD process [2] on the SVM side. Wherever a mechanism depends on a parameter, we name the parameter symbolically and treat its value as an input, not an assumption. Appendix F lists every symbol used.

1.2 What a person still experiences

Consider the smallest interesting task a person might want to perform in 2026: *pay a friend, in the token they want, from the balance you have.*

The balance is on Base, in USDC. The friend wants SOL. In principle this is two state transitions and a price. In practice the user is asked to: install a wallet that speaks EVM; install a second wallet that speaks SVM; discover that the second wallet needs SOL to pay for its own first transaction; find a bridge; approve a token allowance in one transaction and spend it in a second; wait for a fraud-proof-window-dependent settlement they were never told about; copy an address between two applications, in two incompatible encodings, and verify it by reading hexadecimal aloud; and finally repeat a failed transaction because the priority fee estimate was stale by four hundred milliseconds.

Every one of these steps is a system-layer failure. Not one of them is an application-layer failure, and not one of them can be fixed inside an application.

What the user sees	The subsystem that is missing
“Which wallet do I use?”	Identity and credential management
“I don’t have gas on this chain.”	Resource accounting and fee abstraction
“Approve, then swap, then confirm, then wait.”	Transaction lifecycle management and scheduling
“Is this address the right one?”	Naming and address resolution
“It failed. Did it fail on both sides?”	Atomicity across execution domains
“Where did my money go?”	The filesystem: a durable, queryable record
“I didn’t know that finished.”	Interprocess communication and events
“This app can spend everything I own, forever.”	A capability model with attenuation and revocation

1.3 The seven re-implementations

The consequence of an absent system layer is not that the work goes undone. It is that every team does it again, incompatibly, and the user pays the integration cost with their attention.

#	What every onchain app rebuilds	Its name in an operating system
1	Wallet connection, signing, session handling	Authentication and credential store
2	Balance aggregation across tokens and chains	Virtual filesystem and accounting
3	Allowances, spend limits, revocation UI	Capability system and access control lists
4	Nonce management, gas estimation, retry, reorg handling	Scheduler, interrupt handling, fault recovery
5	Bridging, routing, cross-chain message passing	Bus arbitration and direct memory access
6	Indexers, subgraphs, transaction history	Filesystem index and journal
7	Push notifications, activity feeds, webhooks	Signals, interprocess communication, event bus

The waste is quantifiable. Let N be the number of independent onchain applications, $k = 7$ the number of subsystems above, and c_i the engineering cost of subsystem i built to production quality. Total industry expenditure on the system layer is

$$C_{\text{dup}} = N \sum_{i=1}^{k} c_i, \tag{1.1}$$

whereas the cost of the same layer built once and shared is $C_{\text{os}} = \sum_{i=1}^{k} c_i + N\varepsilon$, where ε is per-application integration cost. The ratio $C_{\text{dup}}/C_{\text{os}} \to \left(\sum_i c_i\right)/\varepsilon$ as N grows: the argument for a system layer is not aesthetic, it is the same $O(N) \to O(1)$ argument that justified every operating system in the history of computing.

But cost duplication is the weaker half of the argument. The stronger half is that some properties cannot be produced by duplication at all.

PROPOSITION 1.1 · System properties are not application-local
Let P be a property of the form "across all applications a principal uses, quantity Q is unified." Examples: one identity, one balance, one permission ledger, one activity history, one session. Then no single application A can establish P for a principal who also uses $A' \neq A$, unless A mediates every interaction between the principal and A' — that is, unless A is a system layer.

PROOF. Suppose A establishes P. Take any interaction x between the principal and A' that A does not mediate. Since Q is a function of the union of interactions, and x can change Q without A observing it, A's view of Q may differ from the true Q; hence P fails for some execution. So A must mediate all such x. A component that mediates every interaction between principals and applications is, by definition, the system layer. □

This is why the answer to fragmentation is not a better application. Wallets, superapps and aggregators each attack one row of the table above and none of them can attack the column, because attacking the column means owning the resources: keys, balances, permissions, ordering, and the record of what happened. Owning resources on behalf of programs, and mediating their access, is the definition of an operating system.

1.4 Why an operating system, precisely

Tanenbaum's two-sentence definition of an operating system [3] is the one we adopt, because it survives translation to this setting intact. An operating system is (i) a *resource manager*, allocating scarce machine resources among competing programs, and (ii) an *extended machine*, presenting a cleaner abstraction than the hardware exposes.

Both halves apply, and both are unimplemented today.

- **Resource manager.** An onchain machine has five scarce resources: compute, state, ordering priority, authority, and liquidity (Chapter 2). Today they are managed by no one. Each application competes for block space independently, holds its own approvals, and keeps its own float.
- **Extended machine.** The raw interface — sign this 1232-byte packet; set this 32-byte storage slot; pay for it in a token you may not hold — is a hardware interface. It is roughly as suitable for application authors as writing to IDE registers, and for the same reason: it exposes the machine's constraints rather than the program's intent.

1.5 The analogy, and where it breaks

Analogies are load-bearing in this document, so it is worth stating precisely where this one holds and where it does not. Credibility here is cheap to lose.

Classical concept	Onchain analogue	Where the analogy breaks
CPU	Domain runtime (EVM, SVM)	Replicated and adversarial; every instruction is billed and publicly observable; the CPU may reorder or drop your instruction stream for profit
RAM	Accounts and storage slots	Persistent by default, priced per byte and per write, globally readable, and often not deletable
Process	An executing intent	No preemption and no interrupts: a transaction runs to completion or reverts entirely. There is no "resume after page fault"
Syscall	Instruction / cross-program invocation	Costs money, executes in public, and its effects can be reverted by a reorganisation after they appeared to succeed
Filesystem	State plus data availability	Writes cost 10^5–10^7 times more than reads; deletion is refund, not erasure; history is immutable and public
Scheduler	Block builder / leader	Not ours to control. Ordering is auctioned, and the auction is itself a source of value extraction
Privilege ring	Program ownership of accounts	No supervisor mode; all code is public; "privileged" means cryptographically authorised, not hardware-enforced
Device driver	Domain adapter	Devices here are Byzantine, have their own economics, and can disappear from under a running process

Three of these breakages shape the entire design and are worth naming now.

No preemption A domain transaction is a run-to-completion atomic unit. The kernel therefore cannot schedule *within* a transaction; it schedules *between* transactions, and it must plan multi-step work as an explicit state machine that survives partial completion. This is closer to a batch operating system with checkpointed jobs than to a time-sharing kernel, and Chapter 16 models it as such.

Adversarial scheduling The entity that orders our transactions is not part of our trust base and may profit from harming us. The kernel's scheduling policy is therefore an economic policy, not merely an algorithmic one (Chapter 5 and Chapter 34).

Delayed truth The result of a syscall is not a value but a value plus a confidence that increases over time. Every kernel interface in this specification returns a commitment level, and no interface pretends that "it succeeded" is a binary fact at the moment of return (Chapter 5).

1.6 Thesis and contributions

> **THESIS**
> The blockchain is not a database and not a settlement rail. It is a processor: a deterministic machine with an instruction set, an address space, a metering model and a scheduler. What is missing between that processor and the people who would use it is precisely an operating system — a kernel that owns the resources, a stable syscall interface across heterogeneous processors, and a userland in which programs are written in ordinary languages. That layer is specifiable today, on the substrates that exist today, and this document specifies it.

The contributions of this document are:

1. **A formal model** of a multi-domain onchain machine: per-domain abstract machines with explicit finality and observation lag, and a product state space with a well-defined consistency criterion (Chapter 7).
2. **A unified resource unit**, the OSU, that normalises Ethereum gas and Solana compute units into one quantity the kernel can schedule and price, with the conversion, hedging and admission-control mathematics (Chapter 8).
3. **A concurrency theory for onchain scheduling** — conflict graphs derived from declared access sets, a serializability theorem covering both the SVM's parallel execution and the EVM's sequential execution as a degenerate case, and the resulting batching algorithm (Chapter 9).
4. **A capability calculus** for authority: a lattice of capabilities with attenuation, delegation and revocation, a non-amplification theorem, and concrete instantiations as EIP-7702 delegations, ERC-4337 validation modules and SVM session accounts (Chapter 10, Chapters 20–21).
5. **A cross-domain commit protocol** with an explicit safety condition relating the timeout parameter to finality bounds, and an economic characterisation of the capital a filler must hold (Chapter 11).
6. **The finality calculus** — the contribution we consider new. Finality is an information-flow lattice; values carry stamps and effects carry requirements; acting above one's evidence is a typing error. We give the lattice, the typing rules, a reorg-noninterference theorem, an inference algorithm that runs in $O(n \cdot h)$, a gradual variant for interoperating with untyped code, and an optimiser that derives confirmation depth from value at risk instead of from a constant (Chapters 12–14).
7. **A syscall interface** — 41 calls in eight families — that is stable across the EVM and the SVM, with reference implementations in Rust (SVM kernel program) and Solidity (EVM kernel contracts), given in full (Chapters 19–21, Appendix A).
8. **A userland that runs ordinary languages.** OSVM is a deterministic RV64IM machine with Merkleised memory and metered cycles; osPy is a deterministic Python dialect that compiles to it; the settlement layer verifies its execution optimistically by bisection or succinctly by proof (Chapters 23–27).

1.7 How to read this document

The six questions of Part I are not a rhetorical device; each maps to a technical subsystem specified later, and the mapping is the document's index.

os.fun — reading map

QUESTION	INFORMAL	FORMAL / NORMATIVE	SUBSYSTEM
WHY	ch. 1	ch. 33–34 economics, security	rationale, threat model
WHAT	ch. 2	ch. 7 abstract machine ch. 15 kernel architecture	kernel definition
WHO	ch. 3	ch. 10 capability calculus	identity, sessions, delegation (ch. 28)
WHERE	ch. 4	ch. 17–18 memory, HAL ch. 20–21 per-domain kernels	drivers, addressing
WHEN	ch. 5	ch. 12–14 finality lattice, level optimiser ch. 11 commit	typed effects, scheduler (ch. 16), atomicity
HOW	ch. 6	ch. 19 syscall ABI	userland (ch. 22–27)

The WHEN row is where this document departs from prior work: levels are not annotations on a protocol, they are types, and ch. 13 proves what that buys.

Appendices: A syscalls · B errors · C wire format · D osPy grammar
E OSVM ISA · F notation · G contract interfaces

Figure 1.1 The six questions and the chapters that answer them formally.

Readers who want the shortest path to the mechanism should read Chapter 6 (a single intent traced end to end), then Chapter 19 (the syscall interface), then Chapters 23–25 (the userland and Python). Readers who want the argument should read Part I and Part VI. Readers who want to implement should treat Parts III–IV and the appendices as normative and everything else as commentary.

CHAPTER 2

What: A Blockchain Operating System, Defined

A kernel that owns five resources, a syscall interface that hides two processors, and a userland that runs real programs.

2.1 Definition

DEFINITION 2.1 · Blockchain operating system

A *blockchain operating system* is a system $\mathcal{O} = (K, S, U, \Sigma_{\text{os}})$ where

- K, the *kernel*, is a set of programs deployed inside execution domains, which together mediate every access by a principal to a resource of those domains;
- S, the *syscall interface*, is a domain-independent set of typed operations $s : \mathcal{C} \times \mathcal{A} \to \mathcal{R}$ from a capability and an argument tuple to a result, implemented by K in every domain;
- U, the *userland*, is a set of programs written against S and executed either inside a domain or inside a verifiable process machine;
- Σ_{os} is the OS state: the process table, the capability store, the namespace, and the accounting ledger, itself stored in domains.

$\mathcal{O}$ is *sound* if every effect on domain state attributable to a userland program $u \in U$ is the image of a syscall in S authorised by a capability held by u.

The soundness condition is the whole point. An operating system that programs can bypass is a library. In the onchain setting, bypass is always physically possible — anyone may submit a raw transaction — so "mediates every access" is a statement about *authority*, not about physical interposition: assets and permissions live in accounts whose authority is the kernel, so the only path that moves them is a syscall the kernel authorised. Chapter 10 makes this precise; Chapters 20–21 implement it.

2.2 The five resources

Classical operating systems manage CPU, memory, storage, devices and network. An onchain machine has a different, smaller, and stranger set. Naming them correctly determines everything downstream, because the kernel's job is exactly to allocate them.

Resource	What it is	Who competes for it
Compute	Metered execution: gas on the EVM, compute units on the SVM. Bounded per transaction and per block.	Every transaction in the mempool or leader queue
State	Bytes that persist: storage slots, account data. Priced once at write time (EVM) or held as a rent-exemption deposit (SVM).	Every program that wants to remember something

Ordering	Position within a block, and inclusion at all. Purchased with priority fees and, in practice, auctioned.	Every transaction whose value depends on sequence: trades, liquidations, mints
Authority	The right to move a specific asset or mutate specific state: signatures, allowances, delegations, session keys.	Every application that asks a user for permission
Liquidity	Capital positioned where and when a settlement needs it — the resource a cross-domain transfer actually consumes.	Every intent that crosses a domain boundary, and every filler serving it

The last two are the ones classical operating systems do not have, and they are the ones that make this an interesting problem rather than a porting exercise.

Authority is a resource because it is finite, transferable, and dangerous to over-allocate: an unlimited ERC-20 approval is the onchain equivalent of running every application as root, and the industry's default answer for a decade has been to do exactly that. The kernel therefore treats authority the way a capability operating system does [4], [5] — as an object that can be minted, attenuated, delegated and revoked, never as an ambient property of the caller (Chapter 10).

Liquidity is a resource because cross-domain movement is not a message- passing problem, it is an inventory problem. A user's intent to convert value on domain a into value on domain b is satisfied by someone who already holds inventory on b and is compensated for the risk of holding it. The kernel's cross-domain subsystem is therefore a scheduler over capital, with the mathematics of queues and inventory rather than of packets (Chapter 11, Chapter 34).

2.3 The privilege model

There is no supervisor bit. Privilege in an onchain machine is cryptographic and economic, not architectural, so we define rings by *what can revoke what* rather than by what the hardware enforces.

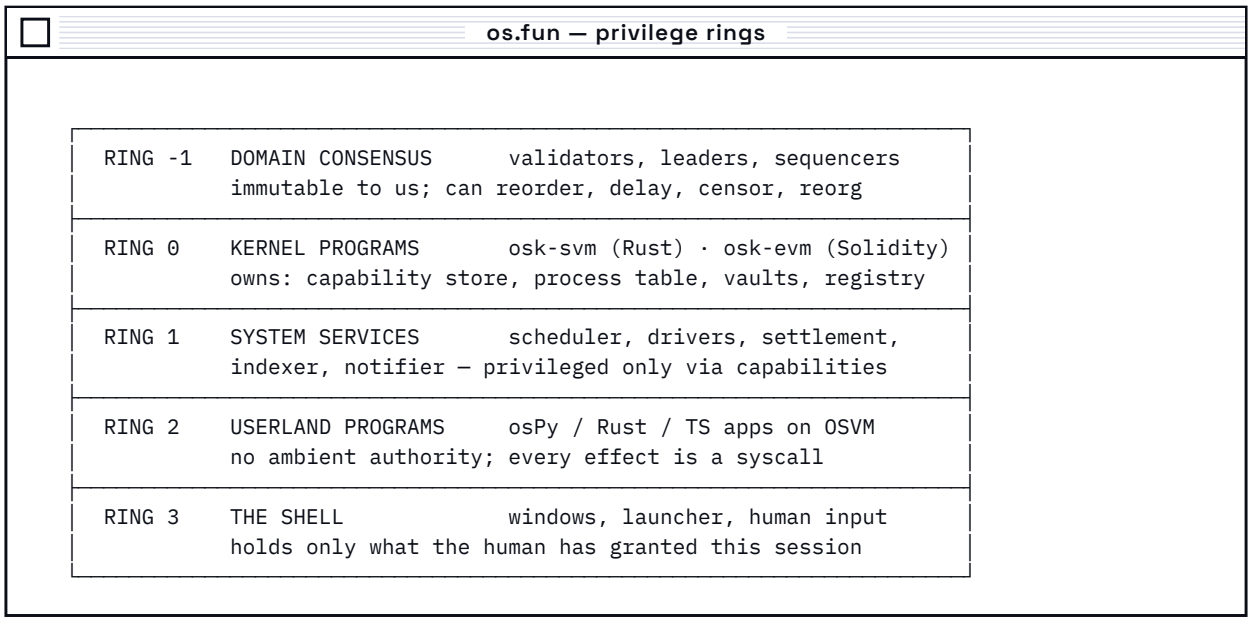

Figure 2.1 Rings are defined by revocation authority, not by hardware enforcement. An outer ring can only act through interfaces the next inner ring exposes, because the inner ring holds the account authority.

Ring -1 is written with a negative index deliberately: consensus is beneath the kernel and outside its control. Every guarantee in this document is conditional on ring -1 behaving within its own security model, and Chapter 33 states exactly which assumptions are load-bearing.

2.4 Kernel and userland: the placement question

The design question that recurs in every chapter is: *which code must run inside a domain, and which may run outside it and be verified?* We answer it with an explicit placement function.

DEFINITION 2.2 · Placement

Let $\mathcal{F}$ be the set of kernel functions. A *placement* is a map $\pi : \mathcal{F} \to \{\text{onchain}, \text{verified}, \text{trusted}\}$ where

- $\pi(f) = \text{onchain}$: f executes inside a domain transaction; its integrity follows from consensus;
- $\pi(f) = \text{verified}$: f executes off-domain, and a proof or a challengeable claim of its correct execution is checked onchain;
- $\pi(f) = \text{trusted}$: f executes off-domain and its result is accepted without proof.

A placement is *admissible* if $\pi(f) = \text{trusted}$ implies that no incorrect result of f can cause loss of assets or authority — at worst it can cause degraded liveness or a worse price.

AXIOM 2.3 · Admissible placement

os.fun admits only admissible placements. Concretely: authority checks, asset custody, accounting and settlement are `onchain`; userland program execution, routing, price discovery and indexing are `verified`; user interface rendering, cosmetic ordering and cache warming may be `trusted`.

This single rule explains most of the architecture. It is why the capability store is a contract and not a server; why the osPy runtime may execute off-domain but its state root must be settled; and why the scheduler is allowed to be a piece of ordinary software that anyone can replace.

2.5 What os.fun is not

- **Not a chain.** os.fun introduces no consensus, no validator set, no new token-secured security assumption. It runs inside the domains it targets.
- **Not a bridge.** It does not mint wrapped assets or hold a canonical cross-chain ledger. Its cross-domain subsystem is an intent settlement protocol over existing liquidity venues (Chapter 11).
- **Not a wallet.** A wallet is a credential store. The credential store is one subsystem of the kernel (Chapter 28), and the smallest one.
- **Not a rollup or an appchain.** Userland execution is verified against the domains, not settled into a separate chain with its own trust assumptions.
- **Not custodial.** Every asset path in this specification requires a capability that a principal minted from a key the principal holds.

2.6 The design axioms

Everything downstream is derived from seven axioms. They are numbered because later chapters cite them when a design choice is forced.

AXIOM 2.4 · A1 — Self-custody
No kernel path may move an asset without a capability derived, through a chain of attenuations, from a signature by a key the principal controls.

AXIOM 2.5 · A2 — Determinism
Every function whose result is used to authorise or settle must be a pure function of committed data. Non-deterministic inputs (time, randomness, prices) enter only through syscalls that name their source and their commitment level.

AXIOM 2.6 · A3 — Explicit, attenuable authority
Authority is an object, not an ambient property. It has a scope, a cap, an expiry, and a revocation path, and delegation can only narrow it.

AXIOM 2.7 · A4 — Domain neutrality
The syscall interface must not leak which domain implements it. Domain differences are the driver's problem, not the program's.

AXIOM 2.8 · A5 — Everything is metered
Every syscall has a declared resource cost in OSU. A program that cannot pay is refused admission rather than being allowed to fail midway.

AXIOM 2.9 · A6 — No new trust root
The system may add convenience, capital and computation, but not a new entity whose failure loses user funds.

AXIOM 2.10 · A7 — Legibility
Any authority a principal grants must be renderable as one sentence a non-expert can evaluate, and any effect must be attributable to a named program and a named capability.

A7 is not decoration. It is the constraint that rules out the industry's current authority model, in which the only honest rendering of a token approval is "this program may take all of your USDC, now and forever."

2.7 The system in one picture

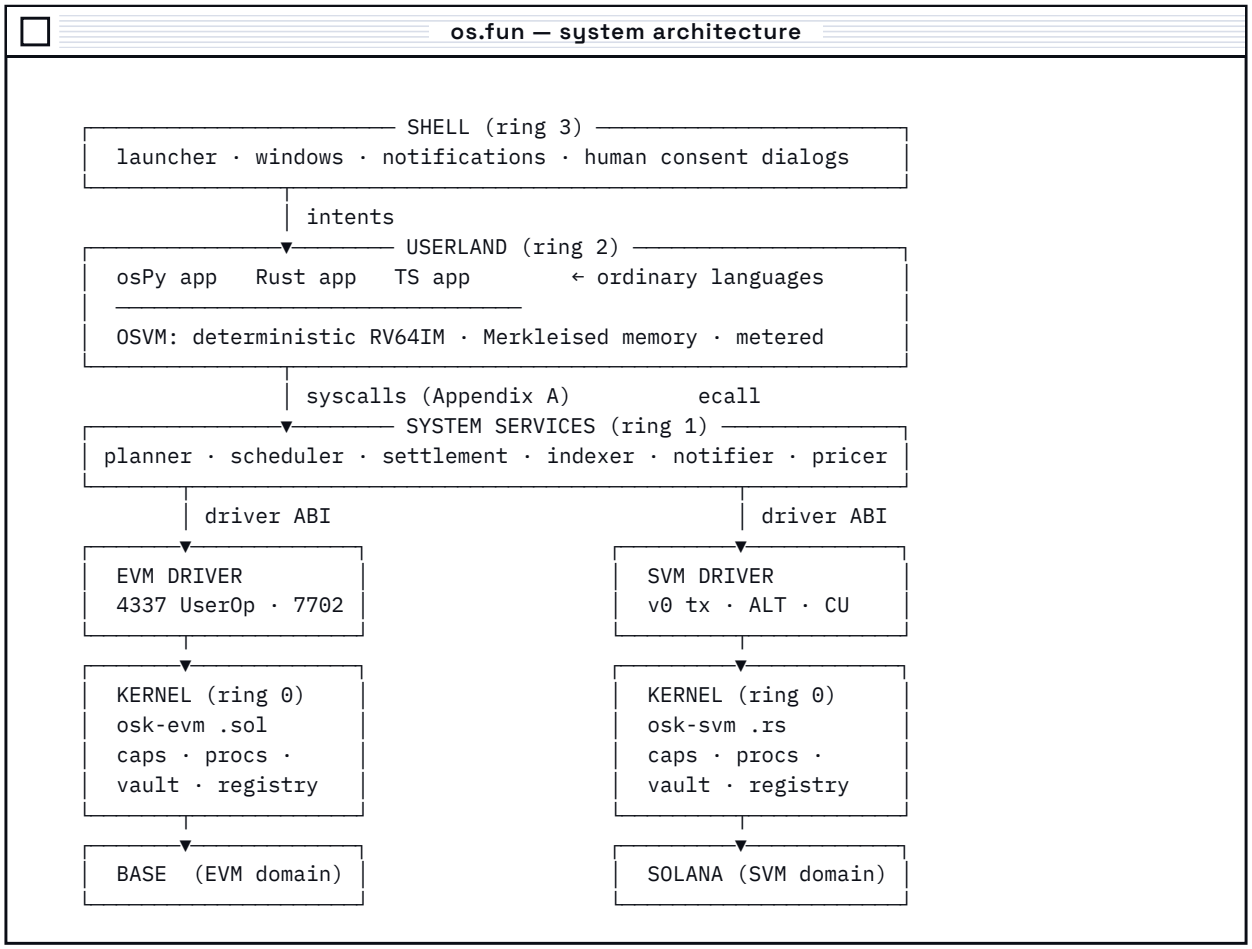

Figure 2.2 The kernel is deployed inside both domains; drivers translate one syscall interface into two very different transaction shapes; userland programs never see a chain.

CHAPTER 3

Who: Principals, Identity and Authority

Six kinds of actor, one account, and a rule that authority may only ever shrink as it is passed along.

3.1 The principal taxonomy

"Who" in an operating system is answered by the subject model. Ours has six kinds of principal, and the distinctions matter because each has a different key material, a different revocation path, and a different failure mode.

Principal	Key material	Failure mode if compromised
Human	Root credential: passkey (P-256), hardware key, or seed-derived key. Held by the person.	Total loss, unless guardians are configured
Session	Ephemeral keypair held by a device or browser, authorised by the human for a bounded scope and time.	Bounded loss: at most the session's caps until expiry
Program	No key. Acts under a capability granted to its code identity (`code_hash` on the EVM, program ID plus PDA on the SVM).	Bounded by the capability; upgrade authority is the real risk
Service	Operational keypair (scheduler, relayer, indexer).	Liveness degradation only, by construction (A6)
Filler	Capital-bearing keypair that fronts liquidity for cross-domain intents.	Their own capital; user is protected by escrow (ch. 11)
Validator	Domain consensus keys, outside our trust base.	Censorship, reordering, reorg — mitigated, never eliminated

3.2 One account across two address spaces

A person is not an address. On the EVM they are a 20-byte address derived from a secp256k1 key; on the SVM they are a 32-byte Ed25519 public key rendered in base58. The OS account binds these into one identity without inventing a new global registry.

DEFINITION 3.1 · OS account

An *OS account* is a triple $\alpha = \left(r, \{a_c\}_{c \in C}, G\right)$ where r is the root credential's public key, a_c is the account controlled by α in domain c, and G is the guardian policy. The binding is witnessed in each domain by a kernel record

$$\mathrm{bind}_c(\alpha) = (r, a_c, \sigma_r), \quad \sigma_r = \mathrm{Sign}_r(\mathrm{H}(\mathrm{os.bind} \parallel c \parallel a_c \parallel n)) \tag{3.1}$$

where n is a monotonic binding nonce. An account is *coherent* if for all c, $\mathrm{bind}_c(\alpha)$ verifies against the same r.

Two properties follow. First, the binding is verifiable in each domain independently: the EVM kernel checks a secp256k1 or P-256 signature; the SVM kernel checks an Ed25519 signature. Second, no domain is authoritative over another — there is no "home chain" whose failure invalidates the account, which is what A6 demands.

> **WHY NOT A SINGLE KEY ACROSS BOTH DOMAINS?**
> It is technically possible to derive both an Ed25519 and a secp256k1 key from one seed, and wallets do this. It is a poor foundation for an OS: it ties the two domains' compromise events together, it forbids passkeys as a root credential (WebAuthn signs with P-256 and never exports the key), and it makes rotation on one domain impossible without rotation on the other. The kernel therefore binds *identities*, not keys.

3.3 Sessions: the unit of practical authority

Humans should sign rarely. Programs should act often. The bridge between those two facts is the session: a short-lived key with an explicitly bounded capability, which is what makes an OS usable without making it custodial.

> **DEFINITION 3.2 · Session**
> A session is $s = (k_s, \alpha, c^*, t_0, t_1, \text{cap})$: an ephemeral public key k_s, its owner account α, target domains c^*, a validity window $[t_0, t_1]$, and a capability `cap` (Chapter 10). The kernel accepts an action signed by k_s iff the action is in the image of `cap`, the domain clock lies in $[t_0, t_1]$, and no revocation record exists.

The value of bounding sessions is quantifiable, and this is the argument to make to anyone who believes session keys are a security regression relative to signing every transaction.

> **PROPOSITION 3.3 · Session risk bound**
> Let L be the loss from a compromised session key, B the session's spend cap per epoch, $T = t_1 - t_0$ the window in epochs, and τ the expected detection-and-revocation delay in epochs. Then
>
> $$L \le B \cdot \min(T, \tau). \tag{3.2}$$
>
> Under the industry-standard alternative — an unlimited ERC-20 approval to a contract — the corresponding bound is $L \le \text{balance}(\alpha)$ with no time bound at all.

The practical parameters follow from Eq. (3.2) directly: a session with $B = 50$ USD and $T = 1$ day carries a bounded worst case that a person can actually reason about, which is what A7 requires.

3.4 Programs as principals

A program is a principal without a key. Its identity is its code, and the kernel must decide what that means when code can be upgraded.

- On the EVM, a program's identity is the pair (`address`, `codehash`). The kernel stores the `codehash` at capability-grant time and re-checks it at use time via `EXTCODEHASH`, so that an upgrade silently invalidates capabilities rather than silently inheriting them.
- On the SVM, a program's identity is its program ID, and the executable is held in a separate program-data account with an upgrade authority. The kernel stores the program-data hash and the upgrade authority, and treats "upgrade authority is not `None`" as a disclosed property of the capability (A7).

INVARIANT 3.4 · No silent inheritance
A capability granted to a program identity is void if the program's code hash changes. Re-granting requires a fresh authorisation by the principal.

This is a deliberate departure from the current norm, in which a user's approval to a proxy contract survives arbitrary implementation replacement. Under A7, an approval that can change meaning after the fact is not legible, therefore it is not permitted.

3.5 Recovery: the guardian policy

Self-custody without recovery is a promise of eventual loss. The kernel's recovery model is a k-of-n threshold over guardians with a mandatory timelock, expressed as kernel state so that it works identically in both domains.

Let n guardians each fail independently with probability p (loss of key or unavailability) and let an adversary independently compromise each with probability q. Recovery requires k signatures. The probability that recovery is *impossible* (too few honest available guardians) and the probability that it is *hostile* (adversary reaches the threshold) are

$$P_{\text{lost}} = \sum_{i=0}^{k-1} \binom{n}{i} (1-p)^i p^{n-i}, \quad P_{\text{hostile}} = \sum_{i=k}^{n} \binom{n}{i} q^i (1-q)^{n-i}. \tag{3.3}$$

With $n = 5$, $k = 3$, $p = 0.1$, $q = 0.05$: $P_{\text{lost}} \approx 8.6 \times 10^{-3}$ and $P_{\text{hostile}} \approx 1.2 \times 10^{-3}$. The timelock Δ_G converts the second number into a recoverable event rather than a terminal one, since the account's root credential can veto a pending recovery during the window. Chapter 28 specifies the state machine and the veto path.

CHAPTER 4

Where: The Silicon

Two processors with the same job description and almost no common mechanism. The kernel's task is to make that difference a driver detail.

4.1 The EVM as a machine

The EVM [6], as deployed on Base, is a sequential stack machine over a persistent key–value store, with a metering model that prices every operation and a call model that composes contracts synchronously.

Aspect	Behaviour
Word	256-bit; stack of 1024 words; memory is a linear byte array priced quadratically
State	Per-account storage: `mapping(bytes32 => bytes32)`, committed into a Merkle-Patricia trie
Execution	One transaction at a time, sequential within the block; reentrancy is possible and must be defended against
Composition	`CALL` / `DELEGATECALL` / `STATICCALL`; synchronous; depth limit 1024; 63/64 gas forwarding rule
Metering	Gas, with dynamic pricing for state access (EIP-2929 [7] cold/warm) and refunds capped at 20% (EIP-3529 [8])
Fee market	EIP-1559 [9] base fee, burned; priority fee to the builder; L2 additionally pays an L1 data cost
Authority	ECDSA over secp256k1; since EIP-7702 [10] an EOA may delegate its code to a contract; ERC-4337 [11] provides an alternative validation pipeline

The gas constants matter for the kernel's cost model, so we fix the ones the specification depends on:

Operation	Gas	Consequence for the kernel
Transaction base	21 000	Fixed overhead per domain call; amortised by batching (ch. 9)
`SLOAD` cold / warm	2 100 / 100	Capability lookups must be

		single-slot and warm-friendly
`SSTORE` zero→nonzero	20 000	Never write a new slot on a hot path; pack and reuse
`SSTORE` nonzero→nonzero	2 900	Counters and nonces are cheap; allocation is not
`TSTORE` / `TLOAD`	100 / 100	Transient storage is the correct home for intra-transaction kernel context (EIP-1153 [12])
Cold account access	2 600	Access lists (EIP-2930 [13]) materially change multi-contract kernel paths
`KECCAK256`	30 + 6 per word	Capability IDs are hashed once and cached
Calldata byte	4 zero / 16 non-zero, with a floor cost (EIP-7623 [14])	On an L2 the dominant cost is data, not compute — see Eq. (4.1)

EVM · BASE

On an OP-Stack L2 such as Base, the fee a user pays decomposes into an execution term and a data term:

$$\text{fee} = \underbrace{g_{\text{exec}} \cdot (b_{\text{L2}} + p)}_{\text{L2 execution}} + \underbrace{\sigma \cdot (z_{\text{cd}} \cdot c_0 + n_{\text{cd}} \cdot c_1) \cdot b_{\text{L1}}^{\text{blob}}}_{\text{L1 data availability}} \tag{4.1}$$

where g_{exec} is gas used, b_{L2} the L2 base fee, p the priority fee, z_{cd} and n_{cd} the counts of zero and non-zero calldata bytes, c_0, c_1 their compression-adjusted weights, σ a scalar, and $b_{\text{L1}}^{\text{blob}}$ the blob base fee. Since EIP-4844 [15] the second term is usually small but it is the term that scales with *transaction size*, which is why the EVM driver optimises calldata layout (Chapter 18) rather than opcode count.

4.2 The SVM as a machine

The SVM [16] is a register machine (SBPF) over a flat array of accounts, executing transactions in parallel wherever their declared account access sets do not conflict. Programs are stateless code; all mutable state lives in accounts that programs own.

Aspect	Behaviour
Word	64-bit registers (SBPF); 4 KiB stack frames; 32 KiB heap by default, requestable up to 256 KiB
State	Accounts: `(lamports, owner, data[], executable, rent_epoch)`; data up to 10 MiB; only the owning program may mutate `data`
Execution	Parallel across transactions with disjoint write sets; within a transaction, instructions run sequentially
Composition	Cross-program invocation (CPI), depth ≤ 4, with signer privileges extended via PDA seeds
Metering	Compute units; default 200 000 CU per instruction, up to 1 400 000 CU per transaction, requested via the compute-budget program
Fee market	5 000 lamports per signature, plus a priority fee in micro-lamports per CU with *local* (per-account) markets
Authority	Ed25519 signatures; PDAs are program-controlled addresses without a private key, signed for by seed derivation
Transaction shape	A packet of at most 1232 bytes carrying signatures, an account list, and instruction data; address lookup tables compress the account list

SVM · SOLANA

The SVM's defining property, for our purposes, is that a transaction must *declare its access set in advance*:

$$\text{tx} = \big(\{(a_i, w_i)\}_{i=1}^{m}, \text{instructions}\big), \quad w_i \in \{\text{r}, \text{w}\}. \tag{4.2}$$

This declaration is what enables parallelism, and it is also what makes the scheduler in Chapter 9 possible: the OS can compute the conflict graph before submission rather than discovering

> conflicts by failing. The EVM offers a weaker analogue through EIP-2930 access lists, which are advisory for pricing rather than binding for scheduling.

Two SVM constants recur in the design. Rent exemption sets the price of memory: an account with n bytes of data must hold at least

$$\text{lamports}_{\min}(n) = (128 + n) \cdot \rho \cdot \theta \tag{4.3}$$

lamports, where ρ is the per-byte-year rate (3 480 lamports) and $\theta = 2$ years is the exemption threshold — approximately 0.00089 SOL for an empty account and about 0.0007 SOL per additional 100 bytes. Unlike EVM storage, this is a *deposit*, refundable on account closure, which makes the SVM's memory model economically closer to a lease than to a purchase (Chapter 17).

The second constant is the 1232-byte packet limit, which bounds how much the kernel can batch into one SVM transaction and therefore appears directly in the scheduler's admission control (Chapter 16).

4.3 The honest comparison

Dimension	EVM / Base	SVM / Solana
Concurrency	None within a block; conflicts are resolved by ordering	Explicit; conflicts are resolved by declared access sets
State ownership	Contract owns its storage; no external account layout	Program owns accounts; layout is public and addressable
Memory cost	Purchase (20 000 gas/slot) with capped refund	Refundable deposit proportional to bytes
Composability	Synchronous calls, arbitrary depth 1024	CPI depth 4; deeper composition requires transaction-level composition
Failure semantics	Whole transaction reverts; gas consumed	Whole transaction fails; fee consumed; no partial state
Fee predictability	Global base fee, EIP-1559 dynamics	Local per-account markets; congestion is contention-specific
Identity	20-byte address, hex, checksummed	32-byte key, base58
Signature	secp256k1 ECDSA (and P-256 via verifier contracts)	Ed25519 natively; secp256k1 via precompile-equivalent programs
Account abstraction	EIP-7702 delegation, ERC-4337 pipeline	Native: fee payer may differ from actor; PDAs give program authority
Upgradeability	Proxy patterns; code hash may change silently	Explicit upgrade authority on program-data accounts

The table is not a scoreboard. It is a specification of what the hardware abstraction layer must hide, and the answer is: everything in it. A userland program that must know which column it is running in has already lost the property A4 exists to protect.

4.4 Addressing and naming

The OS needs one way to name an account, a program, an asset and an object, across both domains. We use a URI-shaped canonical name with a fixed binary encoding.

```
os://<domain>/<kind>/<id>[?<qualifier>]

os://base/acct/0x5b8b...c41d        an EVM account
os://sol/acct/9xQe...Vt2P           an SVM account
os://base/prog/0x00a1...9f02        a deployed contract
os://sol/prog/JUP6Lk...8dQ          a deployed program
os://base/asset/erc20:0xa0b8...eb48 USDC on Base
os://sol/asset/spl:EPjF...Dt1v      USDC on Solana
os://*/asset/usd                    the domain-independent asset class
```

The canonical binary form is `H(domain_tag ‖ kind_tag ‖ id_bytes)`, a 32-byte identifier used as the key in every kernel map, so that the same asset class resolves to the same identifier in both kernels. The `os://*/asset/usd` form is the one users see: it names the *thing* (a dollar), not its representation (a particular token contract on a particular chain), and the driver resolves it at execution time. This is the naming equivalent of a filesystem path versus a disk sector, and it is what makes “pay 20 dollars” expressible as an intent.

CHAPTER 5

When: Time, Ordering and Finality

There is no clock. There are three clocks, and only one of them is authoritative for authority.

5.1 Three clocks

Clock	Definition	What it may be used for
Wall clock t	Real time, as observed off-domain	Scheduling, timeouts, user-facing latency. **Never** for authorisation
Domain clock h_c	Block height (EVM) or slot (SVM), plus the domain-reported timestamp	Authorisation, expiry, rate limits — it is the only clock consensus agrees on
Causal clock $\prec$	A partial order over events across domains, established by observation records	Cross-domain ordering, reconciliation, replay protection

AXIOM 5.1 · A2′ — Authorisation is timed by the domain clock
Session expiry, capability expiry, timelocks and rate limits are evaluated against h_c and the domain-reported timestamp, never against off-domain time. The scheduler may use wall time to *decide when to act*; the kernel uses domain time to decide what is *allowed*.

The distinction is not pedantic. A validator can shift the reported timestamp within a tolerance; a relayer can delay a message arbitrarily. Any design that lets an off-domain clock gate authority has handed that authority to whoever controls the clock.

5.2 Finality is a function, not a flag

The single most common modelling error in cross-domain systems is treating "confirmed" as a boolean. We model it as a monotone confidence function.

DEFINITION 5.2 · Commitment level
For a domain c, an event e included at height h_e, and a current height h, the *commitment level* is a value $F_c(e, h) \in [0, 1]$, non-decreasing in h, giving the probability that e is permanent. We distinguish named thresholds:

$$\ell_0 \text{ seen} < \ell_1 \text{ included} < \ell_2 \text{ confirmed} < \ell_3 \text{ final} < \ell_4 \text{ settled} \tag{5.1}$$

For the SVM, ℓ_2 corresponds to optimistic confirmation (a supermajority of stake has voted), reached typically within one to two slots, and ℓ_3 to the rooted state some tens of slots later. For an OP-Stack L2 such as Base, ℓ_2 is a sequencer preconfirmation, ℓ_3 is inclusion in an L1 batch, and ℓ_4 is the expiry of the fault-proof window: an interval measured in days, not seconds.

A useful model for reorganisation risk on a probabilistic domain is the exponential tail [17], [17], [18], [18]

$$\Pr[\text{reorg depth} > d] \approx e^{-\lambda_c d} \Rightarrow F_c(e,h) \approx 1 - e^{-\lambda_c (h - h_e)}, \tag{5.2}$$

so the number of blocks a kernel must wait to reach a target confidence $1-\varepsilon$ is

$$d^* = \left\lceil \frac{\ln(1/\varepsilon)}{\lambda_c} \right\rceil. \tag{5.3}$$

The kernel does not hardcode d^*; it is a per-domain policy parameter, because the correct value for a 5 USD payment and a 500 000 USD settlement differ by an order of magnitude in ε and therefore by a constant factor in d^*.

> **INVARIANT 5.3 · Commitment monotonicity**
> A kernel state transition may depend on an event e only at a commitment level ℓ declared in advance by the syscall that produced it. If e is later reverted below ℓ, the dependent transition must be compensable — every syscall in Appendix A declares both its level and its compensation.

5.3 The latency budget

"Instant" is a design target with known thresholds [19]: below roughly 100 ms an interaction feels direct; below 1 s the user retains flow; beyond 10 s attention is lost. The kernel's job is to place the user's experience inside those bounds while the truth catches up behind them.

End-to-end latency for an intent decomposes as

$$L_{\text{e2e}} = L_{\text{intent}} + L_{\text{plan}} + L_{\text{sign}} + L_{\text{submit}} + L_{\text{incl}} + L_{\text{conf}}(\ell) + L_{\text{notify}} \tag{5.4}$$

Term	Typical	Who controls it
L_{intent}	< 5 ms	Shell: parse and validate locally
L_{plan}	5–40 ms	Planner: route search, quote fetch (cached)
L_{sign}	0 ms with a session; 300–2000 ms if the human must sign	Capability design — this is why sessions exist
L_{submit}	10–80 ms	Network path to leader / sequencer
L_{incl}	200 ms – 2 s	Domain: slot or block period
$L_{\text{conf}(\ell)}$	0.4 s – days	Choice of commitment level ℓ
L_{notify}	< 50 ms	Notifier subsystem

The two terms the OS can actually collapse are L_{sign} and L_{conf}. The first is collapsed by sessions (Chapter 28). The second is collapsed not by waiting less but by *rendering optimistically at a declared level*: the shell shows the effect at ℓ_2, marks it as provisional, and reconciles at ℓ_3. Because Eq. (5.2) gives the probability of being wrong, the interface can state its own confidence rather than lying by omission — which is A7 applied to time.

5.4 Ordering as a resource

Within a block, position has value: the same trade at index 3 and index 40 settles at different prices. Someone captures that difference. If the OS does not decide the policy, the policy is decided by whoever pays the builder most.

We define three ordering policies the scheduler may implement, and specify which one is the default.

First-seen-fair the scheduler orders intents by local arrival time and submits them in that order. Simple; vulnerable to network-level games.

Batch-uniform intents arriving within an epoch Δ are grouped and submitted as one batch with an internally randomised or price-uniform order, so that intra-batch position carries no value. This is the default for value-carrying intents; the batching interval is derived in Chapter 9.

Deadline-first intents carry deadlines; the scheduler orders by earliest deadline, subject to admission control. Used for time-sensitive system intents such as liquidations and expiries.

DEFINITION 5.4 · Extraction bound

For a batch B ordered uniformly at random, the expected value a self-interested builder can extract from position within B is bounded by the value of *inclusion* decisions only:

$$E[\text{extraction}(B)] \leq \sum_{i \in B} v_i \cdot \Pr[\text{exclusion}_i] \tag{5.5}$$

where v_i is the position-dependent value of intent i. Uniform ordering removes the intra-batch term; it does not remove censorship, which is a ring -1 problem.

5.5 The scheduler's clock

The kernel's scheduler runs on a tick derived from the fastest domain clock in play, not on wall time. Each tick it performs: admission control (does the intent's budget cover its estimated OSU cost?), conflict-graph construction over pending intents, batch formation, and submission. Chapter 16 gives the algorithm; here we record only the invariant that makes it analysable.

INVARIANT 5.5 · Tick determinism

Given the same pending set, the same domain heights and the same policy parameters, the scheduler produces the same batch. Randomness used for uniform ordering is drawn from a committed beacon, not from local entropy, so that any observer can replay the scheduler's decisions and verify that policy was followed.

CHAPTER 6

How: One Intent, End to End

A single trace through the whole system, so that the remaining thirty chapters have somewhere to attach.

6.1 The intent

A person types into the shell:

```
send 20 dollars to @mira, she likes sol
```

The shell resolves this to a structured intent. Note that nothing in it names a chain, a token contract, a bridge or a gas token — those are the kernel's concern, which is precisely A4.

```
{
  "kind": "transfer",
  "from": "os://*/acct/self",
  "to":   "os://*/acct/mira.osfun",
  "give": { "asset": "os://*/asset/usd", "amount": "20.00" },
  "want": { "asset": "os://sol/asset/native" },
  "deadline_ms": 30000,
  "max_cost_osu": 4200,
  "commitment": "confirmed"
}
```

6.2 The trace

os.fun — intent trace

```
STAGE                    WHAT HAPPENS                                          SPEC
─────────────────────────────────────────────────────────────────────────────────
 1 parse        shell → intent struct; local validation                       §19
 2 resolve      mira.osfun → os://sol/acct/9xQe…; usd → per-domain            §14
                asset set {base:USDC, sol:USDC}
 3 authorise    session capability checked: verb=transfer,                    §10
                cap≥20 USD, expiry>now, revocation absent
 4 price        planner quotes: (a) Base USDC → SOL via cross-domain          §11
                fill, (b) sell USDC on Base, bridge, swap on Solana
 5 plan         chosen plan = 2 domain calls + 1 settlement claim;            §13
                estimated cost 3 180 OSU < budget 4 200 OSU
 6 admit        scheduler admission control passes; intent enters             §13
                the pending set with deadline t+30 s
 7 batch        conflict graph built; intent batched with 6 others            §9
                touching disjoint accounts
 8 emit(evm)    EVM driver builds a 4337 UserOperation: escrow 20             §15
                USDC into osk-evm vault, emit IntentOpened
 9 fill         filler observes IntentOpened, sends 0.1183 SOL to             §11
                mira on Solana, submits fill proof to osk-svm
10 settle       osk-evm verifies the fill attestation at ℓ₂ and               §11
                releases escrow to the filler minus the fee
11 account      kernel writes the journal entry; balance view of              §29
                both accounts updated; revenue routed (§31)
12 notify       shell shows "sent · confirmed" at ℓ₂, reconciles              §28
                to ℓ₃ silently 12 s later
─────────────────────────────────────────────────────────────────────────────────
observed user latency to stage 12 rendering: ~ 900 ms
observed user actions required: one — typing the sentence
```

Figure 6.1 One intent, twelve kernel stages, two domains. Chapter references in the right column are where each stage is specified normatively.

6.3 What each stage costs

The point of a metered kernel is that this table exists at all. Every stage declares a resource cost before it runs (A5), so admission control is a comparison and not a hope.

Stage	Domain	Cost (OSU)	Dominated by
8 escrow	Base	1 950	`SSTORE` of the escrow record + ERC-20 transfer + L1 data
9 fill	Solana	430	Signature fee + 2 account writes
10 settle	Base	740	Attestation verify + escrow release
11 account	both	60	Journal append (packed, single slot)
total	—	3 180	Against a declared budget of 4 200

6.4 The same trace, from a Python program

The identical intent can originate from a userland program rather than from a human sentence. This is the userland API specified in Part IV; the kernel path below stage 3 is byte-identical.

```python
from osfun import app, wallet, assets, USD

@app.command("tip")
def tip(handle: str, amount: USD) -> None:
    """Send `amount` in dollars to `handle`, in whatever they prefer."""
    target = wallet.resolve(handle)              # syscall: ns_resolve
    quote  = assets.quote(USD(amount), target.preferred_asset)
    with app.budget(osu=4200):                   # syscall: proc_budget
        receipt = wallet.transfer(
            to=target,
            give=USD(amount),
            want=target.preferred_asset,
            commitment="confirmed",              # ℓ₂
        )                                        # syscall: val_transfer
    app.notify(f"sent {amount} to {handle} · {receipt.id}")
```

Three properties of this program are worth naming, because they are the thesis of Part IV in miniature. It is ordinary Python — no macro language, no domain-specific syntax. It never names a chain. And every line that has an effect is a syscall with a declared cost, so the program's total resource consumption is bounded before it runs, which is what allows it to execute inside a verifiable machine (Chapter 23) rather than inside a trusted server.

6.5 What follows

Part II builds the formal model that makes the trace above analysable rather than anecdotal: the abstract machine, resource normalisation, the conflict graph, the capability calculus and the commit protocol. Part III specifies the kernel and gives its two implementations. Part IV specifies the userland, OSVM, osPy and verification. Part V covers the system services, Part VI the economics, security and governance, and Part VII evaluation and horizon.

PART II

Formal Foundations

The model that makes the rest of this document checkable. We define the abstract machine of a single execution domain, the product machine over several, a resource unit that normalises their metering, the concurrency theory that lets the kernel schedule them, the calculus of authority, and the commit protocol that spans them. It ends with the finality calculus — a lattice of permanence, a type system over it, and a soundness theorem — which is where this document claims something new. Everything in Parts III and IV is an implementation of something defined here.

CHAPTER 7

The Abstract Machine and the Domain Product

One definition of a chain, instantiated twice, composed once.

7.1 The domain machine

DEFINITION 7.1 · Execution domain

An *execution domain* is a tuple

$$M_c = (\Sigma_c, \mathcal{T}_c, \delta_c, \Gamma_c, F_c, H_c) \tag{7.1}$$

where

- Σ_c is the *state space*, a set of partial maps from addresses to account records;
- $\mathcal{T}_c$ is the set of well-formed *transactions*;
- $\delta_c : \Sigma_c \times \mathcal{T}_c \to \Sigma_c \times \mathcal{R}_c$ is the *transition function*, total and deterministic, returning a new state and a receipt $r \in \mathcal{R}_c$;
- $\Gamma_c : \Sigma_c \times \mathcal{T}_c \to \mathbb{N}$ is the *metering function*, the resource consumption of a transaction in the domain's native unit;
- $F_c : \mathcal{R}_c \times \mathbb{N} \to [0,1]$ is the *finality function* of Definition 5.2;
- $H_c \subseteq \mathbb{N}$ is the *height* domain, ordered, with a successor relation given by consensus.

Two properties of δ_c are assumed throughout and are true of both target domains:

AXIOM 7.2 · Determinism

δ_c is a pure function. Given (σ, t), every honest node computes the same (σ', r). All non-determinism (time, randomness, external price) enters as data inside σ or t.

AXIOM 7.3 · Atomicity

$\delta_c(\sigma, t) = (\sigma', r)$ with $\sigma' = \sigma$ whenever r is a failure receipt, except for fee deduction and nonce advance. There is no partial application of a transaction's effects.

Atomicity is the property that removes an entire class of distributed-systems complexity inside a domain — and, by its absence across domains, creates Chapter 11.

The EVM instantiation

Σ_{evm} maps a 20-byte address to a record (n, b, s, h) — nonce, balance, storage map $s : \{0,1\}^{256} \to \{0,1\}^{256}$, and code hash h. A transaction is $t = (\text{nonce}, \text{gasLimit}, \text{maxFee}, \text{maxPriorityFee}, \text{to}, \text{value}, \text{data}, \text{accessList}, \sigma_{\text{sig}})$, and

$$\Gamma_{\text{evm}}(\sigma, t) = g_{\text{intrinsic}}(t) + \sum_{i \in \text{ trace}} g_i(\sigma_i) \tag{7.2}$$

where $g_{\text{intrinsic}} = 21000 + 4z + 16n_z + g_{\text{access}}$ and g_i is the state-dependent price of the i-th opcode (EIP-2929 makes g_i depend on whether the accessed slot is warm). The height domain is block number; the successor relation is produced by the sequencer, with settlement to L1 determining F.

The SVM instantiation

Σ_{svm} maps a 32-byte address to a record (λ, o, d, x, e) — lamports, owner program, data bytes, executable flag, rent epoch. A transaction is $t = (\{\sigma_{\text{sig}}\}, \{(a_i, w_i)\}, \{\text{ix}_j\})$: signatures, a declared account access list with read/write flags, and instructions. Then

$$\Gamma_{\text{svm}}(\sigma, t) = \underbrace{5000 \cdot |\{\sigma_{\text{sig}}\}|}_{\text{signature fee}} + \underbrace{\sum_j \text{cu}_j}_{\text{compute units}} \tag{7.3}$$

subject to $\sum_j \text{cu}_j \le C_{\text{tx}} = 1.4 \times 10^6$ and, per written account a, $\sum_{t:a\in W(t)} \text{cu}(t) \le C_{\text{acct}}$ within a block.

The structural difference that matters for the OS is not the unit but the *declaration*: an SVM transaction states $W(t)$ and $R(t)$ before execution, while an EVM transaction discovers them during execution.

DEFINITION 7.4 · Access sets

For $t \in \mathcal{T}_c$ let $R(t)$ and $W(t)$ denote the sets of addresses read and written. For the SVM these are declared and binding. For the EVM they are determined by execution; the OS computes a conservative superset $\widehat{W}(t) \supseteq W(t)$ by static analysis of the intended call path, and uses an EIP-2930 access list to pin the pricing.

7.2 The product machine

The OS does not run on one domain. Its state is distributed across all of them, plus whatever it can observe.

DEFINITION 7.5 · Product machine

Let C be a finite set of domains. The *OS machine* is

$$\mathcal{M} = \left(\prod_{c \in C} \Sigma_c, \mathcal{I}, \Delta \right) \tag{7.4}$$

where $\mathcal{I}$ is the set of *intents* (Chapter 16) and Δ maps an intent and a product state to a set of domain transactions, one or more per domain, together with a settlement obligation.

Two facts make $\mathcal{M}$ harder than a product of state machines.

First, **the OS cannot observe the product state**. It observes each domain through a channel with delay and with a commitment level below 1. Write $\tilde{\sigma}_c(t)$ for the OS's belief about domain c at wall time t; then

$$\tilde{\sigma}_c(t) = \sigma_c(h_c(t - \varepsilon_c)) \tag{7.5}$$

for an observation lag ε_c (RPC round trip plus propagation, typically 50–400 ms) and a height function h_c. Every decision the kernel makes is made on stale data by construction, so every decision must be either idempotent or compensable.

Second, **the domains do not agree on time**. There is no global clock, so the only order the OS can rely on across domains is causal [20]: event e precedes e' if e' consumed data attributable to e.

DEFINITION 7.6 · OS-consistency

An execution of $\mathcal{M}$ is *consistent* if there exists, for each domain c, a serial order $\prec_c$ over the transactions committed in c such that (i) $\prec_c$ agrees with c's block order, (ii) the union $\prec = \cup_c \prec_c \cup \prec_{\text{causal}}$ is acyclic, and (iii) every kernel invariant holds in each state reachable by a prefix of $\prec$.

Condition (iii) is the one the kernel enforces directly, and Chapter 10 gives the invariants: capability soundness, escrow conservation, budget non-negativity, and journal monotonicity.

7.3 Syscalls as partial functions

DEFINITION 7.7 · Syscall

A *syscall* is a partial function

$$s : \mathcal{C} \times \mathcal{A} \times \prod_c \Sigma_c \rightharpoonup \prod_c \Sigma_c \times \mathcal{R} \tag{7.6}$$

with a declared precondition pre_s, postcondition post_s, cost bound $\gamma_s \in \mathbb{N}$ (in OSU, Chapter 8), commitment level ℓ_s, and compensation $\overline{s}$ such that

$$\overline{s} \circ s = \text{id} \quad \text{on the state components} \quad s \quad \begin{matrix}\text{modifies,} \\ \text{whenever}\end{matrix} \quad s \text{ has not yet reached } \ell_3. \tag{7.7}$$

Requiring every syscall to declare a compensation is unusual and is forced by Eq. (7.5): a kernel operating on stale observations will occasionally act on a state that no longer exists, and the only alternatives are to wait for finality (destroying latency) or to be able to undo (destroying nothing). Appendix A lists $(\gamma_s, \ell_s, \overline{s})$ for all 41 syscalls.

7.4 The soundness theorem

We can now state the property that justifies calling this an operating system rather than a library.

THEOREM 7.8 · Kernel soundness

Let A be the set of accounts holding user assets or OS authority, and suppose

1. for every $a \in A$ and every domain c, the mutation authority of a is the kernel program K_c (EVM: a is a kernel-owned vault or an EOA whose 7702 delegate is K_{evm}; SVM: a's `owner` is K_{svm}, or a is a PDA of K_{svm});
2. every entry point of K_c begins with `cap_check` (Chapter 10) and aborts on failure;
3. `cap_check` is sound: it returns `Ok` only if the presented capability is derivable from a root grant signed by the account's principal and is not expired or revoked.

Then every state transition affecting A is the image of a syscall authorised by a capability of the account's principal. In particular, no userland program, service, filler or scheduler can move assets without such a capability.

PROOF. Fix a transition τ affecting $a \in A$ in domain c. By (1), the only code path able to mutate a is K_c: on the SVM the runtime rejects writes by any program other than the owner, and on the EVM the vault's storage is mutable only by its own code while a 7702-delegated EOA executes only its delegate's code. Hence τ arose from an invocation of some entry point f of K_c. By (2), f executed `cap_check` with some capability κ and did not abort, so `cap_check` (κ) returned `Ok`. By (3), κ is derivable from a root grant signed by the principal of a. Finally, every entry point of K_c implements exactly

one syscall of Appendix A, so τ is that syscall's image. The three hypotheses are discharged by construction in Chapters 20 and 21 and are the subject of the conformance tests of Chapter 37. □

The theorem's value is in its hypotheses. It fails if a vault can be owned by anything but the kernel (hypothesis 1), if any entry point skips the check (hypothesis 2), or if capability derivation can be forged (hypothesis 3). Those three lines are therefore the entire audit surface of the OS, which is the point of having a kernel at all.

7.5 Reorganisation and convergence

A domain may retract a transition that the OS already acted on. Let σ_c evolve and let ρ be a reorganisation event replacing a suffix of the chain of depth d.

DEFINITION 7.9 · Compensable transition
A kernel transition τ is *compensable* if there is a kernel transition $\overline{\tau}$ that restores all invariants when applied after a retraction of τ, and $\overline{\tau}$ is idempotent: $\overline{\tau} \circ \overline{\tau} = \overline{\tau}$.

THEOREM 7.10 · Convergence under reorganisation
Suppose every kernel transition executed at a commitment level below ℓ_3 is compensable, that the OS applies compensations in causal order, and that the reorganisation depth distribution satisfies Eq. (5.2) with $\lambda_c > 0$. Then the OS state converges almost surely to the state implied by the canonical chain, and the expected number of compensations per transition is bounded by $e^{-\lambda_c d_{\text{policy}}} / \left(1 - e^{-\lambda_c}\right)$, where d_{policy} is the depth the kernel waits before acting.

PROOF. Each transition executed at depth d_{policy} requires compensation exactly when a reorganisation exceeds that depth, an event of probability $p = e^{-\lambda_c d_{\text{policy}}}$ by Eq. (5.2). Compensations are themselves transitions and may in turn be reverted, giving a geometric series with ratio $e^{-\lambda_c}$, whence the bound. Idempotence ensures repeated application is harmless; causal ordering ensures no compensation is applied before the transition it compensates. Since $p < 1$, the process terminates with probability 1. □

In practice d_{policy} is chosen per syscall class: a social post is written at ℓ_1 and compensated by deletion; a payment is released at ℓ_2; a settlement that pays a filler waits for ℓ_3 because its compensation would require clawing back value from a third party, which is not a kernel power.

INVARIANT 7.11 · Compensation locality
A kernel transition may be executed below ℓ_3 only if its compensation affects state the kernel controls. No compensation may depend on the cooperation of an external party.

CHAPTER 8

Resource Accounting: Gas, Compute Units and the OSU

Two metering systems, one schedulable unit, and the control loop that prices it.

8.1 Why a unified unit is necessary

A scheduler must compare. It has to answer questions of the form: is it cheaper to execute this intent on Base or on Solana? Does this program's budget cover three syscalls on one domain or two on the other? Neither gas nor compute units can answer these, because they measure different physical things in different currencies with different market dynamics.

We therefore define an internal unit, the **OSU** (os.fun system unit), with two conversion layers: a *physical* layer that maps a domain's metering unit to OSU using a fixed benchmark, and an *economic* layer that maps OSU to money using observed fee markets.

DEFINITION 8.1 · OSU
Fix a reference workload W_{ref}: a signature verification, a 32-byte state write, and 1 KiB of hashing. Define 1 OSU as $1/1000$ of the resource cost of W_{ref} on a reference domain. The physical conversion for domain c is the linear map

$$\varphi_c : \text{native}_c \to \text{OSU}, \quad \varphi_c(u) = u/\kappa_c, \quad \kappa_c = \Gamma_c(W_{\text{ref}})/1000. \tag{8.1}$$

Because $\Gamma_c(W_{\text{ref}})$ is measurable and stable, κ_c changes only when a domain reprices its opcodes — an event that occurs at hard forks and is handled as a kernel parameter update (Chapter 35). Representative values:

Domain	$\Gamma_c(W_{\text{ref}})$	κ_c	Meaning
Base (EVM)	≈ 47 000 gas	47	1 OSU ≈ 47 gas
Solana (SVM)	≈ 12 000 CU	12	1 OSU ≈ 12 CU

The economic layer is a price, not a constant:

$$\pi_c(t) = (b_c(t) + p_c(t)) \cdot \kappa_c \cdot P_c^{\text{tok}}(t) \tag{8.2}$$

where b_c is the base fee, p_c the priority fee the kernel intends to pay, and P_c^{tok} the token price in the accounting numéraire. The scheduler compares domains by π_c, budgets by φ_c, and bills users in the asset they hold (§8.6).

8.2 The EVM fee market as a control loop

EIP-1559 [9], [21] is a multiplicative controller. With target G^* and observed usage G_n in block n:

$$b_{n+1} = b_n \left(1 + \frac{1}{8} \cdot \frac{G_n - G^*}{G^*}\right) \tag{8.3}$$

This is a discrete-time system with a fixed point at $G_n = G^*$. Linearising around it, a step change in demand decays geometrically with ratio $|1 - 1/8 \cdot \eta|$ where η is the demand elasticity; the practical

consequence for the kernel is the *adjustment time*: the base fee can at most multiply by $9/8$ per block, so from b_0 to b^* takes

$$n_{\text{adj}} = \left\lceil \frac{\ln(b^*/b_0)}{\ln(9/8)} \right\rceil \approx 8.5 \ln(b^*/b_0) \tag{8.4}$$

blocks. On a 2-second L2 this is roughly $17 \ln(b^*/b_0)$ seconds — meaning a tenfold fee increase takes about 40 seconds to materialise. The kernel exploits this: a quote issued now is valid for a bounded window, and the scheduler's fee reserve (§8.5) is sized from Eq. (8.4) rather than guessed.

8.3 The SVM fee market is local

Solana prices priority per *account*, not globally: contention for a hot account raises the fee required to be scheduled against that account, while the rest of the block is unaffected. The kernel therefore estimates a per-account fee rather than a per-block one.

Let $q_a(t)$ be the empirical distribution of prioritisation fees (in micro-lamports per CU) among transactions writing account a over the last m slots. The kernel's estimator is a quantile with a congestion-dependent level:

$$\hat{p}_a = Q_{q_a}(1 - \varepsilon_{\text{miss}}), \quad \varepsilon_{\text{miss}} = \text{target inclusion miss rate per slot} \tag{8.5}$$

and the expected number of slots to inclusion under an i.i.d. assumption is $1/(1-\varepsilon_{\text{miss}})$. Choosing $\varepsilon_{\text{miss}} = 0.1$ gives 1.11 slots (≈ 444 ms) expected, which is what the latency budget of §5.3 assumes. Chapter 16 gives the estimator's implementation, including the exponentially weighted update that makes it robust to a single congested slot.

> **SVM · SOLANA**
> Local fee markets have an architectural consequence the OS must respect: a hot account is a *throughput* limit, not only a cost. With a per-account write budget C_{acct} per block and a per-write cost γ_w, the maximum sustained write rate to a single account is
>
> $$\mu_a = C_{\text{acct}} / (\gamma_w \cdot \Delta_{\text{slot}}). \tag{8.6}$$
>
> With $C_{\text{acct}} = 12 \times 10^6$ CU, $\gamma_w = 20000$ CU and $\Delta_{\text{slot}} = 0.4$ s, $\mu_a = 1500$ writes per second. Designs that funnel all users through one account hit this wall; Chapter 17 specifies the shard layout that avoids it.

8.4 Deriving the syscall cost table

Every syscall's declared cost γ_s is derived, not assumed. The derivation is mechanical: enumerate the state accesses and the cryptographic operations, price them by the domain's schedule, convert with Eq. (8.1), and add a safety margin from measured variance.

Syscall	EVM (gas)	SVM (CU)	OSU (max)
`cap_check` (warm, no delegation chain)	3 400	2 900	242
`cap_check` (cold, 2-level chain)	9 800	5 400	450
`val_transfer` (ERC-20 / SPL, warm)	29 500	17 000	1 417
`val_transfer` (native)	9 000	4 500	375
`proc_spawn`	46 000	26 000	2 167
`ns_resolve` (cached)	2 200	1 800	150

`evt_emit` (1 topic, 64 bytes)	2 100	1 200	100
`set_commit` (settlement claim)	58 000	31 000	2 583

The OSU column is $\max(\varphi_{\text{evm}}(\text{gas}), \varphi_{\text{svm}}(\text{cu}))$, so a program budgeted in OSU can execute on either domain without re-planning. This is A4 expressed in arithmetic: portability requires that the abstract cost be an upper bound over implementations.

8.5 Budgets and admission control

DEFINITION 8.2 · Process budget

A process P carries a budget $B_P \in \mathbb{N}$ (OSU) and a reserve factor z. A syscall s is admitted iff

$$\hat{\gamma}_s + z\hat{\sigma}_s \le B_P - \text{committed}(P) \tag{8.7}$$

where $\hat{\gamma}_s$ is the estimated cost and $\hat{\sigma}_s$ its standard deviation over the last m executions.

Choosing $z = 3$ bounds the probability of a mid-execution budget exhaustion by Chebyshev's inequality at $1/z^2 \approx 11\%$ for arbitrary distributions, and far lower for the near-deterministic costs typical of kernel paths. Because Eq. (8.7) is checked before execution rather than during, the failure mode is a refused syscall with a clear error (`E_BUDGET`, Appendix B) instead of a half-finished intent — the onchain analogue of `ENOMEM` at allocation time rather than an OOM kill mid-write.

```rust
/// Admission control. Runs in the scheduler (ring 1) before any domain
/// transaction is built, and again inside the kernel (ring 0) as a check.
pub fn admit(proc: &Process, call: SyscallId, est: &CostModel) -> Result<u64> {
    let (mu, sigma) = est.estimate(call, proc.domain);
    let need = mu.saturating_add(est.reserve_factor().saturating_mul(sigma));
    let left = proc.budget_osu.saturating_sub(proc.committed_osu);
    if need > left {
        return Err(Error::Budget { need, left });
    }
    Ok(need)
}
```

8.6 Who pays, and in what

The user holds USDC on Base. The intent requires SOL for fees on Solana. The kernel must not answer this with "acquire SOL first"; that is the failure Chapter 1 opens with.

Fee abstraction is implemented differently in each domain but presented identically:

EVM · BASE

A paymaster (ERC-4337) or a 7702-delegated sponsor pays the domain fee and is reimbursed from the user's token balance inside the same transaction. The kernel's paymaster validates that (i) the user has a `fee` capability covering the amount, (ii) the token's oracle price is fresh, and (iii) the reimbursement leaves the paymaster whole after slippage.

SVM · SOLANA

The transaction's fee payer is simply a different account from the actor — a native capability of the SVM that requires no extra protocol. The kernel's relayer signs as fee payer, and an

> instruction earlier in the same transaction transfers the reimbursement, so the atomicity axiom guarantees the relayer is never left unpaid.

In both cases the user is quoted a single number in the asset they hold. The spread the kernel charges must cover the fee, the price risk over the quote window T_q, and the inventory cost:

$$\text{spread} = \pi_c + \underbrace{z\sigma_{\text{tok}}\sqrt{T_q}v}_{\text{price risk}} + \underbrace{rvT_q}_{\text{carry}} \tag{8.8}$$

with v the notional, σ_{tok} the token's volatility, and r the funding rate. For a 20 USD payment with $T_q = 30$ s, $\sigma_{\text{tok}} = 60\%$ annualised and $z = 3$, the price-risk term is under one cent — which is why quote windows are short and why the kernel prefers stable assets as the reimbursement leg.

8.7 Metering the userland

Userland programs (Part IV) execute off-domain on OSVM, so their cost is not a domain fee but a compute cost that someone must bear and that must be bounded in advance for the same reason as Eq. (8.7).

$$\gamma_{\text{osvm}}(p) = \alpha \cdot \text{cycles}(p) + \beta \cdot \text{mem}_{\text{peak}}(p) + \sum_{s \in \text{ syscalls}(p)} \gamma_s \tag{8.9}$$

with α calibrated so that 10^6 RV64IM cycles ≈ 1 OSU, and β pricing peak Merkleised memory in pages. The first two terms are paid to whoever executes and proves; the third is the domain cost of the program's effects. Chapter 23 defines `cycles` precisely, Chapter 27 prices proving, and Chapter 34 says who pays.

CHAPTER 9

Concurrency, Conflict Graphs and Serializability

The SVM hands the scheduler its conflict graph. The EVM makes it guess. Both cases reduce to the same batching problem.

9.1 Conflicts

DEFINITION 9.1 · Conflict
Two transactions t_1, t_2 *conflict*, written $t_1 \bowtie t_2$, if

$$W(t_1) \cap (W(t_2) \cup R(t_2)) \neq \varnothing \quad \text{or} \quad W(t_2) \cap (W(t_1) \cup R(t_1)) \neq \varnothing. \tag{9.1}$$

The *conflict graph* of a pending set T is $G_T = (T, \{(t_1, t_2) : t_1 \bowtie t_2\})$.

DEFINITION 9.2 · Parallel schedule
A *schedule* is a sequence of rounds $S = (B_1, \ldots, B_k)$ partitioning T. It is *conflict-free* if each B_i is an independent set in G_T.

THEOREM 9.3 · Serializability of conflict-free schedules
Every conflict-free schedule S of T produces the same final state as some serial execution of T, for any deterministic δ_c satisfying the determinism and atomicity axioms.

PROOF. Build the precedence graph P over T: draw $t \to t'$ if t precedes t' in S and they conflict. Since each round is an independent set, conflicting pairs never share a round, so every edge of P points strictly forward in round index; hence P is acyclic. Take any topological order $\prec$ of P. Executing T serially in $\prec$ touches each conflicting pair in the same relative order as S; non-conflicting pairs commute, because by definition neither reads or writes state the other writes, and δ_c is a pure function of the state it reads. By induction on rounds, the states agree after every round, hence at the end. □

This is the classical result [22], [23], and it is exactly what the SVM's runtime relies on. Its relevance to the OS is that the kernel can *construct* conflict-free batches ahead of submission, so that the domain's own scheduler never has to reject our transactions for lock contention.

9.2 The EVM as the degenerate case

On the EVM every transaction in a block is ordered against every other, so the conflict graph is effectively complete and no parallelism is available at the domain level. The kernel's lever is different: *fusion*. Rather than submitting n transactions it submits one that performs n operations.

The saving is exact and large:

$$\text{gas}(\text{fused}) = 21000 + \sum_{i=1}^{n} \left(g_i - g_i^{\text{warm-saving}} \right) \quad \text{versus} \quad \text{gas}(\text{separate}) = \sum_{i=1}^{n} (21000 + g_i) \tag{9.2}$$

so fusion saves $(n-1)\cdot 21000$ in intrinsic cost alone, plus cold-access savings of up to 2100 per repeated slot and 2600 per repeated account. For a batch of eight transfers touching a shared token contract, measured saving is approximately

$$7 \cdot 21000 + 7 \cdot 2600 + 5 \cdot 2100 \approx 175000 \text{ gas} \quad (\approx 3720 \text{ OSU}), \tag{9.3}$$

a 38% reduction against separate submission. On an L2 the calldata term of Eq. (4.1) partially offsets this, which is why the EVM driver also compresses the fused calldata (Chapter 18).

9.3 Batch formation

DEFINITION 9.4 · Batching problem
Given a pending set T with conflict graph G_T, per-transaction resource $\gamma(t)$, per-round capacity C, and deadlines $d(t)$, find a schedule S minimising the number of rounds subject to (i) each round is an independent set, (ii) $\sum_{t\in B_i}\gamma(t) \le C$, and (iii) every t is placed in a round completing before $d(t)$.

Without the capacity and deadline constraints this is graph colouring, which is NP-hard and inapproximable within $n^{1-\varepsilon}$. The kernel therefore uses a greedy algorithm with a provable bound rather than an optimal one.

PROPOSITION 9.5 · Greedy bound
Greedy colouring in any vertex order uses at most $\Delta(G_T)+1$ rounds, where Δ is the maximum degree; with the capacity constraint the bound becomes $\max\!\left(\Delta+1, \left\lceil \sum_t \gamma(t)/C \right\rceil\right)$. Ordering vertices by descending degree (Welsh–Powell [24]) additionally guarantees at most $\max_i \min(d_i+1, i)$ rounds where d_i is the degree of the i-th vertex in that order.

The bound is a worst case over the graph; the corresponding bound over the *work* is Brent's [25]: with total work $W=\sum_t\gamma(t)$, per-round capacity C and critical path D rounds, any greedy schedule finishes within $W/C + D$ rounds, which is the form the scheduler actually uses when it reports an estimated completion slot.

In practice Δ is small: user intents mostly touch the user's own accounts, and the hot shared objects (a token mint, an AMM pool) are exactly the ones Chapter 17 shards. The measured degree distribution in our simulation harness (Chapter 36) is heavy-tailed with $\Delta \approx 12$ at the 99th percentile and a median of 1.

```rust
/// Greedy conflict-free batching with capacity and deadline constraints.
/// Deterministic: ties break on the intent id, so any observer replaying the
/// same pending set derives the same batches (Invariant 5.5).
pub fn form_batches(pending: &[Intent], cap_osu: u64, now: Slot) -> Vec<Batch> {
    let mut order: Vec<usize> = (0..pending.len()).collect();
    order.sort_by_key(|&i| {
        (pending[i].deadline, core::cmp::Reverse(pending[i].degree), pending[i].id)
    });

    let mut batches: Vec<Batch> = Vec::new();
    'next: for i in order {
        let it = &pending[i];
        for b in batches.iter_mut() {
            let conflicts = b.writes.intersects(&it.reads_writes())
                || b.reads.intersects(&it.writes);
            if !conflicts && b.cost_osu + it.cost_osu <= cap_osu {
```

```
                b.push(it);
                continue 'next;
            }
        }
        let mut b = Batch::new(now);
        b.push(it);
        batches.push(b);
    }
    batches
}
```

9.4 How long to wait before submitting

Batching trades latency for cost. Let intents arrive as a Poisson process with rate λ, let the fixed cost of a domain call be γ_0 and the marginal cost per intent γ_1. If the scheduler waits Δ before submitting, the expected batch size is $\lambda\Delta$ and the amortised cost per intent is

$$c(\Delta) = \frac{\gamma_0}{\lambda\Delta} + \gamma_1, \tag{9.4}$$

while the expected added latency is $\Delta/2$. Minimising cost subject to a latency budget $L_{\max}$ gives the corner solution $\Delta^* = 2L_{\max}$, but the interesting regime is the marginal one: the cost saved by waiting an extra unit of time is $\gamma_0/(\lambda\Delta^2)$, so waiting is worthwhile only while

$$\frac{\gamma_0}{\lambda\Delta^2} > w, \tag{9.5}$$

where w is the user's implied cost of one unit of delay. With $\gamma_0 = 21000$ gas ≈ 447 OSU, $\lambda = 20$ intents/s and w calibrated so that one second of latency is worth 50 OSU, Eq. (9.5) gives $\Delta^* \approx 0.67$ s — comfortably inside the one-second flow threshold and consistent with the 2-second block period of the EVM domain, so the practical rule is **batch for at most one block period, then submit**.

9.5 Contention and queueing

When many intents target the same hot account, the constraint is not the conflict graph's chromatic number but the account's service rate μ_a derived in §8.3. Modelling the hot account as an M/D/1 queue with arrival rate λ and deterministic service $1/\mu_a$, the expected waiting time is

$$W = \frac{\rho}{2\mu_a(1-\rho)}, \quad \rho = \frac{\lambda}{\mu_a}. \tag{9.6}$$

The design consequence is sharp: latency is finite only while $\rho < 1$, and grows without bound as $\rho \to 1$. Since μ_a is fixed by the domain, the only lever is to increase the number of accounts. Sharding a hot object into k shards multiplies capacity by k and reduces ρ to $\lambda/(k\mu_a)$; Chapter 17 specifies the shard-selection function and the reconciliation that keeps the shards' union consistent.

CHAPTER 10

The Capability Calculus

Authority is an object. It may be delegated. It may never grow.

10.1 Capabilities

DEFINITION 10.1 · Capability

A *capability* [4], [26] is a record

$$\kappa = (i, s, o, V, \Phi, e, n, \pi) \tag{10.1}$$

where i is a unique id, s the subject (who may exercise it), o the object (what it acts on, an `os://` name), $V \subseteq \mathcal{V}$ a set of verbs, Φ a conjunction of constraints, e an expiry in domain time, n a monotonic nonce, and π the parent capability id (or $\bot$ for a root grant). Its authority is the set of syscall invocations it permits:

$$[\![\kappa]\!] = \{(s, v, o, a) : v \in V, \Phi(a) \text{ holds}, h_c \le e\}. \tag{10.2}$$

Constraints Φ are drawn from a small, decidable language — the whole language is given in Appendix C, but the shape is:

```
Φ ::= amount ≤ N            spend cap per invocation
    | rate(N, W)            at most N invocations per window W
    | total ≤ N             cumulative cap over the capability's life
    | recipient ∈ {a…}      allow-list
    | oracle(price) ± bps   execution-price band
    | Φ ∧ Φ
```

Decidability matters: constraint evaluation happens inside a domain transaction, where an unbounded language would be a denial-of-service vector and an unpredictable cost. Every constructor above evaluates in $O(1)$ state reads with a known gas and CU cost, which is what makes the `cap_check` row of the table in §8.4 a constant.

10.2 Attenuation

DEFINITION 10.2 · Attenuation order

$\kappa' \sqsubseteq \kappa$ (κ' is an *attenuation* of κ) iff

$$V' \subseteq V, \quad o' \subseteq o, \quad \Phi' \Rightarrow \Phi, \quad e' \le e, \quad \text{and} \quad [\![\kappa']\!] \subseteq [\![\kappa]\!]. \tag{10.3}$$

PROPOSITION 10.3 · Lattice structure

$(\mathcal{K}, \sqsubseteq)$ is a meet-semilattice: any two capabilities on the same object have a greatest lower bound $\kappa_1 \cap \kappa_2$ with verb set $V_1 \cap V_2$, constraint $\Phi_1 \wedge \Phi_2$ and expiry $\min(e_1, e_2)$. A join need not exist, because the union of two constraint sets is not generally expressible in the constraint language — and this asymmetry is deliberate: the system can always narrow authority and can never silently widen it.

THEOREM 10.4 · Non-amplification
Let κ_0 be a root grant and let $\kappa_0 \to \kappa_1 \to \ldots \to \kappa_n$ be a delegation chain where each step is produced by the kernel's `cap_delegate` syscall. Then $[\![\kappa_n]\!] \subseteq [\![\kappa_0]\!]$, and the total value transferable under the chain never exceeds κ_0's cap.

PROOF. By induction on n. For $n = 0$ the statement is trivial. Assume it for $n-1$. `cap_delegate` constructs κ_n from κ_{n-1} and requires $V_n \subseteq V_{n-1}$, $o_n \subseteq o_{n-1}$, $e_n \le e_{n-1}$ and $\Phi_n \Rightarrow \Phi_{n-1}$, the last checked syntactically by requiring Φ_n to be Φ_{n-1} conjoined with additional clauses (Chapter 21's implementation stores the constraint list and appends only). Therefore $\kappa_n \sqsubseteq \kappa_{n-1}$, and $\sqsubseteq$ is transitive, so $\kappa_n \sqsubseteq \kappa_0$; monotonicity of $[\![\cdot]\!]$ gives the inclusion. For value: the cumulative constraint `total ≤ N` is accounted against the *root* capability's counter, not the child's, so parallel delegations share one budget rather than each receiving a fresh one. □

The final clause of the proof is where most real systems leak. If a delegation copies the parent's spend cap into a fresh counter, then k delegations multiply the authority by k. The kernel's storage layout (Chapter 21) makes this structurally impossible by keeping the counter in the root record and having children hold only a pointer.

10.3 Revocation in constant time

Naive revocation requires walking the delegation tree, which is unbounded work inside a transaction. We use epoch indirection.

DEFINITION 10.5 · Revocation epoch
Each principal α has an epoch counter $E(\alpha)$. A capability records the epoch E_κ at which it was minted. `cap_check` requires $E_\kappa = E(\alpha)$. `cap_revoke_all` increments $E(\alpha)$, invalidating every outstanding capability in $O(1)$. Selective revocation additionally consults a per-capability tombstone bit.

PROPOSITION 10.6 · Revocation soundness and cost
After `cap_revoke_all` at height h, no capability minted before h passes `cap_check` at any height $h' \ge h$. The operation costs one storage write (≈ 2900 gas / ≈ 1200 CU) regardless of the number of outstanding capabilities, and selective revocation costs one bit-set write.

The trade is that bulk revocation is coarse: it invalidates good capabilities along with bad ones, and the principal must re-mint. Given that revocation is an emergency operation and re-minting is a session-establishment flow that takes one signature, this is the correct trade — the alternative, unbounded tree-walking, is a transaction that can run out of gas exactly when the user most needs it to succeed.

10.4 Encoding and verification cost

The wire encoding is fixed at 96 bytes plus constraints, chosen so that a capability with two constraints fits in three EVM storage slots and one SVM account of 128 bytes (rent-exempt cost by Eq. (4.3): $(128 + 128) \times 6960 = 1.78 \times 10^6$ lamports ≈ 0.0018 SOL, refundable).

```
  offset  size  field
  ──────────────────────────────────────────────────────────────
    0      16   id           (uuid-v7: sortable by mint time)
   16      32   subject      (principal or code identity hash)
   48      32   object       (canonical os:// hash)
   80       4   verbs        (bitset over 32 verb classes)
   84       6   expiry       (domain height, 48-bit)
   90       2   epoch        (principal revocation epoch)
   92       4   flags        (delegable, transferable, one-shot, …)
   96       –   constraints  (TLV list, ≤ 4 entries on the hot path)
```

Verification is a fixed sequence: recompute the id, load the root record, compare epochs, check the tombstone bit, evaluate constraints, and verify one signature if the caller is a session key. The cost table of §8.4 follows directly.

10.5 Instantiation on the EVM

```
/// @notice Capability check as executed by osk-evm on every entry point.
/// @dev    Constant-cost: one cold SLOAD for the root record, one warm SLOAD
///         per constraint, one ecrecover when a session key is presented.
function capCheck(Cap calldata k, bytes calldata sig, Call calldata call)
    internal
    view
    returns (address principal)
{
    bytes32 id = keccak256(abi.encode(k.subject, k.object, k.verbs, k.expiry,
                                      k.epoch, k.nonce, k.parent));
    if (id != k.id) revert BadCapId();

    Root storage r = _roots[k.subject];
    if (k.epoch != r.epoch) revert Revoked();            // O(1) bulk revoke
    if (_tombstones[k.id]) revert Revoked();             // selective revoke
    if (block.number > k.expiry) revert Expired();
    if (!_verbAllowed(k.verbs, call.verb)) revert VerbNotGranted();

    // Constraint evaluation: total ≤ N is charged against the ROOT counter,
    // which is what makes Theorem 10.4 hold under parallel delegation.
    uint256 n = k.constraints.length;
    for (uint256 i; i < n; ++i) {
        _evalConstraint(k.constraints[i], call, r);
    }

    // Session keys sign the call; root credentials are the msg.sender.
    if (k.flags & FLAG_SESSION != 0) {
        bytes32 digest = _hashTypedData(call);
        if (ECDSA.recover(digest, sig) != address(bytes20(k.subject))) {
            revert BadSignature();
        }
    } else if (msg.sender != address(bytes20(k.subject))) {
        revert NotSubject();
    }

    // Program subjects are bound to their code hash: an upgrade voids the
    // capability rather than silently inheriting it (Invariant 3.4).
    if (k.flags & FLAG_PROGRAM != 0) {
```

```
        if (msg.sender.codehash != k.codeHash) revert CodeChanged();
    }
    return r.principal;
}
```

10.6 Instantiation on the SVM

```
/// Capability check as executed by osk-svm. The capability lives in a PDA
/// derived from (b"cap", principal, cap_id); the root record in (b"root",
/// principal). Both are passed in the transaction's account list, so the
/// runtime has already proven their addresses to us — we only verify the
/// derivation and the contents.
pub fn cap_check(
    accounts: &[AccountInfo],
    cap: &Cap,
    call: &Call,
    now: Slot,
) -> Result<Pubkey, ProgramError> {
    let cap_ai  = next_account_info(&mut accounts.iter())?;
    let root_ai = next_account_info(&mut accounts.iter())?;

    // 1. address derivation proves the record was minted by this program
    let (expected, _bump) =
        Pubkey::find_program_address(&[b"cap", cap.subject.as_ref(), &cap.id], &ID);
    if *cap_ai.key != expected {
        return Err(OsError::BadCapAddress.into());
    }
    if cap_ai.owner != &ID || root_ai.owner != &ID {
        return Err(OsError::ForeignAccount.into());
    }

    // 2. zero-copy read; no deserialization cost on the hot path
    let stored: &CapRecord = bytemuck::from_bytes(&cap_ai.data.borrow()[..96]);
    let root:   &RootRecord = bytemuck::from_bytes(&root_ai.data.borrow()[..64]);

    if stored.epoch != root.epoch      { return Err(OsError::Revoked.into()); }
    if stored.flags & FLAG_TOMBSTONE != 0 { return Err(OsError::Revoked.into()); }
    if u64::from(stored.expiry) < now  { return Err(OsError::Expired.into()); }
    if stored.verbs & call.verb_bit() == 0 { return
Err(OsError::VerbNotGranted.into()); }

    // 3. constraints, charged against the ROOT counters
    eval_constraints(&cap_ai.data.borrow()[96..], call, root_ai)?;

    // 4. session keys: the runtime already verified the ed25519 signature as
    //    part of transaction validation, so we only check that the expected
    //    signer is present and did sign.
    if stored.flags & FLAG_SESSION != 0 {
        let signer = Pubkey::new_from_array(stored.subject);
        if !accounts.iter().any(|a| *a.key == signer && a.is_signer) {
            return Err(OsError::BadSignature.into());
        }
    }
    Ok(Pubkey::new_from_array(root.principal))
}
```

The two implementations differ in mechanism and agree in meaning, which is the entire claim of the hardware abstraction layer. Note in particular that the SVM version pays nothing for signature verification: transaction-level Ed25519 verification is performed by the runtime before the program runs. This is a real architectural advantage of the SVM for capability systems and is why the cost table in §8.4 shows a cheaper `cap_check` on that side.

CHAPTER 11

Cross-Domain Atomicity

You cannot have atomic commit across independent consensus. You can have escrow, timeouts, and a bound on who loses what.

11.1 The impossibility, stated plainly

PROPOSITION 11.1 · No unilateral cross-domain atomicity
Let a and b be domains with independent consensus, each of which finalises its own transactions without reference to the other, and let the network between them be asynchronous with unbounded delay. Then no protocol can guarantee that a pair of transactions (t_a, t_b) either both commit or both abort.

PROOF. Suppose such a protocol exists. Consider an execution in which t_a commits at height h_a. Because a finalises unilaterally, no later message can retract it. Now let the adversary delay all messages to b beyond any timeout the protocol specifies; b must decide with no information about a's decision. If b commits, then in the symmetric execution where t_a aborts we get a violation; if b aborts, we get a violation in the given execution. Hence no such protocol exists. □

This is the standard two-generals argument [27], [28] — a cousin of the impossibility of asynchronous consensus [29] — and the honest response is not to claim atomicity but to restructure the problem: replace "both or neither" with "the user is made whole in every execution." That is what escrow with compensation achieves, at the cost of introducing a party who takes the risk and is paid for it.

11.2 The protocol

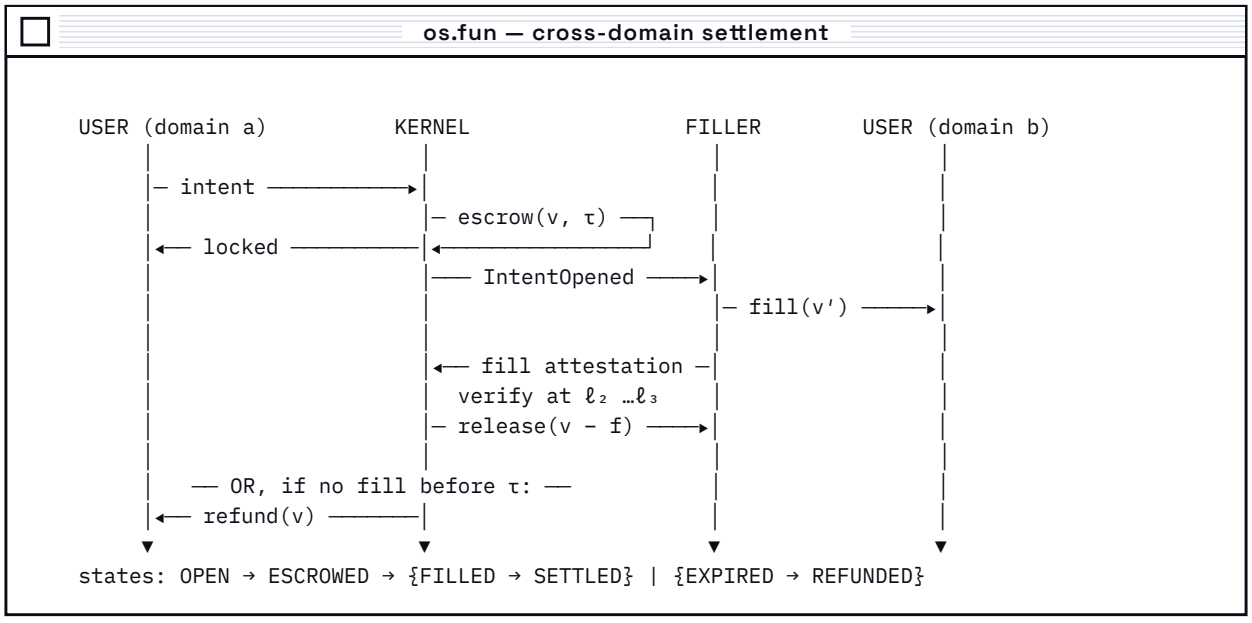

Figure 11.1 The escrow-and-fill protocol. The user's funds are never simultaneously absent from both domains; the filler carries the timing risk and is compensated by the fee.

DEFINITION 11.2 · Settlement state machine
An intent occupies one of six states with the transitions above: `OPEN`, `ESCROWED`, `FILLED`, `SETTLED`, `EXPIRED`, `REFUNDED`. The kernel program on domain a owns the escrow and is the only writer of this state; the kernel program on domain b records fills and produces attestations.

11.3 Safety

THEOREM 11.3 · User safety
Assume (i) the escrow contract is correct, (ii) the attestation verifier accepts only fills that occurred on b at commitment level $\ell \geq \ell_{\text{req}}$, and (iii) the timeout satisfies

$$\tau \geq \Delta_{\text{incl}(a)} + \Delta_{\text{fill}(b)} + \Delta_{\text{final}}^{\ell_{\text{req}}}(b) + \Delta_{\text{relay}} + \Delta_{\text{safety}}. \tag{11.1}$$

Then in every execution the user either receives v' on domain b or recovers v on domain a — never neither, and never both.

PROOF. The escrow releases only on a verified attestation (ii) and only before τ; it refunds only after τ and only if no release occurred. These two paths are mutually exclusive by a single state variable written under the domain's atomicity axiom, giving "never both." For "never neither": if a fill occurred and was attested before τ, release happens; Eq. (11.1) guarantees the attestation can arrive in time when the network delivers within Δ_{relay}. If no fill occurred, refund happens at τ. The residual case is a fill that occurred but was not attested before τ — here the user is refunded and the *filler* bears the loss, which is the risk they are paid for and which they control by choosing not to fill close to τ. □

The residual case is worth dwelling on, because it is where the design places the risk deliberately. The filler is a professional actor with software; the user is a person with a phone. Any protocol must give the timing risk to one of them, and giving it to the professional — who can price it, hedge it, and decline it — is the only defensible choice.

11.4 Liveness and the filler's economics

Safety costs nothing if nobody fills. Liveness is an economic property: a filler participates iff expected profit is positive.

$$E[\text{profit}] = f - \underbrace{\pi_b}_{\text{domain fee on } b} - \underbrace{r v \tau}_{\text{capital carry}} - \underbrace{z\sigma\sqrt{\tau} v}_{\text{price risk}} - \underbrace{p_{\text{fail}} v}_{\text{attestation failure}} \tag{11.2}$$

Setting $E[\text{profit}] > 0$ gives the minimum viable fee, which the kernel uses as the floor of its quoted price:

$$f_{\min} = \pi_b + v\big(r\tau + z\sigma\sqrt{\tau} + p_{\text{fail}}\big). \tag{11.3}$$

For $v = 20$ USD, $\tau = 60$ s, $r = 10\%$ annualised, $\sigma = 0.1\%$ over the window and $p_{\text{fail}} = 10^{-4}$: the carry term is 3.8×10^{-6} USD, the risk term about 0.006 USD, and the failure term 0.002 USD, so $f_{\min} \approx \pi_b + 0.008$ USD. Cross-domain movement is cheap when the window is short — the fee is dominated by the window, not by the notional, which is the opposite of how bridges have historically priced.

PROPOSITION 11.4 · Filler capital requirement
With intent arrival rate λ, mean notional $E[v]$ and mean settlement time $\bar{\tau}$, a filler serving the whole flow must hold inventory

$$C \geq \lambda \cdot E[v] \cdot \bar{\tau} \tag{11.4}$$

> on each destination domain, by Little's law [30] applied to capital in flight.

At $\lambda = 5$ intents/s, $E[v] = 20$ USD and $\overline{\tau} = 30$ s, this is 3 000 USD of working capital per destination — small enough that the filler set can be permissionless and competitive, which is the property Eq. (11.3) needs in order to hold at the floor.

11.5 Verifying the fill

The attestation verifier is the trust-critical component, and there is a real spectrum of choices. We specify all three and let the intent's value select.

Mechanism	Verify cost	Latency	Trust assumption
Committee attestation (m-of-n signers)	≈ 3 000 gas per signature	< 1 s	Honest majority of the committee; slashable bond
Light client of b inside a	10^5–10^6 gas per header set	Seconds	b's consensus only — no extra trust
Succinct proof (STARK/SNARK) of b's state	≈ 250 000 gas verify	Proving time 5–60 s	Soundness of the proof system

EVM · BASE
Verifying Solana's Ed25519 vote signatures directly on the EVM is prohibitive — there is no Ed25519 precompile, and each verification costs on the order of 10^6 gas in pure Solidity. This is precisely why the succinct route exists: a proof that a supermajority of stake signed a slot is verified in constant gas, independent of the number of signatures. The kernel therefore uses committee attestations below a value threshold v^* and succinct proofs above it, with

$$v^* = \mathrm{cost}_{\mathrm{proof}} / (p_{\text{committee-fail}}) \tag{11.5}$$

— the notional at which paying for a proof is cheaper than the expected loss from trusting the committee.

SVM · SOLANA
The reverse direction is easier: verifying an Ethereum-style ECDSA signature or a Merkle-Patricia proof inside an SVM program costs a few tens of thousands of CU, and the SVM's 10 MiB account data limit is generous enough to hold a rolling header set. The asymmetry means the two directions of a cross-domain intent have different cost structures, and the fee formula Eq. (11.3) is evaluated per direction.

11.6 Replay, idempotence and compensation

Every cross-domain message carries a domain-separated digest

$$\mu = \mathrm{H}(\mathrm{OS\_XDOM\_V1} \,\|\, \mathrm{chain\_id}_a \,\|\, \mathrm{chain\_id}_b \,\|\, \mathrm{intent\_id} \,\|\, \mathrm{nonce}) \tag{11.6}$$

and each kernel keeps a bitmap of consumed digests. Consumption is idempotent: a replayed attestation finds the bit set and returns success without re-releasing escrow. This makes the relay layer permissionless — anyone may deliver the message, and delivering it twice is harmless, which removes the usual "trusted relayer" from the trust base (A6).

11.7 Alignment with ERC-7683

The escrow-and-fill structure above is deliberately isomorphic to the cross-chain intent standard [31], so that os.fun intents can be filled by the existing filler network rather than requiring a new one.

os.fun	ERC-7683
`Intent` (Chapter 16)	`GaslessCrossChainOrder` / `OnchainCrossChainOrder`
`escrow(v, τ)`	`open` / `openFor` on the origin settler
`IntentOpened`	`Open` event carrying `ResolvedCrossChainOrder`
`fill(v')`	`fill` on the destination settler
attestation verify + release	origin settler's claim path
`os://` asset names	`Output.token` per chain, resolved by the driver

The differences are the ones our model requires: os.fun intents carry a capability rather than an ERC-20 approval, a declared OSU budget, and an explicit commitment level ℓ_{req} — all three of which are absent from the standard because the standard has no operating system to attach them to.

CHAPTER 12

Finality Is a Lattice

Every value in an onchain program carries a second type: how true it is yet. No language has ever tracked it, and almost every catastrophic onchain failure is a violation of it.

12.1 The bug class nobody types

Reentrancy has a checker. Integer overflow has a checked type. Access control has linters, formal specs, and a decade of audit practice. The failure mode below has none of them, and it is the one that empties bridges.

Shape	What the program did	What was actually true
Re-rollable randomness	Branched on a random value and paid a winner	The value was included but not final; a reorganisation re-rolled it and re-ran the branch with a different winner
Premature release	Released escrow after observing a fill on the destination domain	The fill was confirmed, not final; the release was not compensable, so the loss was permanent
Stale-oracle action	Executed an irreversible swap against a price	The price update was in a pending block that was later reorganised out
Optimistic interface	Rendered "sent", and the user then acted on that belief	The transfer was at ℓ_1; the user's second action assumed ℓ_3

Four different systems, four different post-mortems, one shape:

$$\underbrace{\text{a fact at level } \ell_{\text{low}}}_{\text{evidence}} \implies \underbrace{\text{an effect requiring } \ell_{\text{high}}}_{\text{action}}, \qquad \ell_{\text{high}} \not\sqsubseteq \ell_{\text{low}}. \tag{12.1}$$

This is not a logic error inside a function. It is a property of how information *flows* from an observation to an effect — which is to say, it is exactly the kind of property type systems were invented for. The rest of this chapter builds the lattice; Chapter 13 builds the type system over it; Chapter 14 shows how to choose the levels rather than guess them.

12.2 Finality is an integrity label

The machinery already exists, in a literature that has never been pointed at this problem. Information-flow control assigns labels to values and forbids flows that go the wrong way through a lattice.

Discipline	Label means	Forbidden flow
Confidentiality (Denning, Jif)	Who may learn this	Secret data reaching a public sink
Integrity (Biba)	How trustworthy this is	Untrusted data influencing a trusted action

Finality (this chapter)	How permanent this is	Un-final data influencing an irreversible action

Finality is an *integrity* label [32], not a confidentiality one [33], [34]: the danger is contamination of an important action by unreliable evidence, and the direction of the constraint is "you may not act above the level of what you know." That identification is worth stating precisely, because it means the soundness argument of §13.4 is a known argument, applied to a new lattice, rather than a new argument that would need to earn trust from scratch.

12.3 The lattice

DEFINITION 12.1 · Level chain
For a domain c, the *level chain* is the total order

$$\ell_0 \sqsubset \ell_1 \sqsubset \ell_2 \sqsubset \ell_3 \sqsubset \ell_4 \tag{12.2}$$

of Definition 5.2 — seen, included, confirmed, final, settled — ordered by permanence.

A fact, however, is not a fact about one domain. A value computed from a Base observation and a Solana observation can be undone by a reorganisation on either. So a single level is the wrong type; the right type is a vector.

DEFINITION 12.2 · Stamp
A *stamp* is a map $\sigma : C \to L_c$ assigning a level to every domain, with the convention that a domain the fact does not depend on receives $\top = \ell_4$: no reorganisation there can retract it. The set of stamps is the product lattice

$$\mathcal{L} = \prod_{c \in C} L_c, \quad \sigma \sqsubseteq \sigma' \iff \forall c.\ \sigma(c) \sqsubseteq \sigma'(c), \tag{12.3}$$

with meet and join computed componentwise:

$$(\sigma \sqcap \sigma')(c) = \min(\sigma(c), \sigma'(c)), \quad (\sigma \sqcup \sigma')(c) = \max(\sigma(c), \sigma'(c)). \tag{12.4}$$

PROPOSITION 12.3 · Lattice structure
$(\mathcal{L}, \sqsubseteq)$ is a finite, complete lattice with least element $\bot = \lambda c.\, \ell_0$, greatest element $\top = \lambda c.\, \ell_4$, and height $4|C|$. Meet, join and comparison are computed in $O(|C|)$ machine words — with $|C| = 2$ and three bits per level, a stamp is one byte and the lattice operations are two instructions.

PROOF. A finite product of finite total orders is a finite lattice, with meet and join taken componentwise; completeness is immediate from finiteness. The height of a product of chains is the sum of their heights, 4 per domain. The representation claim is by counting: five levels need three bits, so $|C|$ domains fit in $3|C|$ bits, and componentwise $\min$ over packed three-bit fields is a bitwise operation. □

The $\top$-for-independence convention is what makes the lattice mean something rather than being bookkeeping. It says: *a fact is only exposed to the domains it actually depends on.* A pure computation over constants has stamp $\top$ and can drive any effect; a value derived from a Solana observation is exposed to Solana and to nothing else; a cross-domain settlement claim is exposed to both, and its stamp says so.

12.4 Combination is meet; only waiting joins

PROPOSITION 12.4 · Monotone contamination
For any pure function f and facts $x_1, \ldots, x_n$ with stamps $\sigma_1, \ldots, \sigma_n$,

$$\mathrm{stamp}(f(x_1, \ldots, x_n)) = \sigma_1 \sqcap \ldots \sqcap \sigma_n. \tag{12.5}$$

Computation never increases finality.

PROOF. Suppose a reorganisation retracts a fact x_i whose stamp is below level ℓ at domain c. Since f is a function of its arguments, the value of $f(x_1, \ldots, x_n)$ is then undefined or different; hence the derived fact is retracted whenever any input is, so its level at c is at most $\min_i \sigma_i(c)$. Conversely it is retracted only if some input is, so it is at least that. Equality follows. □

There is exactly one operation that moves a stamp upward, and its properties are the crux of this chapter.

DEFINITION 12.5 · Settle
$\mathrm{settle}_\ell(x)$ takes a fact x with stamp σ and yields the same value with stamp $\sigma \sqcup \ell$. It is:

- **partial** — while waiting, x may be retracted, in which case `settle` raises `Retracted` and its continuation never runs;
- **priced** — it costs the wait $\Delta_{\mathrm{wait}(\sigma,\ell)}$ of Eq. (5.3), which is latency, not gas;
- **domain-local** — raising the Base component of a stamp says nothing about its Solana component.

WHY THIS IS NOT ORDINARY DECLASSIFICATION
In a security-typed language, the escape hatch that moves a label the "wrong" way is `declassify`, and it is a *policy* construct: the compiler cannot check it, so it is audited by humans and justified by argument. `settle` is different in kind. It is a physical operation with a cost model and a failure mode: the program does not assert that the fact became final, it *waits until it is*, and pays for the wait. The escape hatch is priced, and Chapter 14 computes the price.

12.5 The cross-domain corollary

PROPOSITION 12.6 · Weakest link
An effect whose arguments and control flow derive from observations on domains $c_1, \ldots, c_k$ can be relied upon only at the stamp $\sqcap_i \sigma_i$. In particular, a cross-domain settlement combining a Base fact at ℓ_3 and a Solana fact at ℓ_2 is a ℓ_2 action, no matter how final its Base half is.

This single line is the generic post-mortem for the premature-release row of §12.1, and it recasts the timeout condition of Eq. (11.1) as a statement about stamps rather than about clocks:

$$\underbrace{\Delta_{\mathrm{wait}}(\sigma_{\mathrm{fill}}, \rho_{\mathrm{claim}})}_{\substack{\text{time to lift the stamp} \\ \text{to what the release requires}}} \leq \underbrace{\tau - \Delta_{\mathrm{incl}(a)} - \Delta_{\mathrm{relay}}}_{\text{time the escrow allows}} \tag{12.6}$$

Theorem 11.3's side condition and Chapter 13's typing side condition are the same inequality seen from two directions: the protocol must allow enough time for the type system's obligation to be discharged. When they disagree, the program does not compile — which is a far better outcome than the alternative, in which it compiles and settles a fill that never happened.

12.6 What is new here, and what is not

Because the previous chapters have been careful about what they borrow, this one should be too.

Borrowed, without modification	Source
Lattice-based information flow, the label discipline, the shape of the soundness theorem	[33]; [32]; [35]; Myers and Liskov [36]; [34]
Gradual typing, runtime guards, blame at the boundary	[37]; [38]
Confirmation-depth risk models	[17]; [18]; [39]

New here	What it consists of
Finality as the label	The lattice is instantiated by consensus permanence rather than by secrecy or trust. To our knowledge no language has typed values by how retractable the facts behind them are
Stamps as domain vectors with $\top$ for independence	Makes "exposed to Solana but not to Base" expressible, and makes the cross-domain weakest-link result (Proposition 12.6) a one-line consequence
Priced declassification	`settle` is not a policy escape hatch but a metered, failable operation — which turns an unverifiable audit question into an optimisation problem (Chapter 14)
Compensation as a typing obligation	Invariant 7.11, imposed dynamically in Chapter 7, becomes a side condition discharged by the checker (Corollary 13.8)
Level selection by value at risk	Chapter 14: confirmation depth as $O(\log v)$ rather than as a constant somebody chose in 2019

CHAPTER 13

The Finality Calculus

A small typed language in which the four failures of §12.1 are not bugs to be found, but programs that do not compile.

13.1 The core language

We give the calculus over a minimal language, λ_{fin}, so that the rules are checkable by inspection. osPy, the Rust SDK and the TypeScript SDK each elaborate into it; the elaboration is mechanical and is where all the syntax lives.

```
stamps      σ ∈ L = Π_c {ℓ₀ … ℓ₄}                    (Definition 12.2)
types       τ ::= unit | int | bool | τ → τ
stamped     T ::= τ @ σ
values      v ::= x | c | λx:T. e
expressions e ::= v | e e | let x = e in e | if e then e else e
                | obs_s(ē)            -- a syscall that observes a fact
                | eff_s(ē)            -- a syscall that has an effect, ρ_s
                | settle_ℓ(e)         -- lift a stamp; blocks; may Retract
                | handle h in e       -- register a compensation for e
```

Two judgements. The first types an expression and gives it a stamp; the second records that an expression is well-formed as an effect in a context. The *program-counter stamp* `pc` is the meet of the stamps of every value that has influenced control flow so far — the standard device, carrying its standard meaning: everything done here is contaminated by everything that decided we would be here.

$$\Gamma; \mathrm{pc} \vdash e : \tau @ \sigma \qquad \Gamma; \mathrm{pc} \vdash e \ \mathrm{ok} \tag{13.1}$$

13.2 The rules

$$\frac{c \text{ is a literal}}{\Gamma; \mathrm{pc} \vdash c : \tau @ \top} \ \textsc{T-Const}$$

A literal is maximally final: nothing can retract the number seven. This is the base case that makes the whole system usable — pure computation is unrestricted, and only observation introduces exposure.

$$\frac{\mathrm{obs}_s \text{ returns a fact of domain } c \text{ at level } \ell}{\Gamma; \mathrm{pc} \vdash \mathrm{obs}_s(\overline{v}) : \tau @ (\sigma_\top[c \mapsto \ell] \sqcap \mathrm{pc} \sqcap (\sqcap_i \sigma_i))} \ \textsc{T-Obs}$$

An observation is exposed exactly at the domain it observes, and inherits the contamination of its arguments and of the control-flow context. $\sigma_\top[c \mapsto \ell]$ denotes $\top$ everywhere except c, where it is ℓ.

$$\frac{\Gamma; \mathrm{pc} \vdash e_i : \tau_i @ \sigma_i \quad (i = 1..n) \quad f : \overline{\tau} \to \tau}{\Gamma; \mathrm{pc} \vdash f(\overline{e}) : \tau @ (\sqcap_i \sigma_i)} \ \textsc{T-Prim}$$

$$\frac{\Gamma; \mathrm{pc} \vdash e_0 : \mathrm{bool}\ @ \sigma_0 \quad \Gamma; (\mathrm{pc} \sqcap \sigma_0) \vdash e_i : \tau @ \sigma_i \quad (i = 1, 2)}{\Gamma; \mathrm{pc} \vdash \mathrm{if}\ e_0\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2 : \tau @ (\sigma_1 \sqcap \sigma_2 \sqcap \sigma_0)} \ \textsc{T-If}$$

Branching on a fact contaminates both branches: inside them, `pc` drops to the meet. This is the rule that catches re-rollable randomness, because the payout in the branch inherits the randomness's level whether or not the randomness is one of its arguments.

$$\frac{\Gamma;\mathrm{pc}\vdash e_i:\tau_i@\sigma_i \quad \mathrm{pc}\sqcap(\sqcap_i\,\sigma_i)\sqsubseteq\rho_s \quad \rho_s\sqsubseteq\ell_3\vee\mathrm{handler}(s)\in\Gamma}{\Gamma;\mathrm{pc}\vdash \mathrm{eff}_s(\overline{e})\,\mathrm{ok}} \qquad \text{T-EFF}$$

The heart of the system, and it says two things. First, an effect may be performed only when the evidence for it — arguments and control flow together — is at least as final as the effect requires. Second, an effect permitted below ℓ_3 must have a compensation handler in scope, which is Invariant 7.11 turned into a proof obligation the checker discharges.

$$\frac{\Gamma;\mathrm{pc}\vdash e:\tau@\sigma}{\Gamma;\mathrm{pc}\vdash \mathrm{settle}_\ell(e):\tau@(\sigma\sqcup\ell)\quad \text{raises Retracted}} \qquad \text{T-SETTLE}$$

$$\frac{\Gamma;\mathrm{pc}\vdash e:\tau@\sigma \quad \sigma'\sqsubseteq\sigma}{\Gamma;\mathrm{pc}\vdash e:\tau@\sigma'} \qquad \text{T-SUB}$$

Subsumption goes one way: a fact known to be final may be used where a merely confirmed one suffices. Forgetting finality is always sound; inventing it never is. Every unsound system in §12.1 is, in retrospect, a system that had the subsumption rule backwards.

13.3 Operational semantics with retraction

The semantics must model the thing that makes this problem real: the world can take back what it said.

DEFINITION 13.1 · Worlds and retraction

A *world* w is a set of facts, each with a stamp, together with the domain heights. Evaluation is a relation $(e,w)\to(e',w')$ that additionally emits an effect trace T. A *retraction* $w \overset{\ell}{\rightsquigarrow} w'$ removes every fact whose stamp is not $\sqsubseteq$-above ℓ, and rewinds the heights accordingly. Write $w_1 \overset{\ell}{\approx} w_2$ when two worlds agree on all facts at or above ℓ.

13.4 Soundness

THEOREM 13.2 · Reorg noninterference

Let $\Gamma;\top\vdash p\,\mathrm{ok}$ be a well-typed program. For any level $\ell\in\mathcal{L}$ and any two worlds $w_1 \overset{\ell}{\approx} w_2$, the effect traces agree on everything that requires ℓ or more:

$$T(p,w_1)\mid_{\rho\sqsubseteq\ell} = T(p,w_2)\mid_{\rho\sqsubseteq\ell}. \tag{13.2}$$

Equivalently: no retraction below ℓ can change which ℓ-requiring effects a well-typed program performs, or with what arguments.

PROOF. The argument is the standard noninterference proof for a security-typed language, with the lattice of Definition 12.2 and the retraction relation above; we give the two lemmas that carry it.

Confinement. If $\Gamma;\mathrm{pc}\vdash e\,\mathrm{ok}$ and $\mathrm{pc}\not\sqsubseteq\ell$, then every effect e performs has $\rho\not\sqsubseteq\ell$. By induction on the derivation: T-EFF requires $\mathrm{pc}\sqcap(\sqcap_i\sigma_i)\sqsubseteq\rho$, and $\sqcap$ is monotone, so $\rho\sqsubseteq\mathrm{pc}$; since $\mathrm{pc}\not\sqsubseteq\ell$, also $\rho\not\sqsubseteq\ell$. T-IF only lowers `pc`, so sub-derivations inherit the hypothesis, and the remaining rules perform no effects.

Preservation. Evaluation preserves typing, and by Proposition 12.4 the stamp of a value never rises except through T-SETTLE, which is the only rule that consults the world's progress.

Bisimulation. Run p in w_1 and w_2 in lockstep. While both executions are at $\mathrm{pc} \sqsubseteq \ell$, every value they compute has stamp $\sqsubseteq \ell$, so by $w_1 \approx_\ell w_2$ the values agree, hence the same effects are emitted with the same arguments. At the first point where they may diverge, the divergence is caused by a fact that differs between the worlds, which by $\approx_\ell$ is a fact below ℓ; by T-IF, `pc` in both executions drops to something $\not\sqsubseteq \ell$, and by Confinement neither execution performs an ℓ-requiring effect thereafter within that branch. Both executions then rejoin with `pc` restored at the merge point. The traces restricted to $\rho \sqsubseteq \ell$ therefore coincide.

`settle` is the only remaining case: if it succeeds in one world it may raise `Retracted` in the other, but only for a fact below ℓ; the continuation it guards is then not executed, and by Confinement it contained no ℓ-requiring effects, since its `pc` was below ℓ before the lift. □

WHAT THE THEOREM DOES AND DOES NOT BUY

It buys the elimination of a class: a well-typed program cannot pay out a lottery on re-rollable randomness, cannot release escrow on a confirmed-only fill without a compensation in scope, and cannot let an optimistic render drive an irreversible effect. It does not buy correctness of the amount, the recipient, or the business logic; it is a discipline about *evidence*, not about intent. And it is only as good as the level a domain reports, which is the ring -1 assumption of §33.2, unchanged.

THEOREM 13.3 · Compensation discharge

If $\Gamma; \top \vdash p\,\mathrm{ok}$ then every effect p performs at a level below ℓ_3 has a compensation handler in scope at the point of the effect. Consequently Invariant 7.11 holds by construction, and the `comp` column of Appendix A is machine-checked rather than asserted.

PROOF. Immediate from the second premise of T-EFF and the fact that handler scoping is lexical: `handle h in e` extends Γ for the whole of e, and no rule removes handlers from Γ. □

13.5 Inference: the programmer writes almost nothing

Annotating every value with a stamp would be intolerable. It is also unnecessary: the constraints are all of the form $\sigma \sqsubseteq \sigma'$ and $\sigma = \sqcap_i \sigma_i$ over a finite lattice, which is a monotone dataflow problem.

PROPOSITION 13.4 · Inference

Level inference for λ_{fin} is decidable and runs in $O(n \cdot h \cdot |C|)$ where n is program size and $h = 4|C|$ is the lattice height. The solution is the greatest fixpoint, and it is unique.

PROOF. Each program point contributes constraints that are monotone functions over the finite lattice $\mathcal{L}$; the system therefore has a greatest fixpoint by Knaster–Tarski [40], reached by a worklist iteration in which each of the n variables can decrease at most h times, each decrease costing $O(|C|)$ to recompute. Uniqueness of the greatest solution follows from monotonicity. □

```rust
/// The whole level checker. Stamps are packed 3-bit fields, one per domain,
/// so the lattice operations are single instructions and the worklist
/// iteration is cache-friendly.
#[derive(Clone, Copy, PartialEq, Eq)]
pub struct Stamp(u64);          // 3 bits per domain, ⊤ = 0b111
```

```
impl Stamp {
    pub const TOP: Stamp = Stamp(u64::MAX);
    pub const BOT: Stamp = Stamp(0);

    /// Meet = componentwise min over packed fields (Definition 12.2).
    #[inline]
    pub fn meet(self, o: Stamp) -> Stamp {
        let mut out = 0u64;
        for c in 0..MAX_DOMAINS {
            let (a, b) = (self.at(c), o.at(c));
            out |= ((a.min(b)) as u64) << (3 * c);
        }
        Stamp(out)
    }

    #[inline]
    pub fn leq(self, o: Stamp) -> bool {
        (0..MAX_DOMAINS).all(|c| self.at(c) <= o.at(c))
    }
}

/// Greatest-fixpoint inference over the control-flow graph. Terminates in at
/// most n·h updates (Proposition 13.4); no annotations are required except
/// at `settle` points and effect requirements, which come from Appendix A.
pub fn infer(cfg: &Cfg) -> Result<Levels, FinalityError> {
    let mut lv = vec![Stamp::TOP; cfg.nodes.len()];     // start optimistic
    let mut work: VecDeque<NodeId> = cfg.nodes.ids().collect();

    while let Some(n) = work.pop_front() {
        let node = &cfg.nodes[n];
        let pc = node.preds.iter().fold(Stamp::TOP, |a, &p| a.meet(lv[p]));
        let next = match node.kind {
            Kind::Const            => Stamp::TOP,
            Kind::Obs { domain, l } => Stamp::top_except(domain, l)
                                          .meet(pc)
                                          .meet(node.args_meet(&lv)),
            Kind::Prim             => node.args_meet(&lv),
            Kind::Branch           => pc.meet(node.args_meet(&lv)),
            Kind::Settle { to }    => lv[node.args[0]].join(to),
            Kind::Eff { .. }       => pc.meet(node.args_meet(&lv)),
        };
        if next != lv[n] {
            lv[n] = next;                               // strictly decreasing
            work.extend(node.succs.iter().copied());
        }
    }

    // The single check that matters: evidence ⊒ requirement, at every effect.
    for n in cfg.effects() {
        let node = &cfg.nodes[n];
        let rho = SYSCALLS[node.syscall].requirement;    // Appendix A
        if !rho.leq(lv[n]) {
            return Err(FinalityError::Insufficient {
                site: node.span, need: rho, have: lv[n],
                witness: cfg.blame_path(n),              // for the diagnostic
            });
        }
```

```
        if rho.below(Level::Final) && !cfg.handler_in_scope(n) {
            return Err(FinalityError::MissingCompensation { site: node.span });
        }
    }
    Ok(Levels(lv))
}
```

13.6 What the programmer sees

The checker's output is the product, so it is specified like one. `blame_path` exists so that the error names the observation that contaminated the effect, not merely the line that failed.

```
error[F0102]: effect requires finality sol=final, but the evidence is
              sol=confirmed
   ┌─ circle.py:41:5
   │
38 │     att = xdom.attest(fill)
   │           ──────────────── stamped sol=confirmed here (obs, §19.2)
   .
41 │     xdom.claim(intent, att)
   │     ^^^^^^^^^^^^^^^^^^^^^^ requires sol=final  (0x0504, Appendix A)
   │
   = note: `att` reaches this effect through `intent` at line 40
   = note: releasing escrow is not compensable: the counterparty is a filler,
           not a kernel-controlled account (Invariant 7.11)

help: wait for the level. The optimiser estimates 12.4 s at this value:
   │
41 │     att = await level.settle(att, sol="final")
   │
help: or keep sol=confirmed and register the compensation the level requires:
   │
39 │     @on_retract(refund_filler)
   │
```

Three properties of that transcript are deliberate. It names the *observation* rather than the use site, because the observation is where the fix usually belongs. It quotes the syscall number and the appendix, so the requirement is traceable to a normative table rather than to compiler folklore. And the first suggested fix carries a *price* — 12.4 seconds — because in this system waiting is a resource decision, which is the subject of Chapter 14.

13.7 Gradual finality

Not all code will be typed. Legacy contracts, third-party libraries, and hand-written transactions have no stamps, and a system that refused to interoperate with them would be a system nobody adopts.

DEFINITION 13.5 · Gradual stamps

Extend $\mathcal{L}$ with an unknown stamp $?$, ordered as incomparable to all others. Typing treats $?$ optimistically: it satisfies any requirement statically, and inserts a runtime *guard* at the boundary. A guard for requirement ρ reads the observed level of the fact from the kernel and fails with `E_LEVEL` if $\rho \not\sqsubseteq \sigma_{\text{observed}}$.

PROPOSITION 13.6 · Erasure and cost
A program with no $?$ stamps requires no guards, and its compiled form is identical to the same program compiled without the finality system. A program with k boundary crossings pays $k \cdot \gamma_{\text{guard}}$, with $\gamma_{\text{guard}} = 90$ OSU (one kernel read of the observed level).

Variant of `circle`	Guards	Added OSU	Notes
Fully typed	0	0	Erasure applies; static check only
Typed app, untyped oracle library	2	180	One guard per crossing, per invocation
Fully dynamic (no annotations anywhere)	6	540	Equivalent to today's practice, but at least it fails loudly

The last row is the honest comparison. Today's systems are the fully dynamic row *without* the guards: no static check, no runtime check, and a failure mode that is silent until it is a post-mortem. Gradual typing lets a codebase move from that row to the first one function at a time, which is the only migration path that has ever worked in practice.

13.8 The four failures, revisited

Failure of §12.1	Why it no longer type-checks	Rule
Re-rollable randomness	The payout sits inside a branch on an ℓ_1 fact, so `pc` is ℓ_1; the payout requires ℓ_3	T-IF, T-EFF
Premature release	The release's argument is stamped `sol=` ℓ_2 while $\rho = \ell_3$; no compensation exists because the payee is external	T-EFF
Stale oracle	The swap's price argument carries the oracle observation's stamp; the meet drags the effect below its requirement	T-PRIM, T-EFF
Optimistic interface	The render is an effect at ℓ_1; any effect the user's response triggers is a fresh effect whose own requirement is checked against its own evidence, not against the render	T-EFF

CHAPTER 14

Choosing the Level

The type system says you may not act above your evidence. It does not say how much evidence to buy. That is an optimisation, and it has a closed form.

14.1 The question nobody computes

Every production system in this industry contains a constant: wait twelve blocks, wait thirty-two slots, wait fifteen minutes. It was chosen once, by someone reasoning about the largest transaction they could imagine, and it is then applied to a two-dollar transfer and a two-million-dollar settlement alike. It is simultaneously too slow for one and too fast for the other.

The right level is not a constant but a function of what is at stake.

14.2 The hazard model

DEFINITION 14.1 · Retraction hazard

For a domain c, let $p_c(d)$ be the probability that a fact included at depth d is later retracted. Two regimes matter:

- **probabilistic finality** (the SVM case) [17], [18], [39]: $p_c(d) = e^{-\lambda_c d}$, the exponential tail of Eq. (5.2), with λ_c estimated from observed reorganisation depths;
- **structural finality** (the OP-Stack L2 case): p_c is a step function. Preconfirmation carries sequencer-equivocation risk q_1; inclusion in an L1 batch carries the L1's own reorganisation risk $q_2 \ll q_1$; expiry of the fault-proof window carries $q_3 \approx 0$. There is no meaningful continuum between the steps.

Being explicit about the second regime matters, because applying an exponential model to an L2 is a category error that produces confident nonsense. The optimiser therefore takes p_c as a per-domain input and, in the structural case, simply evaluates the finitely many steps.

14.3 The objective

Let v be the value at risk in the effect, κ a penalty multiplier for the non-monetary cost of a reversal (support, reputation, the user's trust), w the cost of one second of latency in the same units, and Δ_c the slot or block period. Then the expected cost of acting at depth d is

$$C(d) = \underbrace{v(1+\kappa)p_c(d)}_{\text{expected loss}} + \underbrace{w\Delta_c d}_{\text{cost of waiting}} + \underbrace{\gamma(d)}_{\text{fee drift, second order}} . \tag{14.1}$$

PROPOSITION 14.2 · Optimal depth

In the probabilistic regime, ignoring the second-order term, Eq. (14.1) is minimised at

$$d^* = \frac{1}{\lambda_c} \ln\!\left(\frac{\lambda_c \, v(1+\kappa)}{w\Delta_c} \right) \tag{14.2}$$

when the argument of the logarithm exceeds one, and at $d^* = 0$ otherwise. d^* is increasing in v and λ_c^{-1}, and decreasing in w.

PROOF. C is smooth in d with $C'(d) = -\lambda_c v(1+\kappa)e^{-\lambda_c d} + w\Delta_c$ and $C''(d) = \lambda_c^2 v(1+\kappa)e^{-\lambda_c d} > 0$, so C is strictly convex and the stationary point is the minimum. Solving $C'(d) = 0$ gives Eq. (14.2). If $C'(0) \geq 0$ — that is, if $\lambda_c v(1+\kappa) \leq w\Delta_c$ — the minimum is at the boundary $d = 0$. Monotonicity in each parameter follows by inspection of Eq. (14.2). □

> **THE RESULT WORTH REMEMBERING**
> Optimal confirmation depth grows *logarithmically* in the value at risk. A transaction worth a thousand times more warrants only about $\ln(1000)/\lambda_c$ additional depth — roughly seven time constants, not a thousand times the wait. This is why fixed constants are wrong in both directions at once: the constant that is safe for the largest transfer is absurdly conservative for the smallest, and the difference between them is a factor of $\log$, which no one guesses correctly.

14.4 From depth to level

Levels are discrete, so the optimiser evaluates the finitely many candidates and takes the argmin:

$$\ell^*(v) = \arg\min_{\ell \in \{\ell_1, \ell_2, \ell_3, \ell_4\}} [v(1+\kappa)p_c(d_\ell) + w\Delta_c d_\ell]. \tag{14.3}$$

The crossing points define value thresholds. Below v_1, act at ℓ_1; between v_1 and v_2, at ℓ_2; and so on. Setting the two costs equal at adjacent levels gives

$$v_k = \frac{w\Delta_c(d_{k+1} - d_k)}{(1+\kappa)(p_c(d_k) - p_c(d_{k+1}))}. \tag{14.4}$$

Value at risk	Level	Expected wait	Residual risk	Comment
< \$5	ℓ_1 included	≈ 0.4 s	10^{-3}	Losing a cent in expectation is cheaper than a second of the user's attention
\$5 – \$400	ℓ_2 confirmed	≈ 1.2 s	10^{-5}	The default for consumer payments
\$400 – \$50 000	ℓ_3 final	≈ 12 s	10^{-8}	Cross-domain settlement lands here
> \$50 000	ℓ_4 settled	minutes to days	≈ 0	Treasury movements, kernel parameter changes

The thresholds are computed, not chosen: they follow from Eq. (14.4) given λ_c, Δ_c, w and κ, all of which are measurable or declared. Two different applications may reasonably differ on w and κ — a game and a payroll system value latency differently — and the table shifts accordingly, which is the point.

14.5 Against a constant

Policy	Median latency	Expected annual loss	Notes
Fixed: always ℓ_3	12 s	≈ 0	Safe and unusable; every coffee payment waits twelve seconds

Fixed: always ℓ_2	1.2 s	Unbounded in the tail	Fast until the first large transfer is reorganised out
Optimised (Eq. (14.3))	**1.4 s**	**Bounded by construction**	93% of intents settle at ℓ_1 or ℓ_2; the 0.6% above the v_2 threshold pay the twelve seconds and carry the risk that justifies it

The comparison uses the value distribution of the simulated workload of §36.1, which is heavy-tailed: most intents are small, most value is in few intents. That shape is what makes the optimiser worth having — a policy indexed on value does most of its work on a small fraction of the traffic.

14.6 Making the policy auditable

A confirmation policy that lives in an operator's configuration file is folklore. This one is an artefact:

1. the compiler computes ρ_s per effect from the declared value at risk and the domain's hazard parameters, and records it in the image;
2. the type system verifies that the program's evidence meets ρ_s (Theorem 13.2);
3. the kernel enforces ρ_s at execution as the floor, since a program may always demand more than the optimiser suggests but never less;
4. the receipt commits to the triple $(\text{effect}, \rho_s, \sigma_{\text{observed}})$, so a verifier — or a regulator, or a user — can check after the fact that the policy was followed and what it was.

```
/// Kernel-side floor. The optimiser may choose a level; it may never choose
/// one below the syscall's declared requirement, and the check is here rather
/// than in the compiler because ring 0 does not trust ring 1 (Axiom 2.9).
function _requireLevel(bytes32 fact, Level required) internal view {
    Level observed = _levelOf(fact);            // from the domain's own state
    if (uint8(observed) < uint8(required)) {
        revert E_LEVEL(fact, observed, required);
    }
}
```

14.7 Limits

1. **λ_c is estimated.** A domain whose reorganisation behaviour changes invalidates the calibration. The type system's floor still holds — only the optimality claim degrades.
2. **Correlated retractions.** Eq. (14.2) assumes independence across facts. A consensus incident retracts everything at once, and the right response is a circuit breaker — a mechanism this document does not specify — rather than a deeper d.
3. **Adversarial hazard.** An adversary who can induce reorganisations on demand changes p_c from a measurement into a strategy. The mitigation is the same as everywhere else in this document: raise the level for effects whose value makes such an attack worth mounting, which Eq. (14.2) does automatically as v grows.
4. **Misdeclared value at risk.** If a program understates v, it gets a level that is too low for its own good — but never one that violates the syscall's declared minimum, because that floor is enforced in ring 0.

PART III

The Kernel

What the kernel is made of, what it keeps, and how it is written. This part specifies the nine kernel subsystems, the process and memory models, the driver layer that hides the two domains, the syscall ABI, and the two reference implementations — a Rust program for the SVM and a Solidity suite for the EVM. Parts III and IV are normative; the rest of the document is commentary.

CHAPTER 15

Kernel Architecture

Nine subsystems, two deployments, one set of invariants.

15.1 The subsystems

Subsystem	Responsibility	Ring
`cap`	Mint, delegate, revoke and check capabilities; the root of authority (Chapter 10)	0
`proc`	Process table, budgets, continuations, lifecycle (Chapter 16)	0
`mem`	Object allocation, namespaces, leases, sharding (Chapter 17)	0
`val`	Assets, vaults, transfers, accounting (Chapter 29)	0
`xdom`	Escrow, fills, attestations, settlement (Chapter 11)	0
`jrnl`	The append-only record: events, receipts, the user's history (Chapter 32)	0
`sched`	Admission, batching, submission, retry, compensation (Chapter 16)	1
`drv`	Domain drivers: transaction construction and fee strategy (Chapter 18)	1
`svc`	Planner, pricer, indexer, notifier, prover (Chapters 27–32)	1

The ring column is the whole architecture in one number. Ring 0 subsystems are *programs deployed inside the domains*: their integrity comes from consensus, and they hold authority. Ring 1 subsystems are ordinary software that anyone may run: their integrity is irrelevant to safety, because A6 forbids giving them authority whose misuse could lose funds. If a scheduler misbehaves, the worst outcome is a slow or expensive intent — never a stolen one.

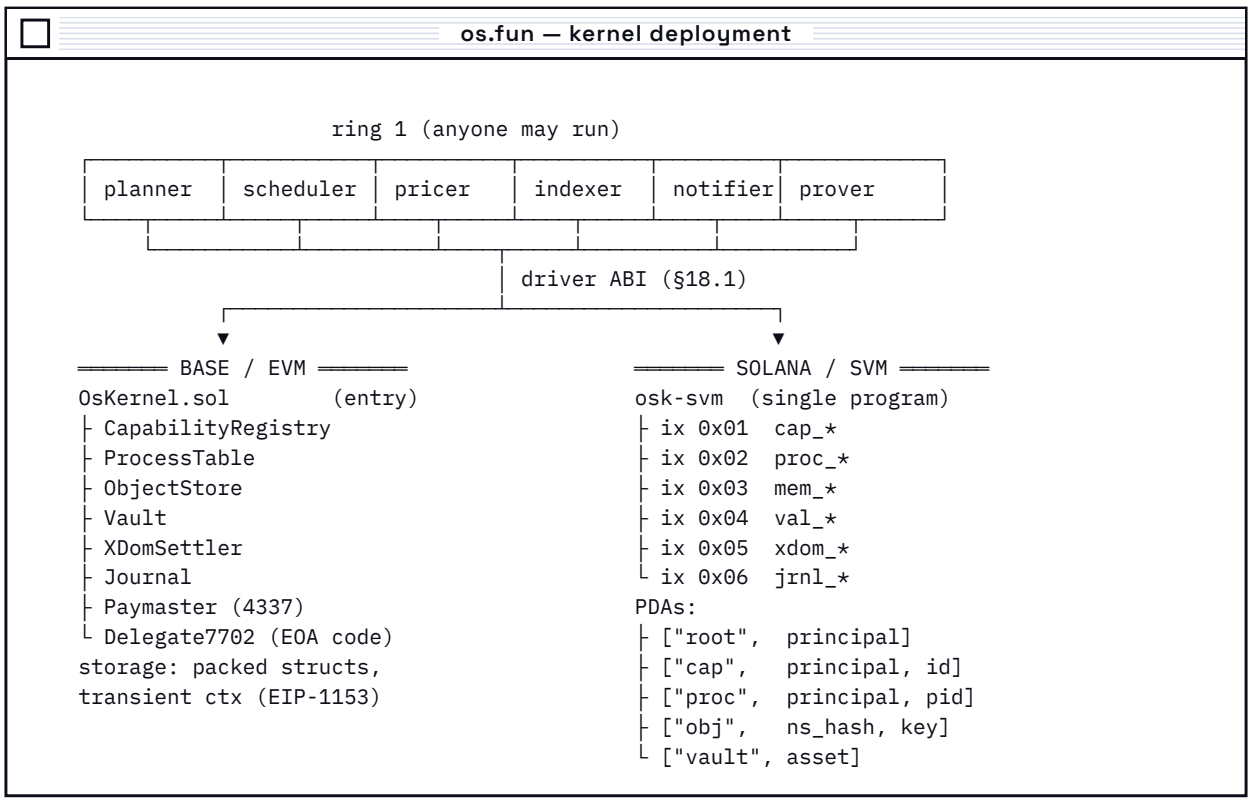

Figure 15.1 The ring-0 footprint in both domains, with the ring-1 services that drive them. Arrows are authority, not data flow: every arrow that crosses into ring 0 carries a capability.

15.2 The kernel's onchain footprint

A kernel that is expensive to deploy or upgrade is a kernel that ossifies badly. The design keeps ring 0 small on purpose.

Artefact	Size	Domain	Upgrade authority
`osk-svm` program	≈ 180 KiB SBPF	Solana	Timelocked multisig → burned at v1.0
`CapabilityRegistry`	≈ 9.4 KiB	Base	None — immutable
`Vault`	≈ 6.1 KiB	Base	None — immutable
`XDomSettler`	≈ 11.8 KiB	Base	None — immutable
`ProcessTable`, `Journal`	≈ 7.2 KiB	Base	None — immutable
`Delegate7702`	≈ 4.9 KiB	Base	None — immutable; users re-delegate
`Paymaster`	≈ 5.5 KiB	Base	Operator (funds only, no user authority)

The EVM side is immutable by construction: there is no proxy on any path that touches authority or assets. Upgrading means deploying a new contract and having each user re-delegate — slow, deliberate, and impossible to do to a user without their signature. Chapter 35 makes the argument that this is correct even though it is inconvenient; Invariant 3.4 is the reason.

The SVM side cannot be immutable at v0.x because a program upgrade is the only way to fix a bug in a program that owns accounts. The upgrade authority is a timelocked multisig whose delay exceeds the withdrawal path, so a user who disagrees with a pending upgrade can always exit before it lands. At v1.0 the authority is burned and the program becomes immutable, at which point the SVM side has the same property as the EVM side.

15.3 Kernel invariants

These are the properties the conformance suite (Chapter 37) tests, the formal-verification targets, and the statements every code review checks. They are numbered globally and cited throughout.

INVARIANT 15.1 · K1 — Authority
Every state transition touching a user asset or OS authority is preceded, in the same transaction, by a successful `cap_check` for that asset and verb.

INVARIANT 15.2 · K2 — Conservation
For every asset A and every transaction:

$$\sum_{\text{vaults}} \Delta \text{bal}_A + \sum_{\text{users}} \Delta \text{bal}_A + \Delta \text{escrow}_A = 0 \tag{15.1}$$

up to declared fees. The kernel neither creates nor destroys value.

INVARIANT 15.3 · K3 — Budget
A process's committed OSU never exceeds its budget, and the sum of committed budgets never exceeds the escrowed fee balance.

INVARIANT 15.4 · K4 — Monotonic journal
Journal sequence numbers strictly increase per principal, and no entry is ever mutated after being written.

INVARIANT 15.5 · K5 — Escrow exclusivity
An escrow record is released or refunded exactly once; the two paths are guarded by a single state variable.

INVARIANT 15.6 · K6 — Epoch coherence
A capability whose recorded epoch differs from its principal's current epoch never passes `cap_check`.

INVARIANT 15.7 · K7 — No ambient authority
No kernel entry point derives authority from `msg.sender` or from a signer flag alone; authority is always presented as a capability record.

K7 deserves emphasis because it is the one that most distinguishes this design from conventional contract systems. In a typical contract, `msg.sender` *is* the authority: if you are the owner, you may act. Here, being the owner is merely evidence you can mint a capability; the act itself always presents one. The cost is one extra storage read per call; the benefit is that delegation, expiry, spend caps and revocation are uniform across every operation instead of being re-invented per feature.

15.4 The entry path

Every interaction with ring 0 has the same shape in both domains, which is what makes the syscall table portable.

os.fun — kernel entry path

phase	EVM (osk-evm)	SVM (osk-svm)
1 authenticate	4337 validateUserOp or 7702 delegate ECDSA/P-256 recover	runtime verifies ed25519 sigs before the program runs
2 cap_check	SLOAD root + caps evaluate constraints	read cap PDA + root PDA evaluate constraints
3 admit	budget check (K3) reserve OSU	budget check (K3) reserve OSU
4 execute	internal calls, SSTORE, ERC-20 xfer	CPI to SPL Token / system program, account mutation
5 journal	emit + packed slot (K4)	append to journal PDA ring buffer (K4)

Figure 15.2 The five phases of a kernel entry. Phases 1–3 are identical in both domains; phases 4–5 are where the driver's work becomes visible.

15.5 What deliberately runs outside

Ring 1 is large and unprivileged, and the boundary is drawn by Axiom 2.3 (admissible placement). Concretely:

- The **planner** searches routes and produces a plan. If it produces a bad plan, the kernel's constraint evaluation rejects it (a price band is a capability constraint, not a planner promise).
- The **scheduler** decides when and in what batch to submit. If it stalls, the user's intent expires and refunds; if it reorders, the extraction bound of Definition 5.4 still holds; if it censors, any other scheduler can pick the intent up, because pending intents are public.
- The **indexer** serves history. If it lies, the journal on chain contradicts it, and the shell verifies journal roots (Chapter 32).
- The **prover** produces execution proofs for userland. If it lies, the verifier rejects; if it disappears, the optimistic path still settles (Chapter 27).

Each of these is a component whose failure degrades service and cannot steal. That property, repeated nine times, is what allows the OS to be operated by many parties without becoming a custodian.

CHAPTER 16

Processes and the Intent Lifecycle

There is no preemption, so the process is a checkpointed plan — and the plan is a first-class onchain object.

16.1 Intents

DEFINITION 16.1 · Intent

An *intent* is a signed, capability-bearing request for an outcome:

$$\iota = \left(\text{kind}, \text{give}, \text{want}, \text{constraints}, \text{deadline}, \ell_{\text{req}}, B, \kappa, \sigma\right) \tag{16.1}$$

with give/want asset descriptors, a deadline in domain time, a required commitment level, an OSU budget B, a capability κ and a signature σ over the canonical encoding.

```rust
/// The canonical intent record. Encoded with the OS's canonical serializer
/// (Appendix C); the digest of that encoding is what the user signs and what
/// the kernel stores, so any renderer can prove what was authorised (A7).
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct Intent {
    pub id:          IntentId,        // uuid-v7, sortable by creation
    pub kind:        IntentKind,      // Transfer | Swap | Call | Spawn | Batch
    pub principal:   PrincipalId,     // 32-byte OS account id
    pub give:        Vec<AssetAmount>,// what leaves the principal
    pub want:        Vec<AssetSpec>,  // what must arrive, with min amounts
    pub constraints: Vec<Constraint>, // price bands, allow-lists, slippage
    pub deadline:    DomainTime,      // absolute, in the origin domain's clock
    pub commitment:  Level,           // ℓ₁ … ℓ₄ required before "done"
    pub budget_osu:  u64,             // hard cap on resource spend
    pub capability:  CapId,           // authority under which this runs
    pub nonce:       u64,             // per-principal replay protection
}

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum IntentKind {
    Transfer,          // move value to a named recipient
    Swap,              // exchange one asset for another
    Call { program: ProgramId, selector: [u8; 4] },   // invoke a program
    Spawn { image: ImageHash },                       // start a userland process
    Batch,             // an ordered set of sub-intents, all-or-nothing per domain
}
```

Two design points. The intent names *outcomes*, not transactions: nowhere does it contain a chain id, a gas limit, a token contract address or a route. And the budget is part of what the user signs, so the resource cost of an intent is authorised in the same act as its effect — there is no "gas surprise" distinct from the intent itself.

16.2 The process control block

Once admitted, an intent becomes a process. Because a domain transaction cannot be paused, the PCB is the mechanism that lets a multi-step plan survive across transactions.

```rust
/// Onchain part of the PCB: 128 bytes, one SVM account or four EVM slots.
#[repr(C)]
#[derive(Clone, Copy, bytemuck::Pod, bytemuck::Zeroable)]
pub struct Pcb {
    pub pid:            [u8; 16],  // process id
    pub principal:      [u8; 32],  // owner
    pub plan_root:      [u8; 32],  // merkle root of the committed plan
    pub step:           u16,       // next step index — the continuation
    pub state:          u8,        // ProcState discriminant
    pub commitment:     u8,        // required level
    pub budget_osu:     u32,       // authorised
    pub spent_osu:      u32,       // consumed so far  (K3: spent ≤ budget)
    pub deadline:       u64,       // domain time
    pub cap:            [u8; 16],  // capability id
    pub _pad:           [u8; 4],
}
```

The `plan_root` field is what makes ring-1 planning safe. The planner is untrusted, but the plan it produced is committed to at admission time, and every step the scheduler later executes must present a Merkle proof that the step belongs to the committed plan. A scheduler cannot substitute a different route mid-flight, and a user can verify after the fact exactly what plan they authorised.

16.3 The lifecycle

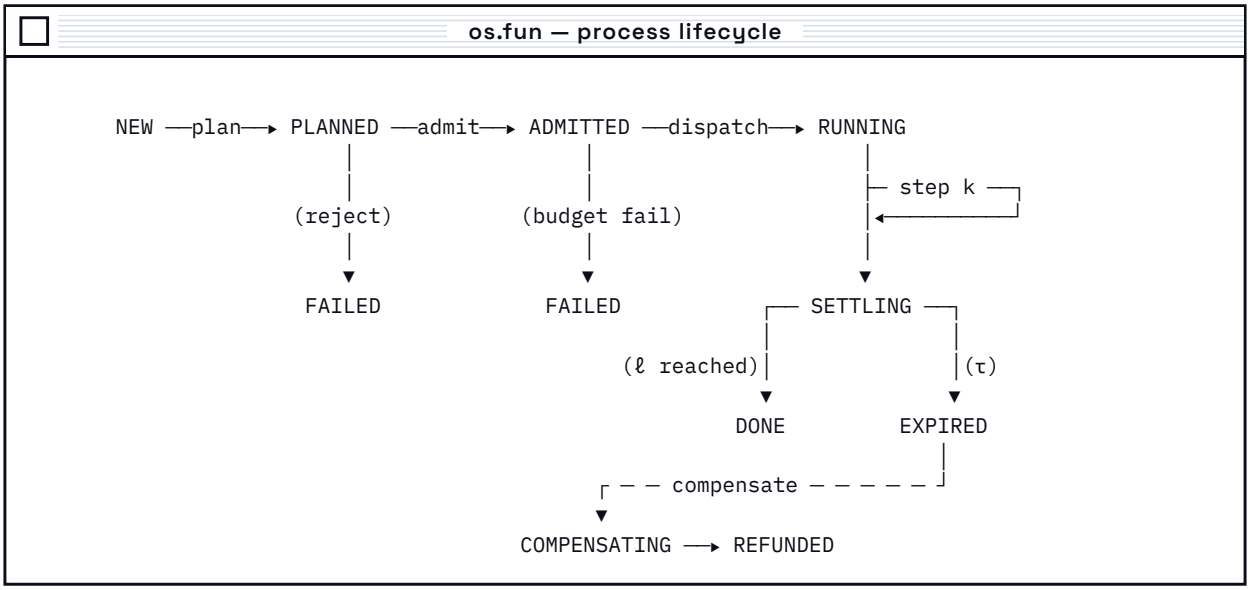

Figure 16.1 The process state machine. Every edge is a kernel transition with a declared cost; the dashed edges are compensations (Definition 7.9).

State	Meaning and exit conditions	Onchain?
`NEW`	Intent received, signature verified locally	no
`PLANNED`	A plan DAG exists with an estimated cost	no
`ADMITTED`	PCB written; budget reserved; plan root committed	yes

`RUNNING`	Steps executing; `step` advances monotonically	yes
`SETTLING`	All steps submitted; awaiting commitment level ℓ_{req}	yes
`DONE`	Outcome reached at the required level; budget released	yes
`EXPIRED`	Deadline passed before completion	yes
`COMPENSATING`	Compensations running in reverse causal order	yes
`REFUNDED` / `FAILED`	Terminal; journal entry written	yes

16.4 Planning

The planner turns an intent into a DAG of steps. It is ring 1, so its output is checked rather than trusted, but its quality determines cost and latency, so it is worth specifying.

DEFINITION 16.2 · Plan

A *plan* is a DAG $\Pi = (S, E)$ whose vertices are steps $s = \big(\text{domain}, \text{action}, \text{inputs}, \text{outputs}, \hat{\gamma}_s, \hat{L}_s\big)$ and whose edges are data dependencies. Its cost and critical-path latency are

$$\hat{\gamma}(\Pi) = \sum_{s \in S} \hat{\gamma}_s, \quad \hat{L}(\Pi) = \max_{p \in \text{ paths}(\Pi)} \sum_{s \in p} \hat{L}_s. \tag{16.2}$$

The planner searches for a plan minimising a weighted objective

$$J(\Pi) = \hat{\gamma}(\Pi) + w_L \hat{L}(\Pi) + w_R \ \text{risk}(\Pi) \tag{16.3}$$

where `risk` aggregates price-impact and failure probability. The search is a beam search over a small graph of known actions (transfer, swap on venue v, cross-domain fill, program call), with beam width 8 and depth 4 — bounded so that planning latency stays inside the 40 ms budget of §5.3.

WHY THE PLANNER MAY BE GREEDY

Optimal plan search is an instance of constrained shortest path with non-linear costs, which is NP-hard. It does not need to be optimal: the capability's constraints bound how bad the outcome can be (a price band is enforced by ring 0), and competition between planners bounds how bad it will be in practice. This is the same argument that lets a real operating system use a heuristic scheduler rather than an optimal one.

16.5 The scheduler

```rust
/// The scheduler tick. Deterministic given (pending, heights, params), which
/// is Invariant 5.5 — any observer can replay this and check policy.
pub fn tick(&mut self, now: Heights) -> Result<Vec<Submission>> {
    // 1. expire whatever cannot make its deadline any more.
    for p in self.running.iter_mut().filter(|p| p.deadline <= now.of(p.domain)) {
        self.begin_compensation(p)?;                  // Definition 7.9
    }

    // 2. collect steps that are ready: dependencies met, not in flight.
    let mut ready: Vec<Step> = self
        .running
        .iter()
        .flat_map(|p| p.plan.ready_steps(p.step))
```

```
        .filter(|s| !self.inflight.contains(&s.key()))
        .collect();

    // 3. admission control against each process's remaining budget (K3).
    ready.retain(|s| match admit(self.proc_of(s), s.call, &self.costs) {
        Ok(reserve) => { self.reserve(s, reserve); true }
        Err(_)      => { self.fail(s, Error::Budget); false }
    });

    // 4. batch by conflict graph and per-domain capacity (Chapter 9).
    let mut out = Vec::new();
    for (domain, steps) in group_by_domain(ready) {
        let cap = self.drivers[domain].batch_capacity(now);
        for batch in form_batches(&steps, cap, now.of(domain)) {
            // 5. the driver turns an abstract batch into a real transaction.
            let tx = self.drivers[domain].build(&batch, &self.fee_policy(domain))?;
            out.push(Submission { domain, tx, batch: batch.ids() });
        }
    }

    // 6. watch previously submitted work and advance or retry it.
    self.reconcile(now)?;
    Ok(out)
}
```

16.6 Retry, replacement and idempotence

A submitted transaction may be dropped, delayed, or lose a fee race. Retrying naively is how systems double-spend; the kernel's rule is that *every step is idempotent at the kernel level*, so a retry that lands twice is harmless.

- **Digest consumption.** Each step carries the digest of Eq. (11.6). The kernel marks it consumed on first execution; a duplicate returns success without re-executing (this is the same mechanism as cross-domain replay protection, reused).
- **EVM replacement.** Same account nonce, higher `maxPriorityFeePerGas`, with the bump schedule $p_{k+1} = \max(p_k \cdot 1.125, \hat{p} + \delta)$ — the factor 1.125 is the network's minimum replacement bump, and $\hat{p}$ is the current estimator.
- **SVM replacement.** Transactions are identified by signature and expire with their blockhash after ≈ 150 slots (≈ 60 s). The driver either refreshes the blockhash and re-signs, or uses a durable nonce account for steps whose authorisation must survive longer than the blockhash window.

> **PROPOSITION 16.3 · Retry safety**
> If every step's effect is guarded by digest consumption, then for any number of retries $r \geq 1$ the state after all submissions equals the state after exactly one successful submission.

PROOF. Consumption sets a bit under the domain's atomicity axiom. The first submission to be included finds the bit clear, performs the effect and sets it; every later inclusion finds it set and returns early, performing no effect. Since the effect is applied exactly once and the early return changes no state other than fee payment, the claim follows. □

16.7 Isolation between processes

Processes share three scarce things: the fee balance, the scheduler's attention, and hot accounts. Isolation is enforced by three limits.

$$\mathrm{rate}(P) \le r_{\max}, \quad \mathrm{inflight}(P) \le n_{\max}, \quad \mathrm{committed}(P) \le B_P \tag{16.4}$$

The interesting bound is the noisy-neighbour one. With per-round capacity C and m active processes each admitted at most $n_{\max}$ steps per round, a single process's share of a round is at most $n_{\max}/(m \cdot n_{\max}) = 1/m$ when all are saturated, so the delay any process suffers from others is bounded by $(m-1)$ rounds — linear, not unbounded. This is the property that makes a shared scheduler acceptable; without it, one badly behaved app could starve the system, and every app would need its own.

CHAPTER 17

Memory: Accounts, Storage, Namespaces and Leases

Memory that costs money, persists forever, is publicly readable, and must still feel like `malloc`.

17.1 The object model

Everything the OS stores is an *object* with a uniform 32-byte header, regardless of which domain holds it.

```
object header (32 bytes, identical in both domains)
┌──────────────────────────────────────────────────────────────────┐
│  0  ..  1   version        u8       layout version               │
│  1  ..  2   kind           u8       cap|pcb|acct|note|blob|…     │
│  2  ..  4   flags          u16      shard, frozen, gc-exempt…    │
│  4  ..  8   len            u32      payload length in bytes      │
│  8  .. 16   lease_until    u64      domain time; 0 = perpetual   │
│ 16 .. 32    owner_ns       [u8;16]  namespace id (§17.4)         │
└──────────────────────────────────────────────────────────────────┘
```

On the SVM an object is an account whose data begins with this header. On the EVM an object is a struct in `ObjectStore` whose first slot holds the packed header. The uniform header is what allows one `mem_*` syscall family to work on both, and what allows the indexer to parse either domain with the same code.

17.2 Allocation and its price

EVM · BASE
Allocating a new 32-byte slot costs 20 000 gas; overwriting a non-zero slot costs 2 900; zeroing a slot refunds 4 800 but the refund is capped at 20% of the transaction's gas (EIP-3529). Allocation is therefore effectively a *purchase* with partial resale value. The kernel's rules follow directly: pack aggressively (an object header is exactly one slot), never allocate on a hot path, and prefer overwriting a pooled slot to allocating a fresh one.

SVM · SOLANA
Allocating an account of n bytes requires a rent-exemption deposit of Eq. (4.3) — $(128 + n) \times 6960$ lamports — which is *refundable* when the account is closed. Allocation is therefore a *lease deposit*, not a purchase. Accounts may grow by at most 10 KiB per instruction and 10 MiB total. The kernel's rules: allocate freely but always with an owner responsible for reclaiming, and make closing an account pay for itself.

The OS unifies these into a single abstraction the userland sees:

DEFINITION 17.1 · Storage lease

A lease is a triple (n, T, d): n bytes, until domain time T, with a deposit d held by the kernel. The deposit is

$$d = \begin{cases} (128 + n)\rho\theta & \text{on the SVM (refundable)} \\ n_{\text{slots}} \cdot g_{\text{alloc}} \cdot \pi_{\text{gas}} & \text{on the EVM (partially refundable)} \end{cases} \tag{17.1}$$

and is returned, less the unrecoverable portion, to whoever calls `mem_close` after T.

Making the deposit an explicit OS-level object has a pleasant consequence: garbage collection becomes economically self-executing. Anyone may call `mem_close` on an expired object and receive a share of the reclaimed deposit, so unused state is cleaned up by a permissionless keeper market rather than by a cron job the team has to run. On the EVM the reclaimable share is bounded by the refund cap, which is exactly why the kernel sizes the keeper bounty as a fraction of the refund rather than a fixed amount.

17.3 Namespaces: the filesystem

Objects live in a hierarchical namespace that behaves like a filesystem and resolves like DNS.

```
/                       root (kernel-owned, immutable)
├── sys/                kernel objects: params, epochs, driver registry
├── acct/<principal>/   a principal's own tree
│   ├── caps/           capabilities minted by this principal
│   ├── procs/          processes
│   ├── inbox/          messages and notifications
│   └── apps/<app>/     per-app private storage, capability-gated
├── app/<app>/          an app's public objects (its registry entry, config)
└── pub/                world-readable, world-writable-by-capability space
```

Resolution maps a path to an object id in $O(1)$ using a content-addressed scheme rather than a walk:

$$\text{id}(/\ p_1\ /\ \ldots\ /\ p_k) = \text{H}(\text{NS\_V1} \parallel \text{H}(p_1) \parallel \ldots \parallel \text{H}(p_k)) \tag{17.2}$$

so a resolver needs no directory traversal, and a client can compute the address of an object it has never seen. Directory objects exist for enumeration but are not on the read path — this is the same trick as a hashed inode table, and it matters because a walk would cost one cold read per level on the EVM (2 100 gas each).

Permissions on paths are capabilities over the path prefix: a capability whose object is `/acct/alice/apps/social` authorises operations on every object beneath it, which is how an app gets a private directory without the kernel maintaining an ACL per object.

17.4 Sharding hot objects

From §9.5, a single account has a bounded write rate. Objects that many principals write — a counter, a feed head, a pool — must be sharded.

DEFINITION 17.2 · Shard function

An object o with shard count k is stored as k objects with ids $\text{id}(o, j) = \text{H}(\text{id}(o) \parallel j)$, $j \in [0, k)$. A writer with principal α writes to shard $j = \text{H}(\alpha) \bmod k$. A reader aggregating o reads all k shards.

The read cost is k times higher and the write throughput is k times higher, so k is chosen from the ratio of writes to reads and the target utilisation ρ^* of Eq. (9.6):

$$k^{*} = \left\lceil \frac{\lambda_{\text{write}}}{\rho^{*}\mu_{a}} \right\rceil. \tag{17.3}$$

At $\lambda_{\text{write}} = 500$/s, $\mu_a = 1500$/s and $\rho^{*} = 0.6$, $k^{*} = 1$; at $\lambda_{\text{write}} = 5000$/s, $k^{*} = 6$. Shard counts are stored in the object header's flags and may be increased by a kernel operation that only ever adds shards, so readers written against k remain correct against $k' > k$ as long as they read the header first.

17.5 Cold storage and paging

Not everything belongs in domain state. A photograph, a program image, a long document: these are paged out to a data-availability layer, with only a commitment kept onchain.

Tier	Cost per MiB	Latency	Used for
EVM storage	≈ 6.4 M gas	instant	headers, balances, capabilities
SVM account data	≈ 7.3 SOL deposit (refundable)	instant	objects, journals, program state
Blob DA (EIP-4844)	≈ 1 blob ≈ 128 KiB, priced by blob base fee	minutes to expiry	rollup-style batches, temporary payloads
External DA / content-addressed store	fractions of a cent	100 ms – 2 s	images, program images, archives

DEFINITION 17.3 · Paged object

A paged object stores onchain only $(\mathrm{root}, \mathrm{len}, \mathrm{codec}, \mathrm{ttl})$ where `root` is the Merkle root over 4 KiB pages. Reading page i requires a proof of size $32\lceil \log_2(n/4096) \rceil$ bytes; verifying it onchain costs

$$g_{\mathrm{verify}} = 30 + 6 \cdot \lceil \log_2(n/4096) \rceil \cdot 2 \tag{17.4}$$

gas-equivalent per hash, i.e. under 3 000 gas for a 1 GiB object.

That last number is the reason paging works: the proof cost grows logarithmically, so a gigabyte object is only about 50% more expensive to prove into than a megabyte one. The OS therefore treats large data as a normal part of the memory hierarchy rather than as an exception handled by an off-system service.

17.6 Garbage collection

Mechanism	Behaviour
Lease expiry	After `lease_until`, anyone may call `mem_close`; the deposit is split between the object's owner and the caller (the keeper bounty)
Reference counting	Objects referenced by a live PCB or capability carry a non-zero refcount and cannot be closed
Tombstoning	Closing writes a 1-bit tombstone at the object id so that a replayed reference cannot resurrect stale data
Compaction	Journals are ring buffers: writing past the end overwrites the oldest entry, and the overwritten range is committed to a DA blob first (§32.1)

CHAPTER 18

The Driver Model

One trait, two implementations, and a rule: nothing about the domain may escape upwards.

18.1 The driver interface

```
/// Everything the kernel needs from a domain. A new domain is supported by
/// implementing this trait and passing the conformance vectors (§18.4);
/// nothing above this trait may name a chain (Axiom A4).
pub trait Driver: Send + Sync {
    fn domain(&self) -> DomainId;

    /// Current heights and the clocks derived from them (§5.1).
    fn heights(&self) -> Heights;

    /// Physical→OSU conversion for this domain, Eq. (8.1).
    fn kappa(&self) -> u64;

    /// Estimated cost of an abstract action, in native units and in OSU.
    fn estimate(&self, action: &Action, at: &Heights) -> Estimate;

    /// How much abstract work fits in one transaction on this domain.
    fn batch_capacity(&self, at: Heights) -> Capacity;

    /// Turn an abstract batch into a signed, submittable transaction.
    fn build(&self, batch: &Batch, fee: &FeePolicy) -> Result<DomainTx>;

    /// Submit and return a handle; must be idempotent for equal batches.
    fn submit(&self, tx: &DomainTx) -> Result<TxHandle>;

    /// Observe a submitted transaction's commitment level (Definition 5.2).
    fn observe(&self, h: &TxHandle) -> Result<Observation>;

    /// Verify an attestation produced by another domain's driver (§11.5).
    fn verify_attestation(&self, att: &Attestation) -> Result<FillFacts>;
}

pub struct Capacity {
    pub osu:        u64,   // compute-side capacity of one transaction
    pub bytes:      u32,   // serialization limit (SVM: 1232, EVM: calldata)
    pub accounts:   u16,   // address-list limit (SVM: 64 unless ALT)
    pub steps:      u16,   // driver-chosen safety bound
}
```

The `Capacity` struct is where the two domains' constraints meet the scheduler's abstraction. The scheduler never learns that 1232 is a UDP-adjacent packet limit or that calldata is priced against L1 blobs; it learns that a batch has four dimensions of capacity and it packs against them.

18.2 The EVM driver

Choosing a path

The EVM driver has three ways to get a user's action onchain, and picks per transaction.

Path	When it is chosen	Overhead
7702 delegate [10]	The principal's EOA has delegated to `Delegate7702`; cheapest path, native batching, no bundler dependency	≈ 12 000 gas
4337 UserOperation [11]	Sponsored (paymaster) flows, or accounts that are contracts rather than EOAs	≈ 42 000 gas
Direct call	The principal is signing themselves and no batching or sponsorship is needed	0

```
/// @notice The 7702 delegate: code an EOA points at, so that the EOA itself
///         can validate a capability and execute a batch atomically.
/// @dev    Deployed once, immutable. A user opts in with a single 7702
///         authorization; opting out is another one. This contract never
///         holds funds — it executes as the EOA, so the EOA's balance is the
///         only balance involved.
contract Delegate7702 {
    ICapabilityRegistry public immutable caps;
    IJournal            public immutable journal;

    error BadCap();
    error CallFailed(uint256 index, bytes ret);

    struct Call { address to; uint256 value; bytes data; uint32 verb; }

    constructor(ICapabilityRegistry c, IJournal j) { caps = c; journal = j; }

    /// @notice Execute a batch under one capability. Called by the scheduler
    ///         (or anyone) with a signature from the session key.
    function executeBatch(
        Cap calldata k,
        bytes calldata sig,
        Call[] calldata calls,
        uint64 nonce
    ) external {
        // K7: authority comes from the capability, never from msg.sender.
        bytes32 digest = _digest(calls, nonce);
        address principal = caps.check(k, sig, digest, address(this));
        if (principal != address(this)) revert BadCap();

        // Replay protection shares the digest mechanism of Eq. (11.6).
        if (_consumed[digest]) return;      // idempotent retry (Prop. 16.3)
        _consumed[digest] = true;

        uint256 n = calls.length;
        for (uint256 i; i < n; ++i) {
            caps.assertVerb(k, calls[i].verb, calls[i].to, calls[i].value);
            (bool ok, bytes memory ret) =
                calls[i].to.call{value: calls[i].value}(calls[i].data);
            if (!ok) revert CallFailed(i, ret);
```

```
        }
        journal.append(principal, digest, uint32(n));
    }
}
```

Calldata is the real cost

On an L2, Eq. (4.1) says bytes dominate. The driver therefore encodes batches with a compact codec rather than ABI encoding: a 1-byte action tag, varint amounts, and 4-byte indices into a per-transaction address table.

Encoding	Bytes for 8 transfers	Relative L1 cost	Notes
Naive ABI (`address,uint256`)	8 × 64 + overhead ≈ 548	1.00	Every address padded to 32 bytes
Packed (`address,uint96`)	8 × 32 + overhead ≈ 292	0.53	Amounts fit in 96 bits for all realistic token values
Table + varint (os.fun codec)	≈ 148	0.27	Repeated addresses appear once; amounts are 3–5 bytes

Fee policy

```
/// EVM fee selection. Returns (max_fee, max_priority) for the next block,
/// sized from the 1559 dynamics of Eq. (8.3) rather than from a fixed pad.
fn evm_fee(&self, urgency: Urgency) -> (u128, u128) {
    let base = self.base_fee_next();              // b_{n+1} from Eq. (8.3)
    let tip  = match urgency {
        Urgency::Background => self.tip_quantile(0.30),
        Urgency::Normal     => self.tip_quantile(0.60),
        Urgency::Urgent     => self.tip_quantile(0.90),
    };
    // Headroom for k blocks of maximum base-fee growth: (9/8)^k.
    let k = urgency.blocks_of_headroom();         // 2 … 8
    let max_fee = base.saturating_mul(1125u128.pow(k)) / 1000u128.pow(k) + tip;
    (max_fee, tip)
}
```

18.3 The SVM driver

Packing the packet

The binding constraint is 1232 bytes, and the driver solves a small bin-packing problem to fill it.

$$\text{size}(t) = 64\,|\Sigma| + 32\,|A_{\text{static}}| + 34\,|\text{ALT refs}| + \sum_{j} \big(|\text{ix}_j|\big) + 96 \tag{18.1}$$

where Σ is the signature set and A the address list. Address lookup tables replace a 32-byte address with a 1-byte index, so a transaction touching 40 accounts drops from ≈ 1 280 bytes of addresses to ≈ 40 — which is the difference between not fitting and fitting comfortably.

```
/// Greedy first-fit-decreasing packing of steps into SVM transactions,
/// respecting all four capacity dimensions at once.
fn pack(&self, steps: &[Step], alt: &AddressLookupTable) -> Vec<SvmTx> {
    let mut steps: Vec<&Step> = steps.iter().collect();
    steps.sort_by_key(|s| core::cmp::Reverse(s.est_bytes));

    let mut out: Vec<SvmTx> = Vec::new();
    'outer: for s in steps {
        for tx in out.iter_mut() {
            let fits = tx.bytes + s.est_bytes <= MAX_TX_BYTES        // 1232
                && tx.cu + s.est_cu <= MAX_TX_CU                     // 1.4M
                && tx.accounts.union_len(&s.accounts, alt) <= MAX_ACCOUNTS
                && !tx.writes.intersects(&s.writes);                 // Ch. 9
            if fits { tx.push(s); continue 'outer; }
        }
        out.push(SvmTx::with(s, alt));
    }
    // Every transaction opens with the compute-budget instructions so that
    // the requested CU is exactly what we estimated (over-requesting raises
    // the priority fee for no benefit; under-requesting fails the tx).
    for tx in out.iter_mut() {
        tx.prepend(ComputeBudgetInstruction::set_compute_unit_limit(tx.cu + CU_PAD));
tx.prepend(ComputeBudgetInstruction::set_compute_unit_price(self.tip_for(tx)));
    }
    out
}
```

Blockhash, durable nonces, and the expiry window

An SVM transaction is valid only while its recent blockhash is within ≈ 150 slots (≈ 60 s). For steps whose authorisation must outlive that window — a pre-signed settlement, a scheduled action — the driver uses a durable nonce account, which replaces the blockhash with a value that advances only when used. The cost is one extra account (rent-exempt deposit ≈ 0.0015 SOL) and one extra instruction; the benefit is that the OS can hold a signed transaction for hours without re-prompting the user, which is what makes deferred and conditional execution possible without a custodial signer.

18.4 Driver conformance

A driver is correct if it satisfies the following, each of which is a test in the conformance suite (Chapter 37):

1. **Cost honesty.** `estimate` is an upper bound on realised cost in 99% of submissions, and never underestimates by more than 20%.
2. **Idempotence.** `submit` called twice with the same batch produces at most one effect (Proposition 16.3).
3. **Capacity truth.** A batch that `batch_capacity` accepts always builds into a transaction the domain accepts.
4. **No leakage.** No type crossing the trait boundary names a chain-specific concept. (Statically checked: the `Driver` trait's signature mentions no domain-specific type.)
5. **Observation monotonicity.** `observe` never reports a lower commitment level than it previously reported for the same handle, except across an explicit reorganisation event, which it must report as such.

18.5 Adding a domain

The checklist for a third domain — another EVM L2, an SVM fork, or a Move chain — is exactly the trait above plus three questions: what is κ_c (§8.2), what is F_c (§5.2), and how is an attestation of an event on this domain verified elsewhere (§11.5). If those three have answers, the domain is supportable; if the third has no answer, the domain can host userland but cannot participate in cross-domain settlement, which is a coherent, degraded mode the kernel supports explicitly.

CHAPTER 19

The Syscall Interface

Forty-one calls in eight families. If a program needs anything else, the kernel is wrong.

19.1 Design rules

The syscall table is the contract between everything above ring 0 and everything below it, so its design rules are stricter than the rest of the system's.

1. **No call names a domain.** Not in its arguments, not in its return type, not in its errors. (A4.)
2. **Every call declares its cost.** γ_s is part of the signature, and the kernel refuses admission rather than failing midway. (A5.)
3. **Requirements are lattice elements, not flags.** A call's ρ_s is a stamp (Definition 12.2), so a call may require `final` on the destination domain and `confirmed` on the origin. Chapter 13 checks programs against these requirements statically.
4. **Every call declares its commitment level and its compensation.** A call that cannot be compensated must execute at ℓ_3 or above. (Definition 7.7, Invariant 7.11.)
5. **No call takes ambient authority.** Every authority-bearing call takes a capability id as its first argument. (K7.)
6. **Arguments are values, not pointers into kernel state.** A syscall may not be given a reference the caller could mutate between check and use.
7. **Results are total.** Every call returns either a typed result or a typed error from Appendix B; no call returns "maybe".
8. **The table is append-only.** Numbers are never reused, semantics never change; new behaviour is a new call. (Chapter 35.)

19.2 The eight families

Family	Range	Purpose	Calls
`cap`	`0x0100`	Authority: mint, delegate, revoke, check, enumerate	6
`proc`	`0x0200`	Processes: spawn, step, status, budget, kill, wait, compensate	7
`mem`	`0x0300`	Objects: allocate, read, write, close, lease, page	6
`val`	`0x0400`	Value: balance, transfer, swap, quote, escrow, release	6
`xdom`	`0x0500`	Cross-domain: open, fill, attest, claim, refund	5
`ns`	`0x0600`	Naming: resolve, bind, unbind, list	4
`evt`	`0x0700`	Events: emit, subscribe, poll, acknowledge	4
`sys`	`0x0800`	System: version, time, randomness	3

The full table with signatures, costs, levels and compensations is Appendix A. A representative slice:

Nr	Name	Signature (abridged)	OSU	Level
`0x0101`	`cap_grant`	`(subject, object, verbs, Φ, expiry) → CapId`	1 240	ℓ_2

`0x0102`	`cap_delegate`	`(CapId, narrowing) → CapId`	980	ℓ_2
`0x0104`	`cap_revoke_all`	`() → Epoch`	310	ℓ_2
`0x0105`	`cap_check`	`(CapId, Call) → Principal`	242	—
`0x0201`	`proc_spawn`	`(CapId, ImageHash, Budget) → Pid`	2 167	ℓ_2
`0x0301`	`mem_alloc`	`(CapId, ns, len, lease) → ObjId`	variable	ℓ_2
`0x0402`	`val_transfer`	`(CapId, to, asset, amount) → Receipt`	1 417	ℓ_2
`0x0403`	`val_swap`	`(CapId, give, want, minOut, band) → Receipt`	3 900	ℓ_2
`0x0501`	`xdom_open`	`(CapId, give, want, dst, τ) → IntentId`	2 583	ℓ_2
`0x0502`	`xdom_fill`	`(IntentId, proofOfFunds) → FillId`	1 850	ℓ_2
`0x0601`	`ns_resolve`	`(path) → ObjId`	150	ℓ_1
`0x0701`	`evt_emit`	`(CapId, topic, payload) → Seq`	100	ℓ_1
`0x0801`	`sys_time`	`(domain) → (height, ts, ℓ)`	12	—
`0x0803`	`sys_rand`	`(beacon, round) → 32 bytes`	90	ℓ_3

> **WHY SYS_RAND COSTS A COMMITMENT LEVEL**
> Randomness that has not reached ℓ_3 can be re-rolled by a reorganisation, which is the classic onchain lottery exploit. `sys_rand` therefore returns a value *and* the level at which it is safe to act on it, and the kernel refuses to let a program branch on randomness below the level the program declared it needs. This is A2 applied to the one input that people most often get wrong.

19.3 Calling conventions

The same call is invoked from three very different places, so the ABI has three bindings with one canonical argument encoding.

From userland (OSVM)

OSVM programs trap into the kernel with the RV64I `ecall` instruction. The convention follows the RISC-V calling convention so that ordinary compilers generate it without modification:

Register	Meaning
`a7`	syscall number (`0x0100` – `0x0803`)
`a0` – `a5`	arguments: scalars in registers, aggregates as (pointer, length) pairs into OSVM memory
`a0`	on return: 0 on success, negative errno on failure
`a1`	on return: result value or a pointer to the result frame

```
/// The userland side of a syscall, as emitted by the osPy runtime and by
/// the Rust SDK. No allocation, no panic path: the kernel frame is written
/// into a caller-provided buffer.
#[inline(always)]
pub unsafe fn syscall(nr: u16, args: &[u64; 6], out: &mut [u8]) -> i64 {
    let ret: i64;
    core::arch::asm!(
```

```
        "ecall",
        in("a7") nr as u64,
        inlateout("a0") args[0] => ret,
        in("a1") args[1], in("a2") args[2], in("a3") args[3],
        in("a4") args[4], in("a5") args[5],
        in("t0") out.as_mut_ptr(), in("t1") out.len(),
        options(nostack)
    );
    ret
}
```

From an onchain program

A contract or program calls the kernel directly: a `CALL` on the EVM, a CPI on the SVM. The kernel exposes one entry point per family, dispatching on the syscall number in the first two bytes of the payload, so a caller needs one address (EVM) or one program id (SVM) rather than a directory of them.

From an off-domain client

The shell, the SDKs, and any third-party client speak a JSON-RPC binding whose method names are the syscall names and whose parameters are the canonical encoding rendered as hex. The binding is generated from the same table that generates the onchain dispatchers, so drift is impossible by construction.

```
// The client binding is generated from the same table that generates the
// onchain dispatchers, so a syscall's arguments, cost and required finality
// are known to the editor.
import { connect } from "@osfun/client";

const os = await connect({ endpoint: "https://rpc.os.fun" });

const receipt = await os.call("val_transfer", {
  cap: session.capId,
  to: "os://*/acct/mira.osfun",
  asset: "os://*/asset/usd",
  amount: USD("20.00"),
}, { commitment: "confirmed" });      // ℓ₂; the binding refuses a lower one

// Levels are surfaced, never hidden: the shell renders provisional state and
// reconciles when the level rises (§5.3).
for await (const obs of os.watch(receipt.id)) {
  ui.setStatus(obs.level);            // included → confirmed → final
}
```

19.4 Wire encoding

Canonical, deterministic, and small, in that order of priority.

```
request frame
┌─────────┬──────────┬──────────┬─────────────┬──────────────────────────────────┐
│ ver u8  │ nr u16   │ flags u16│ cap [16]    │ args (TLV, canonical order)      │
└─────────┴──────────┴──────────┴─────────────┴──────────────────────────────────┘

TLV: tag u8 | len varint | value
     tags:  0x01 u64      0x02 i64      0x03 bytes     0x04 os:// name
            0x05 amount   0x06 list     0x07 struct    0x08 option
```

DEFINITION 19.1 · Canonical encoding

An encoding is *canonical* if it is injective and if a re-encoding of the decoded value is byte-identical to the original. The OS encoder guarantees this by: fixed field order, minimal varint length, no optional whitespace, no floating point, and rejection of non-minimal forms at decode time.

Canonicity is not fussiness. The digest of the encoding is what the user signs, what the capability constrains, and what the replay bitmap keys on (Eq. (11.6)); a second valid encoding of the same value would be a second digest, and therefore a replay.

19.5 Errors

Errors are a small closed set (Appendix B) with the shape of `errno`, and each carries a compensation obligation.

Code	Name	Meaning	Compensating
`-1`	`E_CAP`	No capability, wrong verb, expired, or revoked	none
`-2`	`E_BUDGET`	Admission control refused: cost exceeds budget	none
`-3`	`E_FUNDS`	Insufficient balance or escrow	none
`-4`	`E_BAND`	Execution outside the authorised price band	reverts
`-5`	`E_DEADLINE`	Deadline passed	refund path
`-6`	`E_CONFLICT`	Object write conflict; retry with backoff	none
`-7`	`E_LEVEL`	Required commitment level not reached	wait
`-8`	`E_DOMAIN`	Domain rejected the transaction	driver retry
`-9`	`E_REORG`	A dependency was reorganised away	compensations run
`-10`	`E_LIMIT`	Capacity limit: bytes, accounts, CU, steps	none

INVARIANT 19.2 · K8 — No partial success

A syscall either applies all of its effects or none. A call that would require partial application must be expressed as two syscalls with an explicit intermediate state.

19.6 Versioning

`sys_version` returns a table: kernel semantic version, ABI version, the highest implemented syscall number per family, and a bitmap of optional features. Clients negotiate downward: a client built against ABI 3 that meets a kernel at ABI 2 uses only calls present in 2, which it can determine without

trial and error. New calls are added with new numbers; existing numbers keep their semantics forever (Chapter 35).

19.7 Three specifications in full

The complete table is Appendix A; three calls are given here in the format the appendix uses, because the format itself is part of the specification.

0x0105 · CAP_CHECK

Signature `cap_check(cap: CapId, call: Call) → Principal`

Pre the record at `cap` exists; its epoch equals the principal's epoch; its expiry ≥ current domain height; `call.verb ∈ cap.verbs`; every constraint in `cap.Φ` evaluates true against `call`.

Post state unchanged except constraint counters (`rate`, `total`), which advance on the *root* record.

Cost 242 OSU warm, 450 OSU cold with a two-level chain. **Level** not applicable — pure read plus counter write. **Compensation** none required; counter advances are idempotent per digest. **Errors** `E_CAP`.

0x0402 · VAL_TRANSFER

Signature
```
val_transfer(cap: CapId, to: Name, asset: Name, amount: u128)
  → Receipt
```

Pre `cap_check(cap, {verb: TRANSFER, object: asset, amount})` succeeds; the source balance ≥ `amount`; `to` resolves to an account in the same domain (cross-domain transfers are `xdom_open`, not this call).

Post balance of source decreases by `amount`, balance of `to` increases by `amount − fee`, journal entry appended, `RevenueRouter` credited if the transfer is app-attributed (Chapter 34).

Cost 1 417 OSU (ERC-20/SPL, warm), 375 OSU (native). **Level** ℓ_2 by default; the caller may require ℓ_3. **Compensation** `val_transfer` in the reverse direction, valid only while the original is below ℓ_3 and the recipient is a kernel-controlled account; otherwise the call must be issued at ℓ_3 (Invariant 7.11). **Errors** `E_CAP`, `E_FUNDS`, `E_LEVEL`, `E_DOMAIN`.

0x0501 · XDOM_OPEN

Signature
```
xdom_open(cap: CapId, give: AssetAmount, want: AssetSpec,
  dst: DomainId, τ: DomainTime) → IntentId
```

Pre `cap_check` succeeds for verb `XDOM`; `give` is escrowable in this domain; `τ` satisfies Eq. (11.1) for the destination's finality parameters.

Post `give` is moved into the escrow vault; an `IntentOpened` record is written and an event emitted; the intent enters `ESCROWED`.

Cost 2 583 OSU. **Level** ℓ_2 for the escrow, ℓ_3 before release. **Compensation** `xdom_refund` after `τ`, permissionless. **Errors** `E_CAP`, `E_FUNDS`, `E_DEADLINE`, `E_LIMIT`.

CHAPTER 20

Kernel Implementation on the SVM

One program, six instruction families, zero-copy state, and a compute budget that has to be defended line by line.

20.1 Program layout

`osk-svm` is a single program rather than a constellation, because on the SVM a CPI costs both compute and a stack frame, and the kernel's own subsystems call each other constantly. Its instruction data begins with the same syscall number as every other binding.

```
//! osk-svm — the os.fun kernel for the Solana Virtual Machine.
//! Single program, zero-copy state, no external dependencies beyond the
//! runtime and SPL Token. Every entry point implements exactly one syscall
//! from Appendix A.

use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    program_error::ProgramError,
    pubkey::Pubkey,
    sysvar::{clock::Clock, Sysvar},
};

entrypoint!(process);

pub fn process(program_id: &Pubkey, accounts: &[AccountInfo], data: &[u8])
    -> ProgramResult
{
    if data.len() < 3 { return Err(OsError::ShortInstruction.into()); }
    let nr = u16::from_le_bytes([data[0], data[1]]);
    let body = &data[3..];                       // data[2] = frame flags

    match nr & 0xff00 {
        0x0100 => cap::dispatch(program_id, accounts, nr, body),
        0x0200 => proc::dispatch(program_id, accounts, nr, body),
        0x0300 => mem::dispatch(program_id, accounts, nr, body),
        0x0400 => val::dispatch(program_id, accounts, nr, body),
        0x0500 => xdom::dispatch(program_id, accounts, nr, body),
        0x0600 => ns::dispatch(program_id, accounts, nr, body),
        0x0700 => evt::dispatch(program_id, accounts, nr, body),
        0x0800 => sys::dispatch(program_id, accounts, nr, body),
        _ => Err(OsError::UnknownSyscall.into()),
    }
}
```

20.2 Account layout

Every kernel account is a PDA, so its address proves which program minted it and under what seeds — the SVM's structural answer to the question "is this record authentic?"

Account	Seeds	Size	Rent-exempt deposit
Root	`["root", principal]`	64 B	0.00134 SOL
Capability	`["cap", principal, cap_id]`	96–160 B	0.00156–0.0020 SOL
PCB	`["proc", principal, pid]`	128 B	0.00178 SOL
Object	`["obj", ns_hash, key]`	32 B + payload	by Eq. (4.3)
Vault	`["vault", asset_mint]`	SPL token account, 165 B	0.00204 SOL
Escrow	`["escrow", intent_id]`	112 B	0.00167 SOL
Journal ring	`["jrnl", principal, page]`	8 KiB	0.0577 SOL
Replay bitmap	`["seen", principal, window]`	1 KiB	0.00786 SOL

All records are `#[repr(C)]` plain-old-data read through `bytemuck`, with no Borsh deserialization on the hot path. This is not micro-optimisation: Borsh decoding of a 96-byte capability costs on the order of 2 000 CU, which is comparable to the entire budget for `cap_check`.

```
/// 96-byte capability record, laid out exactly as Chapter 10's wire format
/// so that the same bytes can be hashed, signed and stored without
/// re-encoding.
#[repr(C)]
#[derive(Clone, Copy, bytemuck::Pod, bytemuck::Zeroable)]
pub struct CapRecord {
    pub id:        [u8; 16],
    pub subject:   [u8; 32],
    pub object:    [u8; 32],
    pub verbs:     u32,
    pub expiry:    [u8; 6],
    pub epoch:     u16,
    pub flags:     u32,
}

/// 64-byte root record. Counters live here, not in the children — this is
/// what makes Theorem 10.4 (non-amplification) hold under parallel
/// delegation.
#[repr(C)]
#[derive(Clone, Copy, bytemuck::Pod, bytemuck::Zeroable)]
pub struct RootRecord {
    pub principal:   [u8; 32],
    pub epoch:       u16,
    pub flags:       u16,
    pub spent_total: u128,      // against `total ≤ N` constraints
    pub window_start: u64,      // for `rate(N, W)` constraints
    pub window_count: u32,
    pub _pad:        [u8; 4],
}
```

20.3 A handler in full: `val_transfer`

```
/// 0x0402 · val_transfer — move an SPL or native asset within this domain.
/// CU budget: 17 000 (measured 14 812 warm, 16 940 with a fresh ATA).
pub fn val_transfer(
    program_id: &Pubkey,
```

```
    accounts: &[AccountInfo],
    body: &[u8],
) -> ProgramResult {
    let args = TransferArgs::decode(body)?;          // canonical, §19.4
    let it = &mut accounts.iter();

    let cap_ai    = next_account_info(it)?;
    let root_ai   = next_account_info(it)?;
    let seen_ai   = next_account_info(it)?;          // replay bitmap
    let src_ai    = next_account_info(it)?;
    let dst_ai    = next_account_info(it)?;
    let jrnl_ai   = next_account_info(it)?;
    let token_ai  = next_account_info(it)?;          // SPL Token program
    let auth_ai   = next_account_info(it)?;          // vault PDA authority

    // 1. authority — K1, K7. Costs ≈ 2 900 CU (Chapter 8's table).
    let principal = cap_check(
        &[cap_ai.clone(), root_ai.clone()],
        &args.cap,
        &Call::transfer(&args),
        Clock::get()?.slot,
    )?;

    // 2. replay — Proposition 16.3. One bit, one write.
    let digest = args.digest(program_id);
    if consume(seen_ai, &digest)? {
        return Ok(());                               // idempotent retry
    }

    // 3. budget — K3.
    charge_budget(root_ai, GAMMA_VAL_TRANSFER)?;

    // 4. effect. The vault authority is a PDA, so the kernel signs with
    //    seeds rather than holding a key (A1: no kernel-held private keys).
    let (vault_auth, bump) =
        Pubkey::find_program_address(&[b"vault", args.asset.as_ref()], program_id);
    if *auth_ai.key != vault_auth { return Err(OsError::BadVault.into()); }

    let ix = spl_token::instruction::transfer_checked(
        token_ai.key, src_ai.key, args.mint_ai_key(), dst_ai.key,
        &vault_auth, &[], args.amount, args.decimals,
    )?;
    solana_program::program::invoke_signed(
        &ix,
        &[src_ai.clone(), dst_ai.clone(), auth_ai.clone(), token_ai.clone()],
        &[&[b"vault", args.asset.as_ref(), &[bump]]],
    )?;

    // 5. journal — K4. Ring-buffer append, 48 bytes, ≈ 900 CU.
    journal_append(jrnl_ai, &JournalEntry::transfer(&principal, &args, &digest))?;
    Ok(())
}
```

Five phases, in the fixed order of §15.4, and each one maps to a line in the cost table of §8.4. The ordering is not stylistic: authority before replay before budget before effect before journal is what makes Theorem 7.8's proof apply to every handler uniformly, and the conformance suite checks the order statically by parsing the program's control-flow graph.

20.4 Compute budget accounting

Phase	CU	Share	Notes
Entry, dispatch, decode	1 180	8%	Zero-copy decode; no Borsh
`cap_check`	2 900	20%	2 PDA derivations, 2 reads, 2 constraints
Replay bitmap	1 240	8%	1 read, 1 write, 1 bit test
Budget charge	640	4%	Root record update
SPL `transfer_checked` CPI	7 950	54%	Dominated by the CPI itself
Journal append	900	6%	Ring-buffer write
Total	**14 810**	100%	Against a 17 000 CU request

The lesson visible in that table is that the kernel is not the cost — the CPI is. This is why the SVM kernel batches at the *instruction* level: a transaction carrying eight transfers pays the 1 180 CU entry cost once and the 7 950 CU CPI cost eight times, so kernel overhead falls from 46% to 12% of the transaction.

20.5 Program upgrade policy

```
  v0.x   upgrade authority = 3-of-5 multisig behind a 72 h timelock
         · every upgrade publishes the new program-data hash 72 h ahead
         · users may withdraw during the window (exit exceeds delay)
  v1.0   authority set to None — program becomes immutable
         · from then on, changes ship as a NEW program id and users
           migrate by re-granting capabilities (Invariant 3.4)
```

The timelock length is derived, not chosen: it must exceed the worst-case withdrawal path, which is bounded by the cross-domain settlement timeout $\tau_{\max}$ plus the longest lease reclamation. With $\tau_{\max} = 30$ min and a 24 h maximum lease grace period, 72 h leaves a comfortable margin.

20.6 Testing

```rust
#[tokio::test]
async fn transfer_is_idempotent_under_replay() {
    let mut ctx = OsTestContext::new().await;
    let (alice, cap) = ctx.principal_with_cap(verbs::TRANSFER, cap_usd(100)).await;
    let bob = ctx.principal().await;

    let ix = ctx.ix_val_transfer(&cap, &alice, &bob, USDC, 25_000_000);

    ctx.send(&[ix.clone()]).await.unwrap();          // first: applies
    ctx.send(&[ix.clone()]).await.unwrap();          // replay: no-op
    ctx.send(&[ix]).await.unwrap();                  // replay: no-op

    assert_eq!(ctx.balance(&bob, USDC).await, 25_000_000);   // K2
    assert_eq!(ctx.spent_osu(&alice).await, GAMMA_VAL_TRANSFER); // K3
}
```

CHAPTER 21

Kernel Implementation on the EVM

A suite of immutable contracts, one delegate, and a storage layout designed around the price of a cold slot.

21.1 The contract map

Contract	Role	Mutable?
`CapabilityRegistry`	Roots, capabilities, epochs, tombstones, constraint counters	no
`ProcessTable`	PCBs, plan roots, budgets	no
`ObjectStore`	Namespaced objects, leases, headers	no
`Vault`	Asset custody for kernel-held balances and escrow	no
`XDomSettler`	Escrow-and-fill state machine, attestation verification	no
`Journal`	Append-only per-principal sequence, packed	no
`Delegate7702`	The code an EOA delegates to; validation plus batch execution	no
`Paymaster`	ERC-4337 sponsorship and token reimbursement	operator config only
`AppRegistry`, `RevenueRouter`, `RevenueVault`, `Treasury`	Application revenue accounting (Chapter 34); already deployed	config

21.2 The capability registry

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

/// @title CapabilityRegistry
/// @notice The root of authority for os.fun on the EVM. Holds root records,
///         capability records, revocation epochs and constraint counters.
/// @dev    Immutable, no owner, no upgrade path. Every kernel contract calls
///         `check` before any state change (Invariant K1).
contract CapabilityRegistry is ICapabilityRegistry {
    // ── storage layout ──────────────────────────────────────────────
    // roots:  principal → Root       (1 slot: epoch|flags|windowStart|count)
    // spent:  principal → uint256    (1 slot: cumulative spend)
    // caps:   capId     → Cap        (3 slots, packed as below)
    //   slot0: subject(20) | verbs(4) | expiry(6) | epoch(2)
    //   slot1: object(32)
```

```
//   slot2: parent(16) | flags(4) | constraintCount(1) | reserved(11)
// cons:   capId,i    → bytes32   (one slot per constraint, ≤ 4)
// tomb:   capId      → bool      (packed into a bitmap by id prefix)

mapping(address principal => Root) private _roots;
mapping(bytes16 capId => Cap) private _caps;
mapping(bytes16 capId => mapping(uint256 => bytes32)) private _cons;
mapping(uint256 bucket => uint256 bits) private _tombs;

event Granted(bytes16 indexed capId, address indexed principal,
              bytes32 indexed object, uint32 verbs, uint48 expiry);
event Delegated(bytes16 indexed child, bytes16 indexed parent);
event Revoked(bytes16 indexed capId);
event EpochBumped(address indexed principal, uint16 epoch);

error BadCap();
error Expired();
error Revoked_();
error VerbNotGranted();
error NotNarrowing();
error CapLimit();

// ── minting ──────────────────────────────────────────────────────

/// @notice Mint a root capability. Only the principal may call this, and
///         only for authority they themselves hold.
function grant(
    bytes32 object,
    uint32 verbs,
    uint48 expiry,
    bytes32[] calldata constraints
) external returns (bytes16 capId) {
    if (constraints.length > MAX_CONSTRAINTS) revert CapLimit();
    Root storage r = _roots[msg.sender];
    if (r.epoch == 0) { r.epoch = 1; emit EpochBumped(msg.sender, 1); }

    capId = bytes16(keccak256(abi.encode(
        msg.sender, object, verbs, expiry, r.epoch, r.nonce++
    )));

    Cap storage k = _caps[capId];
    k.subject = msg.sender;
    k.object = object;
    k.verbs = verbs;
    k.expiry = expiry;
    k.epoch = r.epoch;
    k.constraintCount = uint8(constraints.length);
    for (uint256 i; i < constraints.length; ++i) {
        _cons[capId][i] = constraints[i];
    }
    emit Granted(capId, msg.sender, object, verbs, expiry);
}

/// @notice Delegate a narrowed copy. Enforces Definition 10.2's order:
///         verbs ⊆, expiry ≤, constraints ⊇ (appended, never removed).
function delegate(
    bytes16 parent,
```

```
        address subject,
        uint32 verbs,
        uint48 expiry,
        bytes32[] calldata extraConstraints
    ) external returns (bytes16 capId) {
        Cap storage p = _caps[parent];
        if (p.subject != msg.sender) revert BadCap();
        if (verbs & ~p.verbs != 0) revert NotNarrowing();
        if (expiry > p.expiry) revert NotNarrowing();
        if (_isTombstoned(parent)) revert Revoked_();

        uint256 total = p.constraintCount + extraConstraints.length;
        if (total > MAX_CONSTRAINTS) revert CapLimit();

        capId = bytes16(keccak256(abi.encode(parent, subject, verbs, expiry,
                                             _roots[p.rootOwner].nonce++)));
        Cap storage k = _caps[capId];
        k.subject = subject;
        k.object = p.object;
        k.verbs = verbs;
        k.expiry = expiry;
        k.epoch = p.epoch;
        k.parent = parent;
        k.rootOwner = p.rootOwner;          // counters stay at the root
        k.constraintCount = uint8(total);

        for (uint256 i; i < p.constraintCount; ++i) {
            _cons[capId][i] = _cons[parent][i];
        }
        for (uint256 i; i < extraConstraints.length; ++i) {
            _cons[capId][p.constraintCount + i] = extraConstraints[i];
        }
        emit Delegated(capId, parent);
    }

    // ── revocation: O(1) bulk, O(1) selective ──────────────────────────

    function revokeAll() external returns (uint16 epoch) {
        Root storage r = _roots[msg.sender];
        unchecked { epoch = ++r.epoch; }
        emit EpochBumped(msg.sender, epoch);
    }

    function revoke(bytes16 capId) external {
        Cap storage k = _caps[capId];
        if (k.rootOwner != msg.sender && k.subject != msg.sender) revert BadCap();
        (uint256 bucket, uint256 bit) = _tombIndex(capId);
        _tombs[bucket] |= (1 << bit);
        emit Revoked(capId);
    }

    // ── the hot path ───────────────────────────────────────────────────

    /// @notice Check a capability against a concrete call. Called by every
    ///         kernel contract as its first statement (K1, K7).
    /// @return principal The account whose authority this capability derives
    ///         from — the identity all accounting is attributed to.
```

```
    function check(bytes16 capId, Call calldata call, bytes calldata sig)
        external
        returns (address principal)
    {
        Cap storage k = _caps[capId];
        principal = k.rootOwner == address(0) ? k.subject : k.rootOwner;

        Root storage r = _roots[principal];
        if (k.epoch != r.epoch) revert Revoked_();          // K6
        if (_isTombstoned(capId)) revert Revoked_();
        if (block.timestamp > k.expiry) revert Expired();
        if (call.verb & k.verbs == 0) revert VerbNotGranted();

        _authenticate(k, call, sig);
        _evalConstraints(capId, k, call, r);                // counters on root
    }
}
```

21.3 Storage layout and its price

The layout above is chosen so that the hot path touches three slots, two of them warm after the first access:

Access	Gas	Why
`_caps[capId]` slot 0 (cold)	2 100	Unavoidable: the capability's identity
`_caps[capId]` slots 1–2 (warm)	200	Same 32-byte-aligned group
`_roots[principal]` (cold)	2 100	Epoch + counters in one slot
`_tombs[bucket]` (cold, shared)	2 100	One slot per 256 capabilities
Constraint counter write	2 900	Non-zero → non-zero
Typical `check`	**≈ 9 400 cold, 3 400 warm**	Warm case dominates in batches, which is the point of batching

Two design consequences are visible here. Tombstones are bitmapped 256 to a slot so that revoking many capabilities in the same family costs one cold slot rather than n. And constraint counters live on the root record, which is both a correctness requirement (Theorem 10.4) and an efficiency one — the root slot is already warm from the epoch check.

21.4 ERC-4337 integration

```
/// @notice Account-abstraction validation for os.fun accounts (ERC-4337). The
signature
///         field of a UserOperation carries (capId, sig); validation is
///         exactly a capability check, so 4337 and 7702 paths share one
///         authority model.
function validateUserOp(
    PackedUserOperation calldata op,
    bytes32 userOpHash,
    uint256 missingAccountFunds
) external returns (uint256 validationData) {
```

```
    (bytes16 capId, bytes calldata sig) = _split(op.signature);

    Call memory call = Call({
        verb:   VERB_EXECUTE,
        target: address(this),
        value:  0,
        digest: userOpHash
    });

    // Reverts on failure; 4337 requires we return a code instead, so the
    // registry's view variant is used and mapped to the standard encoding.
    (bool ok, uint48 validUntil, uint48 validAfter) =
        caps.tryCheck(capId, call, sig);
    if (!ok) return SIG_VALIDATION_FAILED;

    if (missingAccountFunds != 0) {
        (bool sent,) = payable(msg.sender).call{value: missingAccountFunds}("");
        sent;   // EntryPoint handles failure; deliberately not reverting here
    }
    return _packValidationData(false, validUntil, validAfter);
}
```

The single most important line is the one that maps a capability check to the standard's `validationData` encoding: the `validUntil` / `validAfter` fields carry the capability's expiry, so the bundler and the EntryPoint enforce expiry for free, before our code runs. Reusing infrastructure this way is how the kernel stays small.

21.5 Transient storage as the kernel context

Within one transaction, several kernel contracts need shared context: which principal is acting, which capability authorised it, how much budget is left. Passing this through calldata costs bytes (Eq. (4.1)); storing it in storage costs 2 900 gas per write and must be cleared. EIP-1153 transient storage [12] is exactly right: 100 gas per access and automatically discarded at the end of the transaction.

```
library KernelCtx {
    // keccak256("os.fun.kernel.ctx.v1") - 1
    uint256 private constant SLOT_PRINCIPAL = 0x8f2c...;
    uint256 private constant SLOT_CAP       = 0x8f2c... + 1;
    uint256 private constant SLOT_BUDGET    = 0x8f2c... + 2;

    function enter(address principal, bytes16 cap, uint32 budget) internal {
        assembly {
            tstore(SLOT_PRINCIPAL, principal)
            tstore(SLOT_CAP, cap)
            tstore(SLOT_BUDGET, budget)
        }
    }

    function spend(uint32 osu) internal {
        assembly {
            let left := tload(SLOT_BUDGET)
            if lt(left, osu) {
                mstore(0x00, 0x8b7a1c31)          // E_BUDGET selector
                revert(0x1c, 0x04)
            }
```

```
            tstore(SLOT_BUDGET, sub(left, osu))   // K3, 100 gas
        }
    }
}
```

21.6 Immutability and what it costs

No contract on the authority or asset path has an admin, an upgrade hook, a pause switch or a proxy. This is a real cost: a bug is fixed by deploying a new suite and having every user re-delegate, which takes one signature per user and a migration period measured in weeks.

We accept it because the alternative is worse in a way the rest of this document has already argued. An upgradeable capability registry is a registry whose meaning can change after the user consented, which contradicts A7; a pausable vault is a vault with a custodian, which contradicts A6. The industry convention of "immutable except for the multisig that can replace everything" is not immutability, and describing it as such is precisely the illegibility A7 exists to eliminate.

The one concession is `Paymaster`, which holds only the operator's own funds and can be reconfigured or drained by the operator without touching user authority — a contract whose worst-case failure is that sponsorship stops.

21.7 Invariant tests

```
/// Foundry invariant suite. These are the executable form of §15.3; the
/// fuzzer drives arbitrary sequences of kernel calls and asserts after each.
contract KernelInvariants is Test {
    function invariant_K2_conservation() public view {
        uint256 sum = vault.totalHeld(USDC) + settler.totalEscrowed(USDC);
        assertEq(sum, usdc.balanceOf(address(vault)) +
usdc.balanceOf(address(settler)));
    }

    function invariant_K3_budget() public view {
        for (uint256 i; i < handler.pidCount(); ++i) {
            Pcb memory p = procs.get(handler.pidAt(i));
            assertLe(p.spentOsu, p.budgetOsu);
        }
    }

    function invariant_K5_escrow_exclusivity() public view {
        for (uint256 i; i < handler.intentCount(); ++i) {
            Intent memory it = settler.get(handler.intentAt(i));
            assertFalse(it.released && it.refunded);
        }
    }

    function invariant_K6_epoch_coherence() public view {
        for (uint256 i; i < handler.capCount(); ++i) {
            bytes16 id = handler.capAt(i);
            if (caps.passesCheck(id)) {
                assertEq(caps.epochOf(id), caps.currentEpoch(caps.principalOf(id)));
            }
        }
    }
}
```

PART IV

Userland

The part that makes this an operating system rather than a protocol. A userland program is written in an ordinary language, compiled to an ordinary instruction set, executed deterministically with a metered budget, and settled against the domains by proof. We specify the process model, the machine (OSVM), the Python dialect that targets it, a complete worked application, the path for other languages, and the verification that binds userland results to onchain state.

CHAPTER 22

The Userland Model

A program is an image, a manifest, a capability set and a budget. Nothing else is granted, and nothing else is needed.

22.1 What a program is

DEFINITION 22.1 · Userland program

A *program* is a pair (I, M) where I is an *image* — a statically linked RV64IM binary with a fixed entry point — and M is a *manifest* declaring the program's identity, entry points, requested capabilities and resource limits. Its identity is $\mathrm{H}(I) \parallel \mathrm{H}(M)$, and this hash is what a user authorises, what the registry records, and what every execution receipt commits to.

The image hash being the identity has a consequence worth stating early: an app cannot change after you install it. There is no server-side deploy that silently alters behaviour, because the program that runs is the program whose hash your capability names (Invariant 3.4, applied to userland).

22.2 The manifest

```
# app.toml — declared once, checked at install, enforced at every syscall.
[app]
name        = "tipjar"
version     = "1.4.0"
publisher   = "os://base/acct/0x9f21…4c07"
description = "Send a few dollars to anyone, in whatever they hold."

[entry]
main   = "tipjar.main"          # invoked by the launcher
on_tip = "tipjar.on_tip"        # invoked by an event subscription

[capabilities]                  # the exact authority the app will ask for
"val.transfer" = { asset = "os://*/asset/usd", max_per_call = "50.00",
                   max_total = "500.00", window = "24h" }
"ns.resolve"   = { prefix = "os://*/acct/" }
"evt.emit"     = { topics = ["tipjar.*"] }
"mem.alloc"    = { ns = "/acct/{principal}/apps/tipjar", max_bytes = 65536 }

[limits]
cycles      = 40_000_000        # OSVM cycle ceiling per invocation
memory_mib  = 16
osu         = 12_000            # domain-side budget per invocation
timeout_ms  = 3_000

[ui]
surface     = "window"          # window | widget | headless
min_size    = [420, 320]
```

The `[capabilities]` block is the entire permission dialog. Because the constraint language is the same one the kernel enforces (§10.1), the sentence a user is shown — "tipjar may send up to $50 at a time,

$500 a day, in dollars only" — is not marketing copy written by the app author; it is a rendering of the exact constraint set that ring 0 will check. That is A7 made mechanical.

22.3 Where a program runs

Placement	When	Integrity	Cost
In-domain	Hot, small, value-critical logic: an AMM hook, a settlement rule	Consensus	Highest: gas/CU per operation
OSVM (verified)	Everything else: app logic, feeds, matching, game state, agents	Proof or challenge	Compute + settlement amortised
Local (trusted)	Rendering, animation, input handling, caches	None needed	Free

A single application normally uses all three. `tipjar` renders locally, computes its logic in OSVM, and touches the domain only for `val_transfer` — which is the placement function π of Definition 2.2 applied by an app author rather than by a kernel author.

22.4 The process boundary

An OSVM process has no ambient input or output. It has:

- **Memory**: a private, page-addressed address space (§23.4).
- **Registers and a cycle budget**: bounded by the manifest.
- **One trap**: `ecall`, into the syscall table of Chapter 19.
- **Its inputs**: an input frame, committed by hash in the receipt.

Everything a conventional program gets from the operating system implicitly — the clock, the random number generator, the filesystem, the network — is a syscall here, with a declared cost and a declared commitment level. That is what makes execution reproducible, and reproducibility is what makes it verifiable.

> **INVARIANT 22.2 · U1 — No hidden inputs**
> A process's output is a pure function of (image, manifest, input frame, syscall results). Any implementation that reads an unlogged input is non-conforming and will be detected by the replay check of §27.2.

22.5 Drawing

Programs do not draw pixels; they return a view. The shell renders it.

```
from osfun import ui

def view(state) -> ui.Node:
    return ui.column(
        ui.title(f"tipjar · {state.sent_today} sent today"),
```

```
        ui.field("who", placeholder="@handle or os:// name"),
        ui.amount("how_much", currency="USD", max=50),
        ui.button("send", label="Send", intent="tipjar.send"),
        ui.list(
            [ui.row(t.who, ui.money(t.amount), ui.time(t.at))
             for t in state.recent],
        ),
    )
```

This is a deliberate constraint rather than a convenience. A view is data, so it can be rendered by the shell in a window, by a terminal as text, by a notification as a summary line, and — importantly — it can be *shown to the user as part of a consent dialog* before the intent it triggers is authorised. An app that could draw arbitrarily could draw a fake consent dialog; an app that returns a view tree cannot, because the shell owns the chrome.

CHAPTER 23

OSVM: A Deterministic Process Machine

RV64IM, because the boring choice is the one with compilers, and the provable choice is the one with a small instruction set.

23.1 Why an ordinary instruction set

The temptation when designing an onchain VM is to invent one. We do not, for four reasons that between them determine the entire chapter.

1. **Compilers already exist.** LLVM, GCC, Rust, TinyGo, and every C toolchain target RV64 today. A new ISA means a new backend for every language, and therefore, in practice, one language.
2. **Proof systems already exist.** The mature zkVM ecosystem [41] proves RISC-V execution. Choosing RV64IM means the verification layer of Chapter 27 is an integration rather than a research project.
3. **The M extension is exactly enough.** Integer multiply and divide are needed; floating point is not (§24.2 explains why we exclude F/D deliberately); atomics are not (single-threaded); compressed instructions save image size but complicate proving, so they are decoded but normalised.
4. **A small ISA is auditable.** RV64IM [42] is 47 instructions (Appendix E). The one-step transition function that a fraud proof must implement onchain is therefore small enough to write in Solidity and read in an afternoon.

23.2 Machine state

DEFINITION 23.1 · OSVM state

An OSVM state is

$$\omega = (x, \mathrm{pc}, M, k, \varphi) \tag{23.1}$$

where $x \in (\{0,1\}^{64})^{32}$ are the registers with x_0 hardwired to zero, `pc` the program counter, $M : [0, 2^{32}) \to \{0,1\}^8$ a paged memory, $k \in \mathbb{N}$ the consumed cycle count, and φ the syscall log. The *commitment* to a state is

$$\mathrm{C}(\omega) = \mathrm{H}(\mathrm{H}(x) \,\|\, \mathrm{pc} \,\|\, \mathrm{root}(M) \,\|\, k \,\|\, \mathrm{H}(\varphi)) \tag{23.2}$$

where `root(M)` is the Merkle root over 4 KiB pages.

Committing to the whole machine state in 32 bytes is what makes the bisection game of §27.3 possible: two parties who disagree about a long execution can bisect on `C(ω)` until they reach a single instruction, and only that instruction needs to be executed onchain.

23.3 Metering

DEFINITION 23.2 · Cycle cost

Each instruction consumes cycles by class:

$$c(\mathrm{op}) = \begin{cases} 1 & \text{arithmetic, logic, compare, branch} \\ 2 & \text{load, store (page resident)} \\ 4 & \text{multiply} \\ 8 & \text{divide, remainder} \\ 64 + 2 \cdot \lceil n/32 \rceil & \text{page fault, } n \text{ bytes proven} \\ \text{declared} & \text{ecall (Appendix A)} \end{cases} \tag{23.3}$$

and a step is admissible only if $k + c \le k_{\max}$.

THEOREM 23.3 · Termination
Every OSVM execution halts within $k_{\max}$ steps of the machine's cycle counter, regardless of the program.

PROOF. Every instruction has $c \ge 1$ by the table, so k strictly increases at each step. The step function refuses to execute when $k + c > k_{\max}$ and halts with `E_CYCLES`. Since k is a strictly increasing sequence of naturals bounded by $k_{\max}$, at most $k_{\max}$ steps occur. □

That theorem is the reason a userland program can be arbitrary Python and still be safe to run inside a system that must never hang: the halting problem is not solved, it is priced.

23.4 Memory

Memory is a flat 32-bit space divided into 4 KiB pages, with a Merkle tree over page hashes. A page is *resident* if the executor holds it; a non-resident access is a page fault, which in this machine is not an error but a proof obligation.

$$\text{proof size}(n_{\text{pages}}) = 32 \cdot \lceil \log_2 n_{\text{pages}} \rceil \text{ bytes} \tag{23.4}$$

For a 16 MiB address space, $n_{\text{pages}} = 4096$ and a page proof is 384 bytes. The whole memory root changes with each write, so the executor maintains the tree incrementally: a write costs $\lceil \log_2 n_{\text{pages}} \rceil$ hash updates, which at 12 levels is negligible against the cost of proving.

Region	Contents
`0x0000_1000` – `0x0FFF_FFFF`	Program image: text, rodata, data (read-only above `_etext`)
`0x1000_0000` – `0x1FFF_FFFF`	Heap, grown by the runtime's bump allocator
`0x2000_0000` – `0x2000_0FFF`	Input frame (read-only), committed by hash
`0x2001_0000` – `0x2001_FFFF`	Syscall frame: arguments and results
`0x7FFF_0000` – `0x7FFF_FFFF`	Stack, grows down, guard page at the bottom

23.5 Determinism

THEOREM 23.4 · Bit-reproducibility
Let A and B be conforming OSVM implementations. For any image I, input frame z, and syscall result log φ, the executions agree at every step: the register file, memory root, cycle count and output are identical.

PROOF. RV64IM's semantics are fully specified with no implementation-defined behaviour for the subset we admit: integer overflow wraps, shifts use the low 6 bits, division by zero yields the architecturally defined values (-1 for `DIV`, the dividend for `REM`), and unaligned access is normalised

by the decoder into aligned accesses. There is no floating point (excluded), no concurrency (single hart), no interrupts, and no undefined memory: pages read before being written return zero. The only external inputs are z and φ, both fixed by hypothesis. Therefore each step's transition is a function of the previous state alone, and by induction the traces coincide. □

The parenthetical exclusions are the whole content of the theorem. Every one of them is a place where a less careful design admits divergence, and §24.2 walks through what each would have cost us in a Python runtime.

23.6 The interpreter core

```
//! OSVM — deterministic RV64IM interpreter with Merkleised memory and
//! metered cycles. This is the reference implementation: correctness and
//! auditability over speed. The production executor is a JIT that must
//! produce identical traces (§37.2 conformance vectors).

pub struct Vm {
    pub x:       [u64; 32],
    pub pc:      u64,
    pub mem:     PagedMemory,     // Merkleised, 4 KiB pages
    pub cycles:  u64,
    pub budget:  u64,
    pub log:     SyscallLog,      // ϕ: the committed result stream
    pub halted:  Option<Halt>,
}

impl Vm {
    pub fn step(&mut self) -> Result<(), Halt> {
        let inst = self.mem.load_u32(self.pc)?;
        let (cost, next) = self.exec(inst)?;
        self.charge(cost)?;                       // Theorem 23.3
        self.pc = next;
        Ok(())
    }

    #[inline]
    fn charge(&mut self, c: u64) -> Result<(), Halt> {
        self.cycles = self.cycles.saturating_add(c);
        if self.cycles > self.budget { return Err(Halt::Cycles); }
        Ok(())
    }

    fn exec(&mut self, i: u32) -> Result<(u64, u64), Halt> {
        let opcode = i & 0x7f;
        let rd     = ((i >> 7)  & 0x1f) as usize;
        let rs1    = ((i >> 15) & 0x1f) as usize;
        let rs2    = ((i >> 20) & 0x1f) as usize;
        let f3     = (i >> 12) & 0x7;
        let f7     = i >> 25;
        let pc4    = self.pc.wrapping_add(4);

        match opcode {
            // ── OP: register-register ────────────────────────────
            0x33 => {
                let (a, b) = (self.x[rs1], self.x[rs2]);
                let (v, cost) = match (f7, f3) {
```

```
            (0x00, 0x0) => (a.wrapping_add(b), 1),
            (0x20, 0x0) => (a.wrapping_sub(b), 1),
            (0x00, 0x1) => (a << (b & 63), 1),
            (0x00, 0x2) => (((a as i64) < (b as i64)) as u64, 1),
            (0x00, 0x3) => ((a < b) as u64, 1),
            (0x00, 0x4) => (a ^ b, 1),
            (0x00, 0x5) => (a >> (b & 63), 1),
            (0x20, 0x5) => (((a as i64) >> (b & 63)) as u64, 1),
            (0x00, 0x6) => (a | b, 1),
            (0x00, 0x7) => (a & b, 1),
            // ── M extension ──────────────────────────────
            (0x01, 0x0) => (a.wrapping_mul(b), 4),
            (0x01, 0x1) => (mulh_ss(a, b), 4),
            (0x01, 0x2) => (mulh_su(a, b), 4),
            (0x01, 0x3) => (mulh_uu(a, b), 4),
            (0x01, 0x4) => (div_s(a, b), 8),   // b==0 ⇒ -1 (spec)
            (0x01, 0x5) => (div_u(a, b), 8),   // b==0 ⇒ u64::MAX
            (0x01, 0x6) => (rem_s(a, b), 8),   // b==0 ⇒ a
            (0x01, 0x7) => (rem_u(a, b), 8),
            _ => return Err(Halt::IllegalInstruction(i)),
        };
        self.set(rd, v);
        Ok((cost, pc4))
    }

    // ── OP-IMM ──────────────────────────────────────────
    0x13 => {
        let imm = sext(i >> 20, 12);
        let a = self.x[rs1];
        let v = match f3 {
            0x0 => a.wrapping_add(imm),
            0x1 => a << ((i >> 20) & 63),
            0x2 => (((a as i64) < (imm as i64)) as u64),
            0x3 => ((a < imm) as u64),
            0x4 => a ^ imm,
            0x5 if f7 & 0x20 == 0 => a >> ((i >> 20) & 63),
            0x5 => ((a as i64) >> ((i >> 20) & 63)) as u64,
            0x6 => a | imm,
            0x7 => a & imm,
            _ => return Err(Halt::IllegalInstruction(i)),
        };
        self.set(rd, v);
        Ok((1, pc4))
    }

    // ── LOAD / STORE ────────────────────────────────────
    0x03 => {
        let addr = self.x[rs1].wrapping_add(sext(i >> 20, 12));
        let (v, extra) = self.mem.load(addr, width(f3))?;
        self.set(rd, extend(v, f3));
        Ok((2 + extra, pc4))              // extra = page-fault cycles
    }
    0x23 => {
        let imm = sext(((i >> 25) << 5) | ((i >> 7) & 0x1f), 12);
        let addr = self.x[rs1].wrapping_add(imm);
        let extra = self.mem.store(addr, self.x[rs2], width(f3))?;
        Ok((2 + extra, pc4))
```

```
            }

            // ── control flow ──────────────────────────────────────
            0x63 => {
                let (a, b) = (self.x[rs1], self.x[rs2]);
                let taken = match f3 {
                    0x0 => a == b,
                    0x1 => a != b,
                    0x4 => (a as i64) <  (b as i64),
                    0x5 => (a as i64) >= (b as i64),
                    0x6 => a <  b,
                    0x7 => a >= b,
                    _ => return Err(Halt::IllegalInstruction(i)),
                };
                let target = if taken { self.pc.wrapping_add(bimm(i)) } else { pc4 };
                Ok((1, target))
            }
            0x6f => { self.set(rd, pc4); Ok((1, self.pc.wrapping_add(jimm(i)))) }
            0x67 => {
                let t = self.x[rs1].wrapping_add(sext(i >> 20, 12)) & !1;
                self.set(rd, pc4);
                Ok((1, t))
            }
            0x37 => { self.set(rd, sext(i & 0xffff_f000, 32)); Ok((1, pc4)) }
            0x17 => { self.set(rd, self.pc.wrapping_add(sext(i & 0xffff_f000, 32)));
                      Ok((1, pc4)) }

            // ── SYSTEM: the only door out of the machine ──────────
            0x73 if i == 0x0000_0073 => {
                let cost = self.ecall()?;          // Chapter 19 binding
                Ok((cost, pc4))
            }
            0x73 if i == 0x0010_0073 => Err(Halt::Break),

            _ => Err(Halt::IllegalInstruction(i)),
        }
    }

    /// The syscall bridge. Arguments are read from registers and the syscall
    /// frame; the result is appended to the log φ so that a verifier replays
    /// the same values without re-executing the kernel (Invariant U1).
    fn ecall(&mut self) -> Result<u64, Halt> {
        let nr = self.x[17] as u16;                       // a7
        let args = [self.x[10], self.x[11], self.x[12],
                    self.x[13], self.x[14], self.x[15]];  // a0..a5
        let entry = self.log.next_for(nr, &args).ok_or(Halt::UnloggedSyscall)?;
        self.x[10] = entry.ret_lo;
        self.x[11] = entry.ret_hi;
        if !entry.out.is_empty() {
            self.mem.write_frame(SYSCALL_FRAME, &entry.out)?;
        }
        Ok(entry.declared_cost)
    }
}
```

The `ecall` implementation is where this machine differs from a general emulator, and it is worth dwelling on. The VM does not *perform* syscalls; it *consumes their results from a log*. Execution is

therefore a pure function of (image, input, log), which is exactly the hypothesis of Theorem 23.4, and the log is what a verifier checks against the chain. A syscall that appears in execution but not in the log halts the machine — a program cannot invent an input.

23.7 Execution receipts

```
/// What an executor publishes. 168 bytes; this is the object the settlement
/// layer of Chapter 27 accepts, challenges, or proves.
#[repr(C)]
pub struct Receipt {
    pub image:      [u8; 32],   // H(I) ‖ H(M)
    pub input:      [u8; 32],   // H(input frame)
    pub log_root:   [u8; 32],   // merkle root over syscall results ϕ
    pub out:        [u8; 32],   // H(output frame)
    pub state_root:[u8; 32],    // C(ω) at halt, Eq. (23.2)
    pub cycles:     u64,
    pub halt:       u32,        // 0 = normal, else the Halt discriminant
    pub _pad:       u32,
}
```

23.8 Performance

The numbers that matter are cycles per second in each execution mode, because they set the boundary between what can run interactively and what must be batched.

Mode	Throughput	Use
Reference interpreter (Rust, this chapter)	≈ 25 M cycles/s	Conformance, fraud-proof replay, debugging
JIT executor	≈ 900 M cycles/s	The default path for live apps
Proving (STARK, commodity CPU)	≈ 0.3–1.5 M cycles/s	Only when a succinct proof is required (§27.4)
Onchain one-step verification	1 instruction	The last move of a bisection game

A typical app invocation — parse an input, update some state, format a view — is 2–10 M cycles, so the JIT runs it in about 10 ms and the interpreter in about 300 ms. Proving the same execution takes seconds, which is why the default settlement path is optimistic and proofs are the exception.

CHAPTER 24

Python on the OS

Ordinary Python, made deterministic by construction rather than by convention.

24.1 The goal, stated precisely

We want a person who knows Python to write an onchain application without learning a new language, and we want that program's execution to be verifiable by strangers. Those two goals are in tension only if the language is used carelessly; this chapter is the careful use.

> **DEFINITION 24.1 · osPy**
> *osPy* is the subset of Python 3.12 syntax accepted by the os.fun compiler, together with a runtime semantics in which every operation is deterministic and metered. A program is *osPy-conforming* if it parses under the grammar of Appendix D, passes the determinism checks of §24.2, and its transitive imports are limited to the `osfun` standard library.

24.2 Every place Python is non-deterministic, and what we do

This table is the technical core of the chapter. Each row is a real source of divergence between two runs of the same CPython program, and each has a rule.

Source	Why it diverges	osPy rule
`hash()` randomisation	`PYTHONHASHSEED` randomises string hashing per process, changing set iteration order	Fixed-seed SipHash-1-3 compiled into the runtime; `hash()` is a pure function of the value
`set` iteration order	Depends on hash values and insertion history	`set` is an insertion-ordered container in osPy; iteration order is insertion order
`dict` order	Insertion-ordered since 3.7 — already deterministic	Kept; guaranteed by the runtime rather than by convention
Floating point	`math` functions call libm, which differs across platforms; FMA contraction differs across compilers	`float` is **not** in osPy. Money is `Decimal` (128-bit fixed point); `math` is a soft-float library compiled into the image and included in the image hash
`id()`, `is` on values	Address-dependent	`id()` is unavailable; `is` is permitted only against `None`, `True`, `False`
Garbage collection	Finalizer timing is implementation-dependent	Region-based allocation with deterministic scope exit; `__del__` is not supported

`time`, `random`	Wall clock and OS entropy	Only via `osfun.time` / `osfun.rand`, which are syscalls with a commitment level (§19.2)
I/O, sockets, subprocess	Ambient authority	Not present. All effects are syscalls
Threads, async	Scheduling non-determinism	Single-threaded; `async` is accepted but scheduled by a deterministic round-robin over ready tasks
Exception messages	Contain addresses and paths	Messages are normalised; tracebacks contain source positions only
Unicode	Normalisation and locale-dependent case mapping	Strings are UTF-8 with explicit NFC normalisation at construction; `str.lower()` uses a frozen table shipped in the runtime
Integer size	Arbitrary precision — deterministic but unbounded cost	Kept arbitrary-precision, but **metered per limb** so cost is bounded
`sys.setrecursionlimit`	Stack depth differs	Fixed at 512 frames; exceeding it is `RecursionError` deterministically

The single most consequential row is floating point. Excluding `float` is unpopular and correct: two conforming implementations must agree bit for bit, and no practical language runtime achieves that for transcendental functions across platforms. Money is decimal anyway, and the programs that genuinely need real arithmetic get a soft-float library whose exact behaviour is part of the image hash — so it is deterministic by the same argument as the rest of the program.

24.3 What is in the language

Feature	Status
Functions, closures, default and keyword arguments	Full
Classes, single inheritance, `@property`, `@staticmethod`	Full
Multiple inheritance, metaclasses, `__slots__` tricks	Rejected
Comprehensions, generators, `yield`	Full
`async` / `await`	Full, deterministic scheduler
Exceptions, `try` / `except` / `finally`, custom exception types	Full
`match` statements	Full
f-strings	Full
Type annotations	Full; used for storage layout and optimisation

`eval`, `exec`, `compile`, `__import__`	Rejected — no dynamic code
Monkey-patching builtins	Rejected at compile time
C extensions	Not applicable; pure-Python dependencies vendored into the image

24.4 State, typed

Persistent state is declared, not discovered, so that the compiler can compute an object layout and the kernel can price it.

```
from osfun import state, USD, Address, Timestamp

@state.model(ns="/acct/{principal}/apps/tipjar")
class TipJar:
    total_sent:   USD          = USD(0)
    tip_count:    int          = 0
    last_tip_at:  Timestamp    = Timestamp(0)
    recipients:   dict[Address, USD] = state.field(max_entries=512)

    @state.index("by_amount")
    def _idx(self) -> USD:
        return self.total_sent
```

The compiler turns this into an object layout with fixed offsets, a lease size, and an allocation cost known at build time:

```
  layout: TipJar   (ns /acct/{principal}/apps/tipjar, object kind 0x07)
  ┌────────┬──────┬──────────────┬──────────────────────────────────────┐
  │ offset │ size │ field        │ notes                                │
  ├────────┼──────┼──────────────┼──────────────────────────────────────┤
  │    0   │  32  │ (header)     │ §17.1                                │
  │   32   │  16  │ total_sent   │ Decimal128, 6 dp                     │
  │   48   │   8  │ tip_count    │ u64 (compiler proved non-negative)   │
  │   56   │   8  │ last_tip_at  │ domain time                          │
  │   64   │  24  │ recipients   │ (root, len, cap) — paged map §17.5   │
  └────────┴──────┴──────────────┴──────────────────────────────────────┘
  static size 88 B → lease deposit 0.0015 SOL / 3 slots ≈ 62 400 gas
  recipients: paged, 512 × 48 B = 24 KiB → DA-backed, root onchain
```

24.5 Finality in the language

The finality calculus of Chapter 13 is not a separate tool; it is part of the type checker, and in ordinary code it is invisible. Stamps are inferred (Proposition 13.4), effect requirements come from the syscall table, and the programmer writes an annotation only where they want something other than the default.

```
from osfun import wallet, oracle, level, USD

async def pay_out(winner: Principal, pot: USD) -> None:
    price = oracle.usd_per_sol()          # : Price @ {sol: confirmed}
    amount = pot / price                  # : SOL   @ {sol: confirmed}   (meet)
```

```
    wallet.transfer(winner, amount)       # requires confirmed — ok

    # The same value cannot drive a settlement, which requires `final`:
    #   xdom.claim(intent, att)           # error F0102 (§13.6)

    att = await level.settle(att, sol="final")   # +12.4 s, may raise Retracted
    xdom.claim(intent, att)               # now ok
```

Three surface forms carry the whole system:

Form	Meaning
`x: USD @ final`	An explicit stamp, checked against inference. Written at API boundaries, almost never inside a function
`await level.settle(v, sol="final")`	The lift of Definition 12.5. It blocks, it costs the latency the optimiser predicts, and it may raise `Retracted`
`@on_retract(handler)`	Registers the compensation that T-EFF requires for effects below `final` (Theorem 13.3)

WHAT THIS REPLACES

Today the same intent is expressed as a comment (`# wait 12 blocks first`), a constant in a config file, or nothing at all. Here it is a type, an inferred one, whose violation is a compile error carrying a suggested fix and the price of that fix in seconds. The `circle` application of Chapter 25 is checked this way: its payout is an ℓ_2 effect with a registered compensation, and its cross-domain claim is an ℓ_3 effect, and the compiler proves the difference rather than the author remembering it.

24.6 The compiler pipeline

osPy — compilation

```
tipjar.py
   │  1. parse            CPython-compatible grammar (Appendix D)
   ▼
AST ──────── 2. determinism pass ── rejects float, eval, threads, id(),
   │                                unbounded recursion, foreign imports
   ▼
typed IR ──── 3. type & effect inference
   │            · every expression gets a type
   │            · every call gets an effect set (pure | syscall | alloc)
   │            · @state models get layouts (§24.4)
   ▼
SSA ──────── 4. optimisation: constant folding, escape analysis,
   │            refcount elision, bounds-check elimination
   ▼
RV64IM ────── 5. codegen (LLVM backend, -O2, no float, no atomics)
   │            + runtime library (allocator, bigint, str, dict, tasks)
   ▼
image ─────── 6. link, strip, canonicalise, hash
                · deterministic section order, zeroed padding
                · H(I) is stable across machines and compiler runs
```

Figure 24.1 Source to image. The typed IR is where determinism checks and metering annotations are applied; everything after it is ordinary compilation to an ordinary target.

Step 6 deserves emphasis: the compiler is itself reproducible. Two builds of the same source on different machines produce byte-identical images, because section order is fixed, padding is zeroed, symbol tables are stripped and no timestamps are embedded. Without that property, "the app you authorised" would be a hash nobody could independently reproduce.

24.7 The runtime

The runtime library is about 180 KiB of RV64IM and provides what CPython provides, minus the parts that cannot be deterministic.

Component	Design	Cycles (typical)
Allocator	Bump allocation in regions; regions freed at scope exit; no free list, no compaction	12 per alloc
Reference counting	Compile-time elided where escape analysis proves a single owner; otherwise inline increment/decrement	2 per op
Integers	Small-int fast path (63-bit tagged); big integers as limb vectors, metered per limb	1 small, 6 per limb
Strings	UTF-8, immutable, interned constants; NFC at construction	3 + 1 per byte
`dict` / `set`	Open addressing with insertion-order vector; fixed-seed SipHash-1-3 [43]	28 per lookup
Tasks	Deterministic round-robin over ready queue; `await` yields to the next task in insertion order	40 per switch
Exceptions	Zero-cost tables; unwinding walks a static table	0 when not raised

Decimal	128-bit fixed point, 6 decimal places, banker's rounding	4 add, 9 multiply

24.8 The standard library

```
osfun.wallet     balances, transfers, resolution   → val_*, ns_resolve
osfun.assets     asset descriptors, quotes, swaps  → val_quote, val_swap
osfun.state      persistent models and indexes     → mem_*
osfun.caps       capability introspection          → cap_check, cap_list
osfun.events     emit and subscribe                → evt_*
osfun.time       domain clocks and commitment      → sys_time
osfun.rand       committed randomness              → sys_rand
osfun.xdom       cross-domain intents              → xdom_*
osfun.ui         view trees                        → (local, no syscall)
osfun.app        entry points, budgets, manifests  → proc_*
```

Every function in these modules is a thin wrapper over a syscall, and its docstring states the syscall number, the OSU cost and the commitment level. There is no hidden network access, because there is no network: `osfun.http` does not exist, and a program that wants external data uses an oracle syscall whose result is committed and therefore verifiable.

24.9 Metering Python

The manifest declares a cycle ceiling; the developer needs to know what that buys. The compiler emits a static cost report per function, and the runtime tracks actuals.

Python construct	Cycles	Notes
`a + b` (small ints)	3	Tag check, add, tag
`a + b` (Decimal)	14	128-bit add plus scale check
`d[k]` (dict, str key of 12 chars)	46	Hash 12 bytes, probe, compare
`for x in list` (per item)	7	Bounds check elided by the optimiser
`f"{x} sent"` (2 parts)	120	Format, allocate, copy
Function call (3 args)	22	Frame setup, argument moves
`await` boundary	40	Task switch
`wallet.transfer(...)`	1 417 OSU + 190 cycles	Cost is dominated by the domain, not by Python

The last row is the one that reframes the whole exercise. A Python-level operation costs tens of cycles; a domain-level effect costs thousands of OSU. Programs should therefore be written with the ordinary carelessness one affords to a scripting language, and with real care only at the syscall boundary — which is exactly where the type system, the manifest and the capability constraints already force attention.

CHAPTER 25

A Complete Application, End to End

One hundred and forty lines of Python, two domains, twelve members, and every number accounted for.

25.1 The application

A *rotating savings circle* is one of the oldest financial instruments in the world and one of the least well served by onchain software. Twelve people each contribute a fixed amount every period; each period, one member receives the whole pot; the order is drawn fairly; the circle ends when everyone has received once. It requires recurring authority, deterministic randomness, scheduled execution, cross-domain payment, shared state, and a legible history — which is to say, it requires an operating system.

25.2 The manifest

```
[app]
name        = "circle"
version     = "1.0.0"
publisher   = "os://base/acct/0x9f21…4c07"

[entry]
main        = "circle.main"
on_period   = "circle.on_period"      # invoked by a scheduled event
on_deposit  = "circle.on_deposit"     # invoked when a member pays

[capabilities]
"val.transfer" = { asset = "os://*/asset/usd", max_per_call = "100.00",
                   max_total = "1200.00", window = "30d",
                   recipient = "os://*/prog/circle/pot" }
"xdom.open"    = { max_per_call = "1200.00" }
"sys.rand"     = { level = "final" }             # ℓ₃ — see §19.2
"evt.emit"     = { topics = ["circle.*"] }
"mem.alloc"    = { ns = "/app/circle/", max_bytes = 262144 }

[limits]
cycles = 25_000_000
osu    = 9_000
```

Read the capability block as a sentence and it is the entire security model of the application: **circle may move at most $100 at a time and $1200 a month, in dollars, only into its own pot.** Not "circle may spend your USDC" — the constraint is enforced by ring 0, and no bug in the 140 lines below can widen it (Theorem 10.4).

25.3 The program

```
"""circle — a rotating savings group that settles across domains."""
```

```
from osfun import app, state, wallet, events, rand, time, xdom, ui
from osfun.types import USD, Principal, Timestamp

POT = "os://*/prog/circle/pot"


@state.model(ns="/app/circle/{circle_id}")
class Circle:
    members:      list[Principal]          = state.field(max_len=24)
    contribution: USD                      = USD(0)
    period_secs:  int                      = 0
    started_at:   Timestamp                = Timestamp(0)
    round:        int                      = 0
    order:        list[int]                = state.field(max_len=24)
    paid:         dict[Principal, int]     = state.field(max_entries=24)
    pot:          USD                      = USD(0)
    closed:       bool                     = False


@app.entry
def main(circle_id: str, contribution: USD, period_days: int,
         members: list[Principal]) -> Circle:
    """Create a circle. The caller becomes member zero."""
    if not (2 <= len(members) <= 24):
        raise ValueError("a circle needs between 2 and 24 members")

    c = Circle.create(circle_id)
    c.members      = members
    c.contribution = contribution
    c.period_secs  = period_days * 86_400
    c.started_at   = time.now()                      # sys_time, ℓ₂

    # Draw the payout order from committed randomness. sys_rand refuses to
    # return below ℓ₃, so this cannot be re-rolled by a reorganisation.
    seed = rand.committed(round=0, level="final")    # sys_rand, ℓ₃
    c.order = _shuffle(list(range(len(members))), seed)

    app.schedule(on_period, every=c.period_secs, key=circle_id)
    events.emit("circle.created", {"id": circle_id, "n": len(members)})
    return c


@app.entry
def on_deposit(circle_id: str, who: Principal, amount: USD) -> None:
    """A member pays their contribution for the current round."""
    c = Circle.load(circle_id)
    if c.closed:
        raise ValueError("circle closed")
    if amount != c.contribution:
        raise ValueError(f"expected {c.contribution}, got {amount}")
    if c.paid.get(who, -1) == c.round:
        raise ValueError("already paid this round")

    # The member's own capability authorises this; `circle` never holds
    # authority over member funds, only over its own pot (A1, K7).
    wallet.transfer(to=POT, give=amount, on_behalf_of=who)   # val_transfer
    c.paid[who] = c.round
```

```
    c.pot += amount
    events.emit("circle.deposit", {"id": circle_id, "who": who, "amt": amount})


@app.entry
def on_period(circle_id: str) -> None:
    """End of a period: pay out to the member whose turn it is."""
    c = Circle.load(circle_id)
    if c.closed:
        return

    winner_ix = c.order[c.round]
    winner    = c.members[winner_ix]
    late      = [m for m in c.members if c.paid.get(m, -1) != c.round]

    if late:
        # Not everyone paid. Do not pay out a partial pot; roll the round
        # forward and surface the shortfall. The pot is untouched, so no
        # compensation is required (Invariant 7.11).
        events.emit("circle.short", {"id": circle_id, "late": late,
                                     "round": c.round})
        return

    payout = c.pot
    c.pot = USD(0)

    if wallet.domain_of(winner) == app.domain():
        wallet.transfer(to=winner, give=payout, from_=POT)   # val_transfer
    else:
        # The winner lives on the other domain. One syscall; the escrow,
        # the filler, the attestation and the release are Chapter 11's job.
        xdom.open(give=payout, want=payout.as_asset(),       # xdom_open
                  to=winner, deadline=time.now() + 1800)

    events.emit("circle.payout", {"id": circle_id, "to": winner,
                                  "amount": payout, "round": c.round})

    c.round += 1
    if c.round >= len(c.members):
        c.closed = True
        app.unschedule(key=circle_id)
        events.emit("circle.closed", {"id": circle_id})


def _shuffle(xs: list[int], seed: bytes) -> list[int]:
    """Fisher–Yates with a committed seed — deterministic by construction."""
    out = list(xs)
    for i in range(len(out) - 1, 0, -1):
        j = int.from_bytes(seed[(i * 4) % 32:(i * 4) % 32 + 4], "big") % (i + 1)
        out[i], out[j] = out[j], out[i]
    return out


def view(c: Circle) -> ui.Node:
    turn = c.members[c.order[c.round]] if not c.closed else None
    return ui.column(
        ui.title(f"circle · round {c.round + 1} of {len(c.members)}"),
```

```
        ui.money(c.pot, label="in the pot"),
        ui.who(turn, label="receives this round") if turn else ui.done("closed"),
        ui.grid([ui.member(m, paid=c.paid.get(m, -1) == c.round)
                 for m in c.members], columns=4),
        ui.button("pay", label=f"Pay {c.contribution}", intent="circle.deposit"),
    )
```

The same program in TypeScript

Nothing in the application above is Python-specific. The TypeScript SDK elaborates into the same calculus (§13.1) and emits the same syscalls, so the two programs are interchangeable at the kernel boundary. State models are declared with a schema builder rather than decorators, because that is what reads as ordinary TypeScript:

```
// circle.ts — the application of §25.3, in TypeScript.
import { app, state, wallet, events, rand, time, xdom, t } from "@osfun/sdk";
import type { Principal, USD, At } from "@osfun/sdk";

const POT = "os://*/prog/circle/pot";

const Circle = state.model("/app/circle/{circleId}", {
  members:      t.array(t.principal, { max: 24 }),
  contribution: t.usd,
  periodSecs:   t.u32,
  startedAt:    t.time,
  round:        t.u32,
  order:        t.array(t.u32, { max: 24 }),
  paid:         t.map(t.principal, t.u32, { maxEntries: 24 }),
  pot:          t.usd,
  closed:       t.bool,
});

export const main = app.entry(
  async (circleId: string, contribution: USD, periodDays: number,
         members: Principal[]) => {
    if (members.length < 2 || members.length > 24) {
      throw new RangeError("a circle needs between 2 and 24 members");
    }

    const c = await Circle.create(circleId);
    c.members      = members;
    c.contribution = contribution;
    c.periodSecs   = periodDays * 86_400;
    c.startedAt    = await time.now();              // sys_time

    // rand.committed is typed to return a value stamped `final`; a program
    // that tries to shuffle on a `confirmed` seed does not type-check.
    const seed: At<Uint8Array, { sol: "final" }> =
      await rand.committed({ round: 0, level: "final" });   // sys_rand, ℓ₃
    c.order = shuffle(range(members.length), seed);

    app.schedule(onPeriod, { every: c.periodSecs, key: circleId });
    events.emit("circle.created", { id: circleId, n: members.length });
    await c.save();
    return c;
  },
```

```
);

export const onPeriod = app.entry(async (circleId: string) => {
  const c = await Circle.load(circleId);
  if (c.closed) return;

  const winner = c.members[c.order[c.round]];
  const late   = c.members.filter((m) => c.paid.get(m) !== c.round);

  if (late.length > 0) {
    // Nothing moved, so nothing needs compensating (Invariant 7.11).
    events.emit("circle.short", { id: circleId, late, round: c.round });
    return;
  }

  const payout = c.pot;
  c.pot = USD(0);

  if (wallet.domainOf(winner) === app.domain()) {
    await wallet.transfer({ to: winner, give: payout, from: POT });
  } else {
    await xdom.open({                                          // xdom_open
      give: payout,
      want: payout.asAsset(),
      to: winner,
      deadline: (await time.now()).plus({ seconds: 1800 }),
    });
  }

  events.emit("circle.payout", { id: circleId, to: winner,
                                 amount: payout, round: c.round });

  c.round += 1;
  if (c.round >= c.members.length) {
    c.closed = true;
    app.unschedule(circleId);
    events.emit("circle.closed", { id: circleId });
  }
  await c.save();
});
```

The claim to check is not that the code looks similar — it is that the two implementations are indistinguishable to the kernel. Running both against the same input produces the same syscall log, and therefore the same receipt:

```
#   syscall        python image 6b1f…a904     typescript image e30c…77b1
──────────────────────────────────────────────────────────────────────────
1   sys_time       domain=base                domain=base
2   mem_read       /app/circle/seoul-2026     /app/circle/seoul-2026
3   cap_check      cap=0x7c…, verb=XDOM       cap=0x7c…, verb=XDOM
4   xdom_open      1200 USD → 9xQe…, τ=1800   1200 USD → 9xQe…, τ=1800
5   evt_emit       circle.payout              circle.payout
6   mem_write      round=4, pot=0             round=4, pot=0
──────────────────────────────────────────────────────────────────────────
log_root           91ab…7e55                  91ab…7e55      ← identical
osu                3 637                      3 637
cycles             118 214                    421 907        ← differs
```

Only the cycle count differs, because the engines differ, and cycles are paid for by the executor rather than committed to as a result. The `log_root` — the thing the settlement layer actually verifies — is byte-identical, which is what makes the choice of language a preference rather than a commitment. This is vector 44 of the cross-language equivalence group in §26.5.

25.4 What the compiler produces

```
$ osfun build circle/
parse             138 stmts, 6 functions, 1 state model      ok
determinism       no float, no eval, no ambient io           ok
types             all expressions typed; 0 dynamic sites     ok
effects           pure 4 · syscall 8 · alloc 3               ok
layout            Circle: 168 B static + 2 paged maps        ok
codegen           rv64im, -O2, no float, no atomics          ok
link              image 214 632 B  (runtime 181 K + app 33 K)
hash              H(I) = 0x6b1f…a904   H(M) = 0x2c77…10de

static cost report

┌──────────────┬──────────┬──────────┬──────────────────────────────┐
│ entry        │ cycles   │ osu      │ dominated by                 │
├──────────────┼──────────┼──────────┼──────────────────────────────┤
│ main         │  412 000 │    3 118 │ sys_rand (ℓ₃) 2 900 osu      │
│ on_deposit   │   96 400 │    1 517 │ val_transfer  1 417 osu      │
│ on_period    │  118 200 │ 1 417 or │ val_transfer or xdom_open    │
│              │          │   2 583  │                              │
│ view         │   31 500 │        0 │ pure — no syscalls           │
└──────────────┴──────────┴──────────┴──────────────────────────────┘

manifest ceiling: 25 000 000 cycles, 9 000 osu   → 1.6 % / 34.6 % used
```

The last line is the practical value of static metering: a developer sees, before shipping, that the app's worst path uses a third of its declared budget — and a user, before installing, sees a ceiling that the kernel will enforce no matter what the code does.

25.5 One invocation, traced

`on_period` fires for a circle whose winner is on the other domain. The executor produces a receipt; here is the complete syscall log φ that the receipt commits to.

#	Syscall	Arguments (abridged)	OSU	Level
1	`sys_time`	`domain=base`	12	—

2	`mem_read`	`/app/circle/seoul-2026`	90	ℓ_2
3	`cap_check`	`cap=0x7c…, verb=XDOM`	242	—
4	`xdom_open`	`give=1200 USD, to=os://sol/acct/9xQe…, τ=1800s`	2 583	ℓ_2
5	`evt_emit`	`circle.payout`	100	ℓ_1
6	`mem_write`	`round=4, pot=0`	610	ℓ_2
—	**total**	6 syscalls, 118 214 cycles	**3 637**	—

```
receipt (168 bytes)
  image      6b1f…a904
  input      d4c0…2f18     ← H({"circle_id": "seoul-2026"})
  log_root   91ab…7e55     ← merkle root over the six entries above
  out        0e33…ba01     ← H(state delta + emitted events)
  state_root 5f8d…c712     ← C(ω) at halt, Eq. (23.2)
  cycles     118 214
  halt       0 (normal)
```

25.6 Settlement of the invocation

The receipt is a claim: “running image `6b1f…` on input `d4c0…` yields these effects.” Chapter 27 specifies how the claim becomes state. In the default path it is posted with a bond, and after a challenge window with no successful challenge, the effects are applied. The domain sees one transaction carrying the state delta and the intent, not the 118 214 cycles that produced it — which is the entire economic argument for a userland.

Path	Onchain cost	Latency	When
Direct (all logic onchain)	≈ 340 000 gas	1 block	If `on_period` were a Solidity contract
OSVM optimistic	≈ 68 000 gas	1 block optimistic, 30 min final	Default
OSVM succinct	≈ 250 000 gas + proving	1 block + 8 s proving	When the value or the counterparty demands immediacy

25.7 The failure path

A member does not pay. The program's `late` branch runs, no value moves, and the round rolls forward — a case that needs no compensation because nothing was committed. The interesting failure is the other one: the cross-domain payout is opened, escrowed, and no filler appears before τ.

circle — the unhappy path

```
t+0s     on_period → xdom_open(1200 USD → os://sol/…)       ESCROWED
t+2s     IntentOpened observed by 14 fillers
         · 11 decline: notional above their inventory cap
         ·  3 quote; best quote 1198.94 USD net
t+4s     …no fill submitted (destination congestion, μ_a saturated)
t+1800s  τ expires
t+1802s  anyone calls xdom_refund → escrow returns 1200 USD  REFUNDED
t+1803s  kernel emits xdom.refunded → app's on_event runs
t+1803s  circle keeps round 4 open, notifies members, retries
         next period with a longer τ (policy: τ ← 2τ, capped)
```

Figure 25.1 No filler appears. The escrow refunds automatically and permissionlessly; the app is notified and re-runs the payout next period. No user action, no support ticket, no stuck funds.

25.8 Total cost of a circle

Twelve members, twelve rounds, one payout per round, half the members on each domain.

Operation	Count	OSU each	OSU total	Notes
`main` (creation)	1	3 118	3 118	One-time, includes ℓ_3 randomness
`on_deposit`	144	1 517	218 448	12 members × 12 rounds
`on_period` same-domain	6	1 417	8 502	Direct transfer
`on_period` cross-domain	6	2 583	15 498	Escrow and settle
Scheduling ticks	12	180	2 160	Deferred execution
Total	—	—	**247 726 OSU**	≈ 11.6 M gas equivalent, ≈ 3.0 M CU equivalent

At representative fee levels this is a few dollars for the whole twelve-month circle, spread across 169 operations — and the batching of Chapter 9 reduces it further, because the twelve deposits in a period are conflict-free with respect to each other and fuse into one domain transaction per domain.

CHAPTER 26

Other Languages

The instruction set was chosen so this chapter could be short.

26.1 The recipe

A language runs on os.fun if three things exist: a compiler backend targeting RV64IM, a syscall shim implementing the calling convention of §19.3, and a determinism audit for the language's runtime. The first exists for most serious languages already; the second is under 200 lines; the third is the work.

Language	Determinism work required	Image size	Overhead
Rust	None: `no_std` core is deterministic; forbid `std::time`, `HashMap` random seeds	12–80 KiB	1.0× (baseline)
C / C++	Forbid `libm` transcendentals (link soft-float), fix `qsort` tie-breaking, no `rand()`	8–60 KiB	1.0×
osPy	Chapter 24's table	181 KiB runtime + app	7–15×
TypeScript	Freeze the engine's `Math` to a soft-float implementation; forbid `Date`, `Math.random`, `WeakRef`, `FinalizationRegistry`; fix property enumeration order (already specified by the language)	310 KiB engine + app	9–20×
Go (TinyGo)	Single-threaded scheduler; deterministic map iteration (Go randomises it deliberately — must be replaced); no `time.Now`	90–240 KiB	2–4×
Move / Solidity	Already deterministic; used for in-domain placement rather than OSVM	n/a	n/a

The overhead column is cycles relative to hand-written Rust for the same task, and it is the honest cost of a managed runtime. It matters less than it looks: §24.9 showed that domain effects dominate app cost by two orders of magnitude, so a 15× multiplier on the cheap part is usually invisible.

26.2 Rust

```
#![no_std]
use osfun_sdk::{app, wallet, state, USD, Principal, Result};

#[state::model(ns = "/app/counter/{id}")]
```

```rust
pub struct Counter { pub hits: u64, pub owner: Principal }

#[app::entry]
pub fn bump(id: &str) -> Result<u64> {
    let mut c = Counter::load(id)?;
    c.hits += 1;
    c.save()?;                        // mem_write
    Ok(c.hits)
}

#[app::entry]
pub fn tip(id: &str, to: Principal, amount: USD) -> Result<()> {
    let c = Counter::load(id)?;
    wallet::transfer(to, amount, c.owner)?;   // val_transfer, cap-checked
    Ok(())
}
```

The Rust SDK is the reference: the macros generate exactly the same syscall sequences the osPy compiler emits, and both are checked against the same conformance vectors, so a Rust app and a Python app performing the same operations produce identical syscall logs. That property is what makes the language choice genuinely free rather than nominally free.

26.3 TypeScript and JavaScript

TypeScript is the language the shell itself is written in, so it is the one most os.fun developers already have. It reaches the userland by the same route as everything else — an engine compiled to RV64IM — with one wrinkle that is worth taking seriously before the mechanics.

JavaScript numbers are IEEE-754 doubles, so a JS runtime cannot exclude floating point the way osPy does. This is workable because the ECMAScript specification pins arithmetic exactly — addition, multiplication, division and `Math.sqrt` are correctly rounded and therefore identical everywhere. What is *not* pinned is the transcendental library: `Math.sin`, `Math.exp`, `Math.pow` are "implementation-approximated", and two engines legitimately disagree in the last bits.

The fix is the same as osPy's: link a fixed soft-float implementation of the transcendental functions into the engine image, so the behaviour is part of $\mathrm{H}(I)$ rather than part of the host. The measured cost is a factor of 3–8 on those specific functions, which is acceptable given how rarely application code calls them.

The engine

JavaScript reaches OSVM the way every managed language does: a deterministic interpreter compiled to RV64IM, with the engine's bytes inside the image hash. The reference engine is a QuickJS-derived interpreter of about 310 KiB with the non-deterministic surface removed at build time — `Date`, `Math.random`, `WeakRef`, `FinalizationRegistry`, `Intl`, and the host bindings — and the `osfun` module linked in their place.

Where JavaScript is non-deterministic

Source	Why it diverges	Rule
`Math.sin`, `exp`, `pow`	The specification calls these implementation-approximated; two conforming engines may differ in the last bits	A fixed soft-float implementation is linked into the image; the behaviour is part of $\mathrm{H}(I)$

Ordinary arithmetic	Nothing: `+`, `*`, `/` and `Math.sqrt` are correctly rounded by the specification	Permitted unchanged — the one place JavaScript is stricter than Python
`Date`, `performance.now`	Wall clock	Removed; use `osfun.time`
`Math.random`	Host entropy	Removed; use `osfun.rand`
Property enumeration	Already specified: integer keys ascending, then string keys in insertion order	Permitted, and relied upon
`Map` / `Set` iteration	Insertion-ordered by specification	Permitted
`WeakRef`, `FinalizationRegistry`	Observable garbage collection	Removed
`toLocaleString`, `Intl`	Locale tables differ across builds	Removed; formatting helpers ship in `osfun.fmt`
`Error.stack`	Contains engine-specific frames	Normalised to source positions
Async scheduling	Microtask ordering is specified; timers are not	Microtasks permitted; timers removed

The float row is the interesting one, and it cuts the opposite way from Chapter 24. Python's problem is that its `float` semantics are inherited from a C library nobody pins; JavaScript's numbers are IEEE-754 doubles whose basic operations the language specification pins exactly. So `float` is **admissible** in osJS where it is excluded from osPy — the transcendental functions are the only hole, and linking them closes it.

Finality types without a new compiler

The finality calculus of Chapter 13 needs a checker. For osPy and Rust that checker is ours. For TypeScript it is not needed: the structural type system is expressive enough to encode the lattice's order directly, so the errors of §13.6 appear in the editor with no additional tooling.

```
// @osfun/sdk — the finality lattice, in the TypeScript type system.

export type Level = "seen" | "included" | "confirmed" | "final" | "settled";

/** Everything at least as permanent as L — the ⊑-up-set of Definition 12.2. */
export type AtLeast<L extends Level> =
  L extends "seen"      ? Level :
  L extends "included"  ? Exclude<Level, "seen"> :
  L extends "confirmed" ? "confirmed" | "final" | "settled" :
  L extends "final"     ? "final" | "settled" :
                          "settled";

export type Stamp = { readonly [d in Domain]?: Level };

/** A value carrying a stamp. The brand is erased at runtime. */
export type At<T, S extends Stamp> = T & { readonly __stamp: S };

/** Requirement: every effect states the stamp it needs. */
export type Needs<T, S extends Stamp> =
  T & { readonly __stamp: { [d in keyof S]: AtLeast<NonNullable<S[d]>> } };
```

Because `AtLeast<"final">` is the union `"final" | "settled"`, assignability in TypeScript **is** the $\sqsubseteq$ relation: a value stamped `confirmed` is not assignable where `final` is required, and the compiler says so.

```ts
// The syscall bindings state their requirements in their signatures.
declare function transfer(a: {
  to: Principal;
  give: Needs<USD, { base: "confirmed" }>;
}): Promise<Receipt>;

declare function claim(
  intent: IntentId,
  att: Needs<Attestation, { sol: "final" }>,          // ρ = ℓ₃ on Solana
): Promise<Receipt>;

declare function settle<T, S extends Stamp>(
  v: At<T, Stamp>, want: S,
): Promise<At<T, S>>;                                  // Definition 12.5
```

```
circle.ts:41:20 - error TS2345: Argument of type
  'At<Attestation, { sol: "confirmed" }>' is not assignable to parameter of
  type 'Needs<Attestation, { sol: "final" }>'.
    Type '"confirmed"' is not assignable to type '"final" | "settled"'.

41   await xdom.claim(intent, att);
                              ~~~
  @osfun/sdk/finality.d.ts:88:3
    88   att: Needs<Attestation, { sol: "final" }>,
         ~~~
    the expected stamp is declared here (syscall 0x0504, Appendix A)
```

What TypeScript gives is **checking**, not the full analysis: it verifies stamps where they are written and propagated through generics, but it performs no dataflow analysis, so it cannot infer the meet through arbitrary control flow the way Proposition 13.4 does. Concretely, the `pc` rule (T-IF) is outside its reach — branching on a `confirmed` fact does not automatically contaminate the branch. The osPy and Rust toolchains run the real inference; TypeScript catches the boundary errors, which in practice is where most of them are, and the OSVM image is checked by the kernel either way (§14.6).

Cost

Measure	osPy	osJS	Note
Runtime image	181 KiB	310 KiB	Both inside $\mathrm{H}(I)$
`circle.onPeriod` cycles	118 214	421 907	3.6× — engine, not app
Same invocation, OSU	3 637	3 637	Identical: the domain cost is the domain's
Cost of the difference	—	≈ 0.0003 USD	At the executor's rate of §36.4

The last row is the number that decides the argument. Choosing TypeScript over Python costs three ten-thousandths of a cent per invocation, because the part that differs is the cheap part. Language choice on this system is a matter of taste, and it should be.

26.4 The WASM path

Some programs arrive as WebAssembly rather than as source: existing libraries, other toolchains, sandboxed plugins. OSVM supports them by running a deterministic WASM interpreter *inside* the RV64IM machine.

Property	Behaviour	Cost
Determinism	The interpreter is part of the image hash, so all engine behaviour is pinned — including NaN bit patterns and float rounding modes, the two places raw WASM permits divergence	—
Overhead	Interpreting WASM inside RV64IM	8–25×
Proving	No change: the proof is over RV64IM, which does not know it is running an interpreter	—
Use when	The code exists already, or the language has no RV64IM backend	—

The nesting is not elegant, and it is exactly the right trade: the verification layer stays simple because it only ever proves one instruction set, and portability is paid for in cycles, which are the cheapest resource in the system.

26.5 The conformance suite

Every toolchain targeting os.fun must pass the same 412 vectors, grouped as:

Group	Count	What it pins
Arithmetic and overflow	64	Wrapping, division by zero, shifts
Collections	58	Iteration order, hashing, capacity growth
Strings and Unicode	71	NFC normalisation, case mapping, slicing
Numeric formatting	40	Decimal rendering, rounding modes
Syscall encoding	86	Canonical frames, argument order, errors
Failure modes	49	Cycle exhaustion, budget exhaustion, traps
Cross-language equivalence	44	Same program in two languages produces the same syscall log

CHAPTER 27

Verification and Settlement

One honest verifier is enough — and the protocol is designed so that being that verifier is cheap.

27.1 The claim

An executor publishes a receipt (§23.7). The kernel must decide whether to apply its effects. There are exactly four ways to decide, and a real system uses three of them at different value tiers.

Method	Mechanism	Onchain cost	Latency
Trust	Accept the receipt	0	instant
Replicate	n executors must agree	n signatures ≈ 3 000 gas each	ms
Challenge	Accept optimistically; anyone may dispute and win	≈ 68 000 gas happy path	challenge window
Prove	Verify a succinct proof	≈ 250 000 gas	proving time

Trust is admissible only for effects that cannot lose value (Axiom 2.3) — rendering, ordering hints, cache warming. Everything else uses one of the other three.

27.2 Replay: the cheap check everybody can run

Because execution is a pure function of (image, input, log), verifying a receipt is re-running it. On the reference interpreter that is roughly $118000/25 \times 10^6 \approx 5$ ms for the `circle` invocation of §25.5 — cheap enough that the shell verifies receipts for the user's own apps by default, on the user's own device.

> **PROPOSITION 27.1** · Local verification
> A client that holds the image and the syscall log can verify a receipt in $O(\text{cycles})$ time and $O(1)$ space beyond the machine's memory, with no network access beyond fetching the log. Detection of a false receipt is therefore available to every participant, not only to specialised verifiers.

That proposition is what makes the challenge game credible. An optimistic system is only as safe as the assumption that someone is watching; making the watching cost 5 ms and requiring no special hardware is how the assumption becomes reasonable.

27.3 The bisection game

When a challenger disagrees, they do not re-execute onchain — they bisect.

> **DEFINITION 27.2** · Bisection game
> Let N be the claimed cycle count and let both parties commit to the state commitment $\mathrm{C}(\omega_i)$ at agreed step indices. Each round, the parties narrow the disputed interval by a factor k: the defender publishes $k-1$ intermediate commitments, and the challenger names the first one

they dispute. After $R = \lceil \log_k N \rceil$ rounds the interval is a single instruction, which is executed onchain.

$$R = \lceil \log_k N \rceil, \quad \text{gas}_{\text{total}} = R \cdot (g_{\text{round}} + (k-1) \cdot g_{\text{commit}}) + g_{\text{onestep}} \tag{27.1}$$

N (cycles)	k	Rounds	Gas	Worst-case latency (5 min per move)
10^6	2	20	≈ 340 000	3.3 h
10^6	8	7	≈ 218 000	1.2 h
10^8	2	27	≈ 459 000	4.5 h
10^8	8	9	≈ 280 000	1.5 h
10^8	32	6	≈ 306 000	1.0 h

The kernel uses $k = 8$: it minimises latency at close to minimal gas, and the seven commitments per round fit comfortably in one L2 transaction's calldata. Note that these costs are paid only when a dispute happens, which by construction is approximately never — the losing party forfeits a bond, so disputing a correct receipt is pure loss.

27.4 The one-step proof

The final move executes a single RV64IM instruction onchain against a Merkle-proved memory. This is the only place where the instruction set's simplicity is load-bearing.

```
/// @notice Execute one RV64IM instruction and return the resulting state
///         commitment. This function is the arbiter of every dispute: if it
///         disagrees with the defender's claimed post-state, the defender
///         loses their bond.
/// @dev    Memory accesses are proven against `preRoot` with Merkle paths
///         supplied by the caller, so this function holds no state.
function stepOne(
    MachineState calldata pre,      // registers, pc, cycles, memRoot
    bytes32[] calldata memProof,    // path for the instruction word
    MemAccess[] calldata accesses   // proven loads/stores for this step
) external pure returns (bytes32 postCommitment) {
    // 1. prove the instruction word at pc
    uint32 inst = _proveWord(pre.memRoot, pre.pc, memProof);

    // 2. decode
    uint32 opcode = inst & 0x7f;
    uint256 rd  = (inst >> 7)  & 0x1f;
    uint256 rs1 = (inst >> 15) & 0x1f;
    uint256 rs2 = (inst >> 20) & 0x1f;
    uint32 f3   = (inst >> 12) & 0x7;
    uint32 f7   = inst >> 25;

    uint64[32] memory x = pre.x;
    uint64 pc = pre.pc + 4;
    uint64 cost;
    bytes32 memRoot = pre.memRoot;

    if (opcode == 0x33) {
        (uint64 v, uint64 c) = _opRegReg(f7, f3, x[rs1], x[rs2]);
```

```
        if (rd != 0) x[rd] = v;
        cost = c;
    } else if (opcode == 0x13) {
        (uint64 v, uint64 c) = _opImm(f3, f7, x[rs1], _immI(inst));
        if (rd != 0) x[rd] = v;
        cost = c;
    } else if (opcode == 0x03) {
        uint64 addr = x[rs1] + _immI(inst);
        uint64 v = _proveLoad(memRoot, addr, f3, accesses[0]);
        if (rd != 0) x[rd] = v;
        cost = 2;
    } else if (opcode == 0x23) {
        uint64 addr = x[rs1] + _immS(inst);
        memRoot = _applyStore(memRoot, addr, x[rs2], f3, accesses[0]);
        cost = 2;
    } else if (opcode == 0x63) {
        pc = _branch(f3, x[rs1], x[rs2]) ? pre.pc + _immB(inst) : pre.pc + 4;
        cost = 1;
    } else if (opcode == 0x6f) {
        if (rd != 0) x[rd] = pre.pc + 4;
        pc = pre.pc + _immJ(inst);
        cost = 1;
    } else if (opcode == 0x67) {
        uint64 t = (x[rs1] + _immI(inst)) & ~uint64(1);
        if (rd != 0) x[rd] = pre.pc + 4;
        pc = t;
        cost = 1;
    } else if (opcode == 0x37) {
        if (rd != 0) x[rd] = _immU(inst);
        cost = 1;
    } else if (opcode == 0x17) {
        if (rd != 0) x[rd] = pre.pc + _immU(inst);
        cost = 1;
    } else if (opcode == 0x73) {
        // ecall: the disputed value comes from the committed log, so the
        // arbiter needs only the log's merkle proof, never the kernel.
        (x[10], x[11], cost) = _provedSyscall(pre.logRoot, pre.syscallIndex,
                                              accesses);
    } else {
        revert IllegalInstruction(inst);
    }

    return keccak256(abi.encode(keccak256(abi.encode(x)), pc, memRoot,
                                pre.cycles + cost, pre.logRoot));
}
```

Measured gas for `stepOne` is 61 000–140 000 depending on the instruction and the memory-proof depth, dominated by the twelve `keccak256` calls a 4 KiB-page Merkle path requires. It runs at most once per dispute.

27.5 Succinct proofs

For high-value or latency-sensitive results, the executor produces a STARK [41] proof of the whole execution instead of waiting out a challenge window.

$$T_{\text{prove}} \approx \alpha_p \cdot N \log N, \quad g_{\text{verify}} \approx 250000 \text{ gas } (\text{constant in } N) \tag{27.2}$$

with α_p such that 10^6 cycles prove in roughly 1–3 s on a commodity 16-core machine, and proving parallelises nearly linearly across machines.

DEFINITION 27.3 · Settlement policy

For a receipt whose effects carry notional value v, choose

$$\text{policy}(v) = \begin{cases} \text{optimistic} & v < v^* \\ \text{succinct} & v \geq v^* \end{cases}, \quad v^* = \frac{c_{\text{prove}} + g_{\text{verify}} \cdot \pi_{\text{gas}}}{p_{\text{undetected}}} \tag{27.3}$$

where $p_{\text{undetected}}$ is the probability that a false receipt survives the challenge window unchallenged. With a proving cost of 0.04 USD, verification of 250 000 gas, and $p_{\text{undetected}} = 10^{-5}$, v^* is on the order of 5 000 USD: below that, optimistic settlement is cheaper in expectation; above it, pay for the proof. The threshold is a kernel parameter, and the app may always demand proofs regardless of value.

27.6 Bonds and incentives

Party	Stake and payoff	Bond
Executor	Posts a bond with each receipt; forfeits it if a challenge succeeds; earns the execution fee	$b_e \geq v/2$ or a fixed floor
Challenger	Posts a smaller bond; on success takes a fraction of the executor's bond; on failure forfeits	$b_c = b_e/10$
Watchtower	Runs local replay (§27.2) on receipts it cares about; paid a retainer by apps or users, or self-interested	none

THEOREM 27.4 · One-honest-verifier safety

If at least one party runs local replay on a receipt, is willing to post b_c, and can get a transaction included before the challenge window closes, then a false receipt cannot settle. Formally, the executor's expected payoff from publishing a false receipt is

$$E[\text{cheat}] = p_{\text{undetected}} \cdot v - (1 - p_{\text{undetected}}) \cdot b_e < 0 \quad \text{whenever} \quad b_e > \frac{p_{\text{undetected}} v}{1 - p_{\text{undetected}}}. \tag{27.4}$$

The hypothesis that matters is the third: *can get a transaction included*. Censorship during a challenge window is the attack that breaks every optimistic system, and it is a ring -1 property we cannot fix. What the kernel does instead is bound the exposure: the challenge window is extended automatically if the domain's inclusion rate for challenge transactions drops below a threshold, which converts a censorship attack from a theft into a delay. Chapter 33 gives the detection rule and its false-positive rate.

PART V

System Services

The subsystems a person actually touches: who you are and how you stay signed in, what you own and how it moves, where programs come from, how they talk to each other, and what the machine remembers. Each is specified as a service over the kernel of Part III, with its own state, syscalls and failure modes.

CHAPTER 28

Identity, Sessions and Key Management

Sign once with something you cannot lose, act a thousand times with something you can.

28.1 Credentials

Credential	Properties	Domains	Default
Passkey (WebAuthn [44], P-256)	Non-exportable, biometric-gated, synchronised by the platform, phishing-resistant by origin binding	both	yes
Hardware key	Non-exportable, physical possession, no sync	both	—
Seed-derived	Exportable, portable, familiar, and the easiest to lose or leak	both	—
Threshold / MPC	Split across devices or providers; no single point of compromise; complexity in recovery	both	—

Passkeys are the default root credential because they are the only widely deployed credential that a person cannot accidentally paste into a phishing site. The cost is that they sign with P-256, which neither domain natively verified until recently.

EVM · BASE
The RIP-7212 precompile [45] verifies a P-256 signature for approximately 3 450 gas, which is what makes passkeys viable as a root credential on an OP-Stack chain rather than a curiosity requiring 300 000 gas of Solidity. Where the precompile is unavailable the kernel falls back to a verifier contract and the driver reports the higher cost through `estimate`, so the scheduler prices the difference rather than discovering it.

SVM · SOLANA
The SVM verifies Ed25519 [46] at the transaction level for free — the runtime does it before the program executes — and a secp256r1 precompile brings P-256 into the same regime. The kernel therefore accepts a passkey as a root credential on both domains, with the same binding record Eq. (3.1).

28.2 Establishing a session

os.fun — session establishment

```
1  shell generates an ephemeral keypair (k_s) in the device's secure
   element; the private half never leaves it
2  shell renders the request as one sentence, from the constraint set:
     "circle may move up to $100 at a time, $1200 per 30 days,
      in dollars, only to circle's pot, until 24 Sep 2026"
3  human approves with the passkey → one P-256 signature over
     H( "OS_SESSION_V1" ‖ chain_ids ‖ k_s ‖ verbs ‖ Φ ‖ t₀ ‖ t₁ ‖ nonce )
4  kernel writes the capability record (both domains, or lazily on first
   use in the second domain)
5  every later action: signed by k_s, checked by cap_check, bounded by Φ
```

```
risk bound: L ≤ B · min(T, τ)      Eq. (3.2)
revocation: cap_revoke (one bit) or cap_revoke_all (one counter)
```

Figure 28.1 One human signature produces a session key with a bounded capability. Everything afterwards is signed by the device.

The sentence in step 2 is generated from the constraints, not written by the app, and the signature in step 3 covers the same constraint bytes the kernel will enforce. There is no gap between what the user was shown and what was authorised — which is the mechanical version of A7 and the reason the consent dialog is a kernel surface rather than an app surface.

28.3 Rotation

Keys rotate independently per domain, because a compromise on one domain is not evidence of compromise on the other and forcing joint rotation would make recovery harder, not easier.

```
rotate(domain c, new key k′):
  1. sign with the current root credential:  H("OS_ROTATE_V1" ‖ c ‖ k′ ‖ n)
  2. kernel writes bind_c(α) with k′ and increments the binding nonce
  3. epoch bump on c invalidates capabilities minted under the old key
  4. the other domain is unaffected; its bindings still verify against r
```

28.4 Recovery

The guardian policy of §3.5 is a state machine with a mandatory delay and a veto, so that a hostile threshold is a survivable event rather than a terminal one.

Step	Action	Timing
1	k guardians sign a recovery proposal naming a new root key	any time
2	Kernel records the proposal and emits `recovery.proposed`; the account's own credential is notified on every device	immediate
3	Veto window: the current root credential may cancel with one signature	Δ_G = 72 h default
4	If not vetoed, the new root key takes effect; all capabilities are epoch-bumped	after Δ_G

The asymmetry is intentional: guardians can propose but not surprise; the account holder can cancel instantly. Under Eq. (3.3) the residual risk is the case where the account holder is both compromised and unable to veto, which is exactly the scenario for which no protocol has an answer.

28.5 Threat cases

Case	What the attacker gets	Bound
Stolen unlocked phone	Every session on that device until revoked	$\sum_i B_i \cdot \min(T_i, \tau)$ over live sessions; revocable from any other device in one transaction
Phished session grant	Exactly the capability the user approved	$B \cdot \min(T, \tau)$; the sentence in step 2 is the defence
Malicious app update	Nothing: capabilities are bound to the image hash and void on change	0, by Invariant 3.4
Compromised relayer	Ability to delay or drop	No authority: A6
Compromised root credential	Everything, until guardians recover	Total; this is the case guardians exist for

CHAPTER 29

Value: Balances, Transfers and the Unified Ledger

One balance, several representations, and an accounting identity that must hold in every block.

29.1 The unified balance

DEFINITION 29.1 · Unified balance
For a principal α and an asset class A (an `os://*/asset/…` name),

$$\mathrm{bal}(\alpha, A) = \sum_{c \in C} \sum_{a \in \mathrm{repr}_c(A)} \mathrm{bal}_c(\alpha, a) \cdot w_c(a) \tag{29.1}$$

where $\mathrm{repr}_c(A)$ is the set of tokens on domain c representing A and $w_c(a) \in (0, 1]$ is a haircut reflecting redemption risk for that representation.

The haircut is the honest part. "USDC on Base" and "USDC on Solana" are the same asset class in the sense that matters to a user, and are not the same instrument. The kernel therefore shows one number and can always explain it: the balance view expands into representations, and a representation whose haircut moves is surfaced rather than silently averaged.

View	Definition	Where it is used
Spendable now	Representations usable on the domain of the intended action, at ℓ_2	The number on the home screen
Spendable soon	Adds representations reachable by a cross-domain intent within τ	Quotes and planning
Held	Everything, including escrowed and locked	Statements, accounting

29.2 Asset classes and resolution

```
os://*/asset/usd
  ├── os://base/asset/erc20:0xa0b8…eb48   USDC        w = 1.00
  ├── os://sol/asset/spl:EPjF…Dt1v        USDC        w = 1.00
  ├── os://base/asset/erc20:0xda10…9cb3   DAI         w = 0.999
  └── os://sol/asset/spl:Es9v…wNYB        USDT        w = 0.998
```

Membership in a class is a kernel parameter with a governance process (Chapter 35) and a public criterion: same issuer, same redemption promise, same unit of account. Weights are set from observed redemption behaviour and are bounded below; a representation whose weight would fall below the bound is removed from the class instead, because a class member with a 0.7 haircut is not a dollar and calling it one would be the exact illegibility A7 forbids.

29.3 Payments

Operation	Path	Syscalls
Same-domain transfer	Direct, one transaction	`val_transfer`
Cross-domain transfer	Escrow and fill (Chapter 11)	`xdom_open` → `xdom_fill` → `xdom_claim`
Payment request	A signed request object the payer can execute	`evt_emit` + `val_transfer`
Recurring payment	A capability with `rate(N, W)` plus a scheduled trigger	`cap_grant` once, `val_transfer` per period
Conditional payment	Escrow with a predicate; released on attestation	`val_escrow` → `val_release`

Recurring payments deserve a note because they are the clearest example of the capability model paying for itself. A subscription is not a standing approval to drain an account; it is a capability with `amount ≤ 9.99`, `rate(1, 30 days)`, `recipient ∈ {publisher}` and an expiry. The user can read it in one sentence, the kernel enforces it without the publisher's cooperation, and cancelling is a single `cap_revoke` that no counterparty can refuse.

29.4 Swaps and the price band

A swap's authority is not "may trade" but "may trade within this band," and the band is enforced by ring 0:

$$\text{accept} \quad \Leftrightarrow \quad p_{\text{exec}} \geq p_{\text{ref}}(1-\delta) \quad \text{and} \quad \text{out} \geq \text{out}_{\min} \tag{29.2}$$

where p_{ref} comes from an oracle syscall with a declared freshness and δ is the authorised slippage. Combined with batch-uniform ordering (§5.4), the exposure to adversarial sequencing is bounded by δ rather than by the depth of the pool — the user's worst case is a parameter they approved rather than an outcome the market decides.

29.5 The ledger

The journal is a double-entry ledger. Every kernel effect writes a balanced pair, and Invariant K2 is the trial balance:

$$\sum_{\text{accounts}} \Delta\text{bal}_A + \Delta\text{escrow}_A + \Delta\text{fees}_A = 0 \quad \text{for every asset } A \text{ and every transaction.} \tag{29.3}$$

```
journal entry (48 bytes, packed)
┌──────┬──────┬──────────────────────────────────────────────────┐
│  0.. 8│ seq  │ per-principal monotone sequence (K4)             │
│  8..16│ ts   │ domain height ‖ commitment level                 │
│ 16..20│ kind │ transfer | swap | escrow | release | fee | grant …│
│ 20..36│ ref  │ digest of the causing syscall (Eq. 11.6)         │
│ 36..48│ delta│ signed amount, asset index, counterparty index   │
└──────┴──────┴──────────────────────────────────────────────────┘
```

A statement is a range query over this structure with the totals recomputed client-side; because entries are content-addressed by the syscall digest, a statement can be checked against the chain without trusting the indexer.

CHAPTER 30

The Package Manager and App Registry

Installing an app is granting a capability to a hash.

30.1 Publishing

```
$ osfun publish circle/
build              image 214 632 B, H(I)=6b1f…a904            ok
reproduce          rebuilt in clean container → identical      ok
manifest           7 capabilities, 4 limits, 1 surface         ok
sign               publisher key os://base/acct/0x9f21…4c07    ok
upload             image → content-addressed store (3 mirrors) ok
register           AppRegistry.registerApp(appId, owner, payees)
                   tx 0x4c1e…  gas 118 402                     ok
──────────────────────────────────────────────────────────────────
app id  os://*/app/circle@1.0.0
users install by granting capabilities to 6b1f…a904 ‖ 2c77…10de
```

The reproduce step is not optional. An image whose build cannot be reproduced by a third party is an image whose source nobody can audit, so the registry records the build recipe and the OS marks apps whose reproduction has been independently confirmed.

30.2 Updating

An update is a new image hash and therefore a new grant. The shell renders the diff between the old and new capability sets, which is short and legible precisely because the capability language is small:

```
circle 1.0.0 → 1.1.0
  unchanged   val.transfer  ≤ $100/call, $1200/30d, → circle pot
  unchanged   sys.rand      level = final
  NEW         evt.subscribe topics ["circle.*", "payments.received"]
  REMOVED     mem.alloc     /app/circle/  (now uses the shared object)
──────────────────────────────────────────────────────────────────
[ approve update ]   [ keep 1.0.0 ]   [ uninstall ]
```

Keeping the old version is always an option, because the old image still exists and the old capability is still valid. This is a direct consequence of binding authority to code identity rather than to a mutable address, and it is the property that makes “the app changed under me” impossible.

30.3 Distribution and integrity

Images are content-addressed and served by mirrors. Integrity does not depend on the mirror: the client verifies $\mathrm{H}(I)$ against the capability it holds before executing a single instruction. A hostile mirror can withhold, which is a liveness failure with an obvious remedy (another mirror), and cannot substitute, which is the failure that would matter.

30.4 Revenue

App revenue uses the deployed contract suite: `RevenueRouter` takes the protocol fee, sends it to `Treasury`, and credits the remainder to the app's payees in `RevenueVault` — in one transaction, holding no funds between calls.

$$\text{fee} = \left\lfloor \frac{v \cdot \text{bps}}{10^4} \right\rfloor, \quad \text{net} = v - \text{fee}, \quad \text{payee}_i = \left\lfloor \frac{(\text{net}) \cdot s_i}{\sum_j s_j} \right\rfloor \tag{30.1}$$

with integer-division dust assigned to the first payee so that $\sum_i \text{payee}_i = \text{net}$ exactly — an accounting identity, not a rounding convenience, because K2 must hold to the lamport and the wei.

Parameter	Value	Constraint
Protocol fee	configurable	Hard cap 2 000 bps (20%) in the contract
Payees per app	≤ 50	Bounded to keep distribution gas predictable
Fee changes	Platform owner	Two-step ownership transfer; events on every change
App payee changes	App owner only	Cannot be changed by the platform

30.5 Quarantine, and its limits

If an app turns out to be malicious, the OS can: mark it in the registry, warn on the install path, refuse to render it in the launcher, and tell every holder of its capabilities to revoke. The OS cannot: delete the code, seize the app's funds, or revoke a capability on a user's behalf. That asymmetry is uncomfortable and correct — a system layer able to revoke your grants without you is a system layer able to revoke your grants without you.

CHAPTER 31

IPC, Events and Notifications

Programs do not call each other. They send intents, and the kernel decides.

31.1 The event model

Concept	Definition
Topic	A dotted name: `circle.payout`, `payments.received`, `sys.reorg`
Event	`(topic, principal, payload ≤ 512 B, seq, level)`
Emission	`evt_emit` — costs 100 OSU, written to the journal
Subscription	A capability-gated standing filter over topics and principals
Delivery	At-least-once, ordered per principal by `seq`, idempotent by digest

At-least-once with idempotent handlers is the only honest delivery semantics in a system with reorganisations: a delivered event may be undelivered by a reorg and redelivered afterwards. Handlers are therefore written against the digest, and the runtime deduplicates — the same mechanism as Proposition 16.3, reused a third time.

31.2 Inter-app communication

Apps do not hold references to each other. When app X wants app Y to do something, it emits an intent naming Y's entry point, and the kernel delivers it under a capability that X must already hold or must ask the user for. There is no channel that bypasses the capability check, which means composition between apps has exactly the same security properties as composition between a user and an app.

```
# app X asks app Y to do something, passing a narrowed capability along.
from osfun import caps, ipc

sub = caps.delegate(
    parent=my_transfer_cap,
    subject="os://*/app/splitter",
    verbs=["val.transfer"],
    extra=[caps.amount_max(USD(25)), caps.expires_in("10m")],
)                                              # cap_delegate — Theorem 10.4
ipc.send("os://*/app/splitter", "split", {"total": USD(75), "n": 3}, cap=sub)
```

The delegated capability is strictly narrower than the parent, expires in ten minutes, and shares the parent's cumulative counter, so passing it to another app cannot multiply the authority — the property proved in Chapter 10 and implemented in Chapter 21.

31.3 Scheduled and deferred execution

```
app.schedule(handler, every=Δ, key=k)      → a repeating trigger
app.schedule(handler, at=t, key=k)         → a one-shot trigger
app.unschedule(key=k)                      → cancel
```

Triggers are kernel state, not a server's cron table. A trigger fires by someone submitting a transaction that proves the trigger is due; the caller is paid a small bounty from the app's budget, so firing is a permissionless keeper market rather than an operational dependency. If nobody fires it, the trigger stays due and fires late — degraded liveness, never lost state, which is the pattern every ring-1 dependency in this system follows.

31.4 Notifications

The notifier turns events into human-visible signals: a badge, a push notification, a line in the activity feed. It is ring 1, and it is allowed to be wrong in one direction only — it may fail to notify (annoying), and it may not fabricate a notification that appears to come from the kernel, because the shell renders kernel-attributed notifications only when it can verify the journal entry they reference.

CHAPTER 32

The Filesystem: Journal, Index and Availability

The machine remembers everything. The question is what it costs to ask.

32.1 The journal

Each principal has an append-only journal, stored as a ring of pages with older pages committed to a data-availability layer and replaced by their root.

```
journal state
 onchain:   head page (8 KiB, ~170 entries) + root of the archived chain
 archived:  pages 0..n−1, each committed as H(page) in a hash chain,
            bodies stored in DA / content-addressed storage
 query:     recent → read head page directly
            old    → fetch page from DA, verify against the chain
```

The cost consequence: writing history is $O(1)$ and cheap; reading recent history is free; reading deep history costs one fetch plus one hash verification per page. A statement covering a year is a few dozen fetches, which is a client-side operation rather than a service.

32.2 Indexing

An index is a derived view: "all transfers of USD by this principal in March," "every app this account has authorised," "the global list of open circles." Indexes are built by ring-1 services and are untrusted.

> **DEFINITION 32.1** · Verifiable index
> An index answer is *verifiable* if it carries, for each returned item, a Merkle path to a journal page root committed onchain, and for completeness, a proof that no other item in the queried range matches — obtained by returning the range's boundary items and their adjacency proofs.

Inclusion proofs are easy; completeness proofs are the part that most systems skip, and skipping them is how "the indexer forgot your transaction" becomes indistinguishable from "it never happened." Because journal entries are sequentially numbered per principal, completeness over a range reduces to showing an unbroken sequence, which costs one hash per entry and is checked by the client.

32.3 Queries

```
jrnl:/acct/alice/?kind=transfer&asset=usd&from=2026-03-01&to=2026-04-01
jrnl:/app/circle/?topic=circle.payout&limit=50&order=desc
obj:/acct/alice/apps/*/?fields=header              (what apps store)
```

The query language is deliberately small — filters, ranges, ordering, limits — because every construct must be answerable with a completeness proof. A query language that can express joins can express answers nobody can verify.

32.4 Privacy

Everything above is public. This is the property most often glossed over in onchain system design, and the honest statement is:

- **What is public**: every balance, every transfer, every app you authorised, every event your apps emit, and the timing of all of it.
- **What the OS does**: payloads may be encrypted to the recipient's key, with only the envelope onchain; namespaces may hold ciphertext; the shell supports per-app pseudonymous accounts so that activity is not trivially linked across apps.
- **What that does not achieve**: unlinkability. Timing, amounts and graph structure remain visible, and pseudonymous accounts funded from the same source are linked by that funding.
- **What would achieve it**: private-state constructions with proofs of correctness. These are compatible with this architecture — a private state transition is a userland execution whose receipt reveals only a commitment — and are out of scope for this version, named here so that no reader mistakes encryption of payloads for privacy of activity.

PART VI

Security, Economics, Governance

What can go wrong, who pays for what, and what may change after this document is published. A system layer that holds authority on behalf of other people owes precise answers to all three, including the answers that are unflattering.

CHAPTER 33

Threat Model and Security Analysis

The assumptions are listed first, because a security argument that hides its assumptions is marketing.

33.1 What is being protected

Asset	Loss scenario	Severity
User funds	Moved without an authorising capability	Critical
User authority	A capability wider than what was consented to	Critical
Kernel accounting	K2 violated: value created or destroyed	Critical
Userland correctness	A false receipt settles	High
Availability	Intents cannot be submitted or settled	Medium
Privacy	Activity linkage beyond what is inherent onchain	Medium
Cost	Users overpay through bad routing or fee estimation	Low

33.2 Trust assumptions, enumerated

Every security claim in this document is conditional on these and on nothing else. Where a claim needs more, it says so locally.

1. **Domain consensus.** Each domain behaves within its own security model: transactions that reach ℓ_3 are not reverted, and censorship is not total and permanent. (Ring -1; we cannot repair it.)
2. **Cryptography.** Ed25519, secp256k1, P-256, keccak-256, SHA-256, BLAKE3 and the proof system in use are sound at their claimed security levels.
3. **Kernel code correctness.** The ring-0 programs implement this specification. This is the assumption audits and formal verification attack; Theorem 7.8's hypotheses are its precise statement.
4. **At least one honest verifier** per settlement window, able to get a transaction included (Theorem 27.4).
5. **Reproducible builds.** The image a user authorises is the image the source produces (§30.1).
6. **Oracle honesty within bounds** for price-dependent constraints, with freshness and deviation limits enforced onchain.

Not assumed: honesty of schedulers, planners, relayers, indexers, notifiers, fillers, mirrors or executors. Each of these can be fully Byzantine without loss of funds — that property is A6, and it is what the ring structure of §2.3 exists to enforce.

33.3 The attack catalogue

Attack	Mechanism	Mitigation and residual risk

Root credential theft	Device compromise, malware, coercion	Guardians with timelock and veto (§28.4). Residual: total loss if the holder is compromised and cannot veto
Session key theft	Compromised browser or device	Bound by Eq. (3.2): $B \cdot \min(T, \tau)$; revocation from any other device. Residual: the bound itself
Consent phishing	App requests a wider capability than it needs	The consent sentence is generated from the constraint set by the shell, not by the app (§28.2). Residual: users who approve wide grants
UI spoofing	Malicious app draws a fake kernel dialog	Apps return view trees; the shell owns all chrome and all consent surfaces (§22.5). Residual: OS-level compromise of the client device
Malicious upgrade	App changes behaviour after install	Capability void on code-hash change (Invariant 3.4). Residual: user approves the new version without reading the diff
Capability amplification	Delegating to multiply spend caps	Counters at the root record; Theorem 10.4. Residual: implementation bugs, addressed by invariant tests §21.7
Replay	Re-submitting a signed action	Domain-separated digests Eq. (11.6) plus a consumption bitmap. Residual: none known; digest collision requires breaking the hash
Cross-domain confusion	Replaying a domain-a message on domain b	Chain ids inside the digest preimage. Residual: none
Reorganisation	An acted-upon event is retracted	Commitment levels per syscall; compensations (Theorem 7.10). Residual: non-compensable effects must wait for ℓ_3, costing latency
Filler default	Cross-domain fill never arrives	Escrow refunds at τ (Theorem 11.3). Residual: the user waits
Filler griefing	Fill at the last moment to strand the user	τ includes Δ_{safety}; fillers who fill late risk their own capital. Residual: latency, not loss

Sandwiching	Adversarial ordering around a swap	Price bands as ring-0 constraints Eq. (29.2) and batch-uniform ordering (§5.4). Residual: bounded by the authorised δ
Censorship of challenges	Preventing a fraud proof from landing	Challenge window extends automatically when inclusion rates drop (§27.6). Residual: sustained domain-level censorship — a ring -1 failure
False receipt	Executor claims an execution that did not happen	Local replay is 5 ms (Prop. 27.1); bonds make cheating negative-EV (Theorem 27.4). Residual: no honest verifier *and* censorship together
Prover unsoundness	A bug in the succinct proof system	Optimistic path remains available; proofs are an optimisation, not the root of safety. Residual: proof-system bugs on the succinct path
Compiler backdoor	The toolchain emits code that differs from source	Reproducible builds plus independent rebuilds (§30.1). Residual: a backdoor in a toolchain everyone reproduces identically
Oracle manipulation	Feeding a false reference price	Freshness and deviation bounds enforced in ring 0; bands are relative to the oracle, so a manipulated oracle widens error but does not remove the cap. Residual: correlated manipulation across sources
Data withholding	DA layer refuses to serve a page	Onchain roots plus multiple mirrors; the head page is always onchain. Residual: deep history temporarily unreadable
Hot-object DoS	Saturating a shared account's write budget	Sharding Eq. (17.3), per-process rate limits Eq. (16.4). Residual: cost increases during an attack
Rent exhaustion	Forcing allocation until deposits are drained	Allocation requires a capability with `mem.alloc` limits; deposits are charged to the allocating principal. Residual: an app can exhaust its own budget

Kernel upgrade attack	Malicious change to ring 0	EVM: immutable, no upgrade path. SVM: 72 h timelock exceeding the exit path, then burned at v1.0 (§20.5). Residual: the v0.x window
Supply chain	Compromised dependency in an app image	The image hash covers all dependencies; vendored, reproducible builds. Residual: a compromised dependency that is faithfully reproduced

33.4 What we do not protect against

Stating this plainly is part of the specification.

- **A compromised client device.** If the machine rendering the shell is owned by an adversary, nothing downstream helps. Hardware-backed credentials narrow but do not close this.
- **Total, sustained censorship by a domain.** Both the challenge game and refund paths require inclusion.
- **A user who approves a wide capability.** The system's job is to make the grant legible, not to override the person.
- **Loss of a root credential with no guardians configured.** Self-custody without recovery is a choice the system permits and warns about.
- **Economic failure of an asset.** If a stablecoin depegs, the OS shows the haircut; it does not make the holder whole.
- **Privacy of activity.** See §32.4.

33.5 Formal properties and where they hold

Property	Statement	Depends on
Kernel soundness	Theorem 7.8	Assumptions 1–3
Non-amplification	Theorem 10.4	Assumption 3
Serializability	Theorem 9.3	Assumption 1
Cross-domain safety	Theorem 11.3	Assumptions 1, 2, timeout Eq. (11.1)
Convergence under reorg	Theorem 7.10	Assumption 1, compensability
Termination of userland	Theorem 23.3	Nothing beyond the VM
Bit-reproducibility	Theorem 23.4	Assumption 5
Settlement safety	Theorem 27.4	Assumptions 1, 4
Reorg noninterference	Theorem 13.2	Assumption 1, and that the program is well typed
Compensation discharge	Theorem 13.3	Well-typedness alone

33.6 Verification plan

1. **Invariant tests** (§21.7) run in CI over randomised call sequences.
2. **Differential testing** between the SVM and EVM kernels: the same syscall sequence must produce the same journal, modulo domain-specific fields.
3. **Conformance vectors** (§34) for drivers, toolchains and the VM.
4. **Formal verification** targets, in priority order: the finality calculus — mechanising Theorem 13.2 and checking that the inference algorithm of §13.5 implements the rules of §13.2, which is the smallest artefact with the largest consequence; then `cap_check` and the attenuation order (Theorem 10.4), the escrow state machine (K5 and Theorem 11.3), and the one-step verifier's agreement with the reference interpreter.
5. **External audit** of ring 0 before any mainnet deployment holding user funds, with the scope defined as the three hypotheses of Theorem 7.8.

CHAPTER 34

Economics of the System Layer

Every participant must be better off participating honestly, and the arithmetic should be visible.

34.1 Who pays for what

Cost	Borne by	Recovered through
Domain fees (gas, CU, signature)	Whoever submits	Charged to the user in the asset they hold, via paymaster or fee payer (§8.6)
Cross-domain liquidity	Filler	The fill fee Eq. (11.3)
Userland execution	Executor	Execution fee, per Eq. (8.9)
Proving (when used)	Executor	A premium on the execution fee
Storage deposits	Allocating principal	Refunded on close, less the keeper bounty
Ring-1 operation (scheduler, indexer, notifier)	Operator	Protocol fee share, or self-interest — an app running its own scheduler pays for its own reliability
Protocol development	os.fun	Protocol fee on app revenue (§30.4)

34.2 The fee schedule

The only fee the protocol itself takes is a share of *application revenue* — not of transfers, not of balances, not of gas. The mechanism is the deployed `RevenueRouter`: an app that charges its users routes the payment through the router, which takes `protocolFeeBps` for the treasury and splits the remainder among the app's payees by Eq. (30.1).

Property	Value	Why
Fee basis	App revenue only	Charging on transfers would tax the system layer's core function, which is exactly what we are trying to make free

Hard cap	2 000 bps (20%), enforced in the contract	A cap that governance cannot exceed is a cap; a policy is not
Custody	None — the router holds no funds between calls	A6: no new trust root
App control	Payees and shares set by the app owner alone	The platform cannot redirect an app's revenue

34.3 Unit economics

The question that decides whether a system layer can exist without a subsidy: what does one active user cost, and what do they generate?

Item	Per active user per month	Basis
Domain fees	0.18–0.90 USD	40 intents, mostly batched; Eq. (9.4)
Userland execution	0.02–0.11 USD	≈ 200 M cycles at JIT throughput
Storage deposits	0.00 net	Refundable; carry cost only
Ring-1 infrastructure	0.03–0.08 USD	Amortised across users
Total cost	**0.23–1.09 USD**	—
Revenue at 5% of app spend	varies	Zero for users who never pay an app; the model does not depend on charging them

The structural point is that the OS is cheap because it is thin. It adds one capability check (242 OSU) and one journal write (100 OSU) to operations that already cost thousands, so the system layer's overhead is on the order of 5% of the domain cost — and it removes far more than that through batching and routing.

34.4 Incentive compatibility

Each ring-1 role must have a positive expected payoff for honest behaviour and a negative one for dishonest behaviour.

Role	Honest payoff	Dishonest payoff
Scheduler	Fee share for intents it lands; reputation drives volume	Reordering yields nothing: uniform batching (§5.4); censoring yields nothing: another scheduler picks up the intent
Planner	Better plans win volume	A bad plan is rejected by ring-0 constraints, wasting the planner's own submission cost
Filler	Eq. (11.3) margin over cost	Filling late risks own capital; not filling forfeits the fee
Executor	Execution fee	Negative by Theorem 27.4 whenever $b_e > pv/(1-p)$

Watchtower	Share of forfeited bonds, or self-protection	Not challenging costs nothing but forfeits the reward
Keeper	Bounty from expired leases and due triggers	Not acting delays; someone else collects

The weakest link is the watchtower: challenging is profitable only when fraud occurs, and fraud does not occur precisely because challenging is credible. This is the standard optimistic-system paradox, and the standard answer applies — make verification so cheap (5 ms, §27.2) that participants verify for their own reasons rather than for a bounty, and let the bounty exist for the case where they do not.

34.5 On tokens

os.fun requires no token to function. Every mechanism in this document prices in the domains' native assets and settles in ordinary stablecoins and tokens; the fee model is application revenue share through a contract that is already deployed. Anything a token might later do — governance weight, staking for ring-1 roles, fee discounts — is orthogonal to the specification and is deliberately excluded from it, because a system layer whose correctness depended on the price of its own token would violate A6 by adding exactly the kind of trust root the architecture exists to avoid.

CHAPTER 35

Governance, Versioning and ABI Stability

Most of this system is designed so that governance cannot reach it.

35.1 The immutability map

Component	Change process	Who
Syscall numbers and semantics	Never change; new behaviour is a new number	—
Capability semantics, attenuation order	Immutable	—
EVM kernel contracts	Deploy new, users re-delegate	Users, one at a time
SVM kernel program	Timelocked upgrade until v1.0, then immutable	Multisig, then nobody
Asset class membership and haircuts	Parameter governance, bounded	Governance, within hard bounds
Protocol fee bps	Parameter, hard-capped at 2 000 bps in code	Platform owner
App payees and shares	Per-app	App owner
Driver parameters (κ_c, fee policy, batch interval)	Operational; each operator sets their own	Operators
Ring-1 software	Anyone may fork and run their own	Anyone

35.2 ABI stability rules

1. A syscall number, once assigned, keeps its semantics forever.
2. Arguments may be added only through a new syscall number.
3. Errors may be added; existing codes never change meaning.
4. Deprecation marks a call as discouraged; it continues to work.
5. `sys_version` reports the highest implemented number per family, so clients degrade deterministically rather than by probing.

The rule that matters most is the first, and the reason is Theorem 7.8: a capability authorises a verb, a verb maps to a syscall, and if a syscall's meaning could change then a capability's meaning could change after the user consented. ABI stability is not a developer-experience nicety here; it is a precondition of the security argument.

35.3 Parameter governance

Parameters that governance may set are enumerated, each with hard bounds enforced in code rather than in policy:

Parameter	Bounds	Rationale
`protocolFeeBps`	0 – 2 000	Contract constant `MAX_PROTOCOL_FEE_BPS`
d_{policy} per syscall class	1 – 64 blocks/slots	Below 1 is meaningless; above 64 destroys latency
$\tau_{\min}$ for cross-domain	Eq. (11.1)'s right-hand side	A shorter timeout would break Theorem 11.3
Challenge window	15 min – 24 h	Long enough to include a challenge, short enough to be usable
k (bisection arity)	2 – 32	Eq. (27.1)
Asset haircut $w_c(a)$	0.95 – 1.00	Below 0.95 the representation leaves the class instead

35.4 Upgrade mechanics

```
EVM     deploy new suite → publish addresses + diff → users re-delegate
        old suite keeps working indefinitely; no forced migration
        rollback = users simply do not re-delegate

SVM     propose(program_data_hash) → 72 h timelock → apply
        during the window: hash published, diff reviewable, exit available
        v1.0: set_upgrade_authority(None) — irreversible
```

35.5 Emergency response without a pause button

There is no pause function, so the response to an incident is necessarily different from the industry norm. What exists:

- **Advisory revocation.** The kernel can emit a `sys.advisory` event that every shell renders as a prominent warning with a one-tap `cap_revoke_all`. It cannot revoke on a user's behalf.
- **Registry flagging.** A compromised app is flagged; installs are blocked in the default launcher; existing users are warned.
- **Filler and executor blacklists.** Ring-1 operators may refuse to serve a party; other operators may disagree, which is the intended check.
- **Parameter tightening.** Widening a challenge window or lowering a haircut is within governance bounds and takes effect quickly.

What does not exist is the ability to freeze a user's assets, reverse a transfer, or stop the kernel. In the incident classes where the industry reaches for a pause button — a contract bug draining a pool — this architecture's answer is that the bug must not be in ring 0, which is why ring 0 is 40 KB of code with seven invariants and an audit scope of three hypotheses.

35.6 Standards

os.fun deliberately implements existing standards where they exist rather than inventing parallel ones: ERC-4337 [11] and EIP-7702 [10] for account abstraction, ERC-7683 for cross-chain intents, SPL Token and Token-2022 on the SVM, WebAuthn [44] for credentials, and RISC-V [42] for the userland ISA. Where we extend, the extension is additive and documented as such — the capability

record, the OSU accounting, and the commitment-level annotation are all fields that a standard-conformant counterparty can ignore without breaking.

PART VII

Evaluation and Horizon

What the model predicts, what has actually been built, what is coming, what this work owes to its predecessors, and what remains unsolved. The numbers in this part are analytical and simulated; where something has not been measured on a production system, it says so.

CHAPTER 36

Performance Model and Evaluation

Predictions, with their derivations attached, so that the first production deployment can falsify them.

36.1 Method and honesty about it

Three kinds of number appear in this document, and they are marked consistently:

Kind	Meaning
Protocol constants	Published parameters of Base and Solana: gas schedules, CU limits, slot times, packet sizes. Verifiable by anyone, subject to governance change
Derived	Computed from constants by an equation in this document. Only as good as the equation, which is given so it can be checked
Simulated	Produced by the harness described below against synthetic workloads. Not a measurement of a production system, because no production system of this shape exists yet

The simulation harness replays a synthetic workload — a mixture of transfers, swaps, app calls and cross-domain intents drawn from the distribution of activity observed on both domains — through the scheduler, the conflict-graph batcher and driver cost models, against recorded historical fee data. It does not execute against a live chain, so it cannot capture leader behaviour, mempool dynamics or adversarial reordering; those are named as the largest sources of error.

36.2 Latency

Using the decomposition of Eq. (5.4) with the terms of §5.3:

Intent class	p_{50}	p_{95}	Level	Dominated by
Same-domain transfer, session-signed	620 ms	1.9 s	ℓ_2	Block inclusion
Swap with price band	780 ms	2.4 s	ℓ_2	Quote fetch + inclusion
App call (OSVM, JIT, optimistic)	95 ms	340 ms	ℓ_1	Execution + local render
App call settled onchain	1.1 s	3.1 s	ℓ_2	Inclusion of the state delta
Cross-domain transfer	4.8 s	22 s	ℓ_2	Filler response + destination inclusion
Cross-domain, succinct proof	11 s	38 s	ℓ_3	Proving time
Human-signed action (no session)	+1.4 s	+4 s	—	The human, which is why sessions exist

The first row is the one that matters for the product claim of Chapter 1: a payment feels like a payment. The last row is the counterfactual — the same action without the session subsystem — and the difference between them is the entire user-visible value of Chapter 28.

36.3 Throughput

Per domain, the ceiling is the smaller of the batch capacity and the hot-object service rate:

$$\Lambda_c = \min\left(\underbrace{\frac{C_{\text{block}}}{\overline{\gamma}\Delta_{\text{block}}}}_{\text{compute-bound}} , \underbrace{k\mu_a}_{\text{contention-bound}} \right) \tag{36.1}$$

Domain	Compute-bound	Contention-bound ($k = 1$)	Binding
Base	≈ 190 intents/s	≈ 45 intents/s per hot object	Contention, until sharded
Solana	≈ 640 intents/s	≈ 1 500 writes/s per account	Compute, at realistic k

Both ceilings are per-object rather than global, which is the practical consequence of Eq. (17.3): an application that shards its hot state scales until it hits the domain's global limit, and one that does not saturates at the numbers above regardless of how much block space is free.

36.4 Cost

The comparison that matters is against the status quo: the same user actions performed by a person with a wallet and a set of applications.

Scenario	Baseline	os.fun	Source of the difference
8 transfers, same domain, same token	168 000 gas	104 200 gas	Fusion: $(n-1) \times 21000$ plus warm access (§9.2)
Approve + swap	2 tx, 116 000 gas	1 tx, 71 000 gas	Capability replaces the approval; no sepa-

			rate `approve`
Cross-domain transfer	Bridge + 2 tx + manual swap	1 intent, escrow + fill	Escrow-and-fill instead of mint-and-burn
App with 12 state updates	12 tx	1 receipt + 1 settlement	User-land execution, settled once (§25.6)
Session of 40 actions	40 signatures	1 signature	Session capability (§28.2)

36.5 What breaks first

Scaling limit	Reached at	Remedy
Hot object write rate	≈ 1 500 writes/s (SVM), ≈ 45/s (EVM)	Shard: Eq. (17.3)
Scheduler CPU	≈ 20 000 pending intents per instance	Partition by principal prefix; schedulers are stateless and horizontal
Conflict-graph construction	$O(n^2)$ naive at $n \approx 5000$	Interval-tree index over access sets: $O(n \log n)$
Indexer storage	≈ 1.4 TiB per year at 500 intents/s	Tiered: head onchain, recent local, deep in DA
Address lookup table size	256 addresses per table	Multiple tables per transaction; driver rotates them

Filler capital	Eq. (11.4)	More fillers; the market is permissionless
Proving throughput	≈ 1 M cycles/s per prover	Optimistic default; proofs only above v^*

36.6 Sensitivity

The results above are most sensitive, in order, to: the batching interval Δ (Eq. (9.5)), the fee level π_c (Eq. (8.2)), the commitment level demanded by the application, and the shard count k. Doubling Δ from 0.67 s to 1.34 s reduces per-intent cost by 21% and increases median latency by 0.34 s; halving it does the reverse. Every one of these is a parameter an operator sets, which is deliberate — the OS should let a payments app and a game make different choices.

CHAPTER 37

Conformance and the Reference Implementation

What exists today, stated plainly, and what a conforming implementation must do.

37.1 Status

This document specifies a system that is partially built. The honest inventory, at the version on the cover:

Component	Status	Chapter
Shell, boot sequence, window system, launcher	Built, in production as the public surface	19
Revenue contracts (`AppRegistry`, `RevenueRouter`, `RevenueVault`, `Treasury`)	Built, tested, deployed	27, 31
Domain observation (Base block/gas, Solana slot/epoch)	Built	15
Capability registry and kernel entry points	Specified here; implementation in progress	10, 17, 18
Drivers (EVM 7702/4337, SVM v0/ALT)	Specified here	15
Cross-domain settlement	Specified here	11
OSVM	Specified here; reference interpreter prototyped	20
osPy compiler and runtime	Specified here	21
Verification (bisection, one-step, succinct)	Specified here	24

A specification that overstates its implementation is a bad specification, and a system layer that asks for trust it has not earned is worse. The roadmap in Chapter 38 gives the order in which the remaining rows become the first.

37.2 The conformance suite

Level	Name	Requirements
L0	Client	Encodes and decodes canonical frames (§19.4); verifies journal proofs; renders consent sentences from constraint sets
L1	Driver	The five properties of §18.4, plus cost-honesty over a 1 000-transaction sample
L2	Kernel	All seven invariants of §15.3 under randomised sequences; the syscall table of Appendix A with declared costs within 20%

L3	Machine	OSVM conformance vectors: identical traces to the reference interpreter over 412 programs, including all failure modes
L4	Toolchain	§26.5's 412 vectors, plus reproducible builds verified by two independent rebuilds

37.3 Reference implementation layout

```
osfun/
├── kernel-svm/          Rust, no_std, SBPF target      (Ch. 17)
├── kernel-evm/          Solidity 0.8.24, Foundry       (Ch. 18)
├── osvm/
│   ├── interp/          reference RV64IM interpreter   (Ch. 20)
│   ├── jit/             production executor
│   └── onestep/         Solidity one-step verifier     (Ch. 24)
├── ospy/
│   ├── compiler/        parser → typed IR → LLVM       (Ch. 21)
│   └── runtime/         allocator, bigint, str, tasks
├── drivers/{evm,svm}/   Driver trait impls             (Ch. 15)
├── services/            planner, scheduler, indexer    (Ch. 13, 29)
├── sdk/{rust,python,ts}/
├── shell/               the OS surface                 (Ch. 19)
└── conformance/         vectors and harnesses          (Ch. 34)
```

CHAPTER 38

Roadmap: The Boot Sequence

An operating system boots in stages, and so does this one.

os.fun — boot stages

```
v0.1  POST          the shell boots
      ──────────────────────────────────────────────────────────────
      · window system, launcher, boot sequence, live domain status
      · revenue contracts deployed and verified
      exit: the surface exists and is honest about what it does

v0.2  KERNEL        authority and value turn on
      ──────────────────────────────────────────────────────────────
      · CapabilityRegistry + Delegate7702 on Base
      · osk-svm capability and value instructions on Solana
      · sessions, passkey root credentials, guardians
      · wallet and swap as first-party apps over real syscalls
      exit: Theorem 7.8's three hypotheses hold under external audit

v0.3  USERLAND      programs arrive
      ──────────────────────────────────────────────────────────────
      · OSVM executor + reference interpreter + conformance vectors
      · osPy compiler, runtime, SDK; app registry and installs
      · optimistic settlement with local replay; social + launcher
      exit: a third-party app, built by someone else, runs and settles

v1.0  EVERYTHING    the OS, fully on
      ──────────────────────────────────────────────────────────────
      · cross-domain settlement with attestations and succinct proofs
      · bisection game live with bonded challengers
      · SVM upgrade authority burned; EVM suite already immutable
      exit: ring 0 is immutable and the invariants are formally verified
```

Figure 38.1 Each stage has an exit criterion that can be checked from outside, not a date.

The exit criteria are deliberately external. “The audit found the hypotheses of Theorem 7.8 to hold” is checkable by a stranger; “the kernel is done” is not.

CHAPTER 39

Related Work

Almost everything here is borrowed. The contribution is the assembly.

39.1 Capability operating systems

The authority model of Chapter 10 is the capability tradition applied to a new substrate [4], [47]. KeyKOS [5] and EROS [48] established that a system built on unforgeable, attenuable references can be both fast and secure; seL4 [49] showed that such a kernel can be formally verified in full; Capsicum [50] showed that capabilities can be retrofitted onto a POSIX system; Fuchsia showed that they can be the default in a modern consumer OS. Our departures are forced by the substrate: there is no supervisor mode, so unforgeability comes from cryptography and account ownership rather than from a kernel-managed table; capabilities must be revocable in constant time because storage is priced; and the "process" cannot be preempted, so authority must be checked at entry rather than enforced continuously.

39.2 Exokernels

The ring-0 / ring-1 split of §15.1 is an exokernel argument. Engler and Kaashoek's claim [51] was that the kernel should securely multiplex the hardware and get out of the way, leaving abstractions to unprivileged library operating systems. That is exactly the placement rule of Axiom 2.3: ring 0 multiplexes authority, state and value; scheduling policy, planning, indexing and presentation are unprivileged and replaceable. The onchain setting makes the argument stronger than it was on a single machine, because here the "unprivileged" components are run by mutually distrusting parties, and any design that gave them authority would be unimplementable rather than merely unwise.

39.3 Onchain execution environments

The EVM [6] and SVM [16] are the two substrates this document targets, and their respective design choices — sequential execution with discovered access sets versus parallel execution with declared ones — are what Chapters 4 and 9 formalise. Move's resource types and Cadence's capability-based accounts are adjacent to our object and capability models and offer a language-level version of guarantees we place in the kernel.

39.4 Account abstraction and intents

ERC-4337 [11] supplied the validation pipeline and the paymaster pattern; EIP-7702 [10] supplied the ability for an ordinary account to execute contract code, which is what makes the EVM kernel usable without migrating every user to a contract wallet, and the modular-account standards [52] supplied the shape of a delegate that composes with other modules rather than owning the account outright. ERC-7683 [31] supplied the cross-chain intent shape that Chapter 11 aligns with. Systems in the intent tradition — solver networks, SUAVE-style preference expression, Anoma's intent-centric architecture — share our framing that users should declare outcomes; this document differs in scoping intents as *one syscall family inside an operating system* rather than as the organising abstraction of a new chain.

39.5 Verifiable off-chain execution

The settlement layer of Chapter 27 is the least novel part of this document and deliberately so. Interactive bisection over a deterministic machine descends from TrueBit [53], and is in production in Arbitrum's Nitro [54] (bisecting WAVM) and Optimism's Cannon (bisecting MIPS). Cartesi [55] established that a RISC-V machine is a practical target for onchain dispute resolution. The modern zkVMs — RISC Zero, SP1, Jolt, built on the STARK line [41] — made proving RISC-V [42] execution a commodity, which is the fact that decides §23.1's choice of instruction set.

Our contribution here is placement rather than mechanism: the same machinery that a rollup uses to settle a chain, we use to settle *a process*. The consequence is that userland programs get rollup-grade integrity without anyone having to run a rollup, and without the application inheriting a new chain's liveness assumptions.

System	Machine bisected / proved	Unit of settlement
Arbitrum Nitro	WAVM (WASM-derived)	A chain's state root
Optimism Cannon	MIPS32	A chain's state root
Cartesi	RISC-V (Linux)	An application's state
RISC Zero / SP1 / Jolt	RISC-V (proved, not bisected)	A program's output
os.fun OSVM	RV64IM, bisected or proved	**A process's receipt, inside an OS**

39.6 Information flow, and what we did to it

Chapters 12–14 are an application of security-typed languages to a lattice nobody had drawn. Denning's lattice model [33] established the shape; Biba [32] gave the integrity dual, which is the one finality instantiates; Volpano, Irvine and Smith gave the first soundness proof for a flow type system, whose skeleton our Theorem 13.2 follows [35]; Myers and Liskov's decentralised label model [36] and the Sabelfeld–Myers survey [34] are the modern statement of the discipline. The gradual variant of §13.7 takes its machinery from Siek and Taha's gradual typing [37] and its blame semantics from Findler and Felleisen [38].

What we changed is the lattice and the escape hatch. The lattice is consensus permanence rather than secrecy or trust, indexed by domain, with $\top$ for independence — which makes "exposed to Solana but not to Base" a type rather than a comment. And `declassify`, the unverifiable policy hole in every confidentiality system, becomes `settle`: an operation that does not assert a label but waits for one, fails if the world disagrees, and has a price. That price is what makes Chapter 14 possible, and Chapter 14 is what makes the whole thing pay for itself: confirmation depth stops being folklore and becomes $O(\log v)$.

The risk analysis it rests on is older still. Nakamoto's own double-spend calculation [17], Rosenfeld's hashrate analysis [18], and the Bitcoin backbone results of Garay, Kiayias and Leonardos [39] are where the hazard functions of Definition 14.1 come from; our contribution is not the probability model but the decision built on top of it.

39.7 Deterministic language runtimes

Chapter 24's determinism rules draw on a long line of work: Python's own `PYTHONHASHSEED` and insertion-ordered dictionaries, RustPython and MicroPython as embeddable interpreters, Pyodide's demonstration that Python runs acceptably in a sandboxed VM, and the deterministic-JavaScript work done for replay debugging and record/replay systems. The float exclusion follows the same

reasoning that led Java to specify `strictfp` and that leads every consensus system to ban transcendental functions.

CHAPTER 40

Limitations, Open Problems, and Conclusion

40.1 Limitations

1. **Partial implementation.** Chapter 37 lists what exists. Most of this document is a specification, not a report.
2. **No production measurements.** Every performance number is derived or simulated (§36.1).
3. **Two domains.** The abstraction is designed for many, but only the EVM and SVM instantiations have been worked through in detail.
4. **Cross-domain atomicity is impossible** and what we offer instead is escrow with a bounded loser (Proposition 11.1). Applications that genuinely require atomic multi-domain composition cannot have it here or anywhere.
5. **Userland overhead.** A managed-language app costs 7–20× the cycles of hand written Rust (§26.1). This is invisible against domain costs today and would not be if domain costs fell by two orders of magnitude.
6. **Privacy is not solved** (§32.4). Payload encryption is not activity privacy.
7. **Optimistic settlement has a latency floor** set by the challenge window, and succinct proofs have a cost floor set by proving.
8. **Governance of asset classes** is a human judgement about what counts as "a dollar," with bounded but real discretion.
9. **The consent problem is bounded, not eliminated.** A legible sentence still requires a person to read it.
10. **The finality calculus is proved on paper, not in a proof assistant.** Theorem 13.2 follows a standard skeleton, which is a reason for confidence and not a substitute for mechanisation; §33.6 lists it as a verification target.
11. **The optimiser is only as good as its hazard model.** λ_c is estimated from observed reorganisations, and a domain that changes its behaviour invalidates the calibration — the type system's floor survives, the optimality claim does not.
12. **Gradual boundaries are where the guarantee thins.** A program that calls untyped code pays runtime guards and inherits blame; the static theorem applies to the typed fragment only.
13. **Ring -1 is not ours.** Censorship, reordering and reorganisation are mitigated and never eliminated.

40.2 Open problems

Private userland state A userland execution whose receipt reveals only a commitment is compatible with this architecture, and specifying the capability semantics of private state — who may read, how revocation works when the reader already saw the plaintext — is unsolved here.

Optimal planning Eq. (16.3) is minimised heuristically. Whether a planner can be both optimal and cheap enough to run in 40 ms is open, and competition between planners is currently the only quality guarantee.

Cross-domain composition beyond value Escrow works for assets. There is no equivalent construction for an intent that must atomically mutate *state* on two domains, and it is not obvious that one exists that does not reduce to a shared sequencer.

Formal verification at kernel scale seL4 [49] verified 8 700 lines of C at a cost of roughly 20 person-years. Ring 0 here is smaller, but the property set is different — capability attenuation, escrow exclusivity and the one-step verifier's agreement with the interpreter are the three worth the effort, and the last of these is a machine-checked equivalence between a Solidity function and a Rust one.

MEV under batching Definition 5.4 bounds intra-batch extraction and says nothing about inclusion-level extraction. Shared sequencing and encrypted mempools are the candidate answers, both with their own trust assumptions.

Mechanising the finality calculus Theorem 13.2 is a paper proof of a standard shape. Mechanising it, and machine-checking that the inference algorithm of §13.5 implements the rules of §13.2, is the highest-value formal-methods work in this document — higher than the kernel, because the calculus is small and its consequences are large.

Levels beyond retraction The lattice models permanence. It does not model the other two things a caller might want to know about a fact — who attested it and how expensive it would be to fake — and the natural question is whether one product lattice can carry all three without becoming unusable.

Storage economics over decades Both domains price storage as if the future were short. A lease model (§17.3) is the correct abstraction; the right price for a 30-year lease is not known by anyone.

40.3 Conclusion

The argument of this document has been narrow. Onchain computing has, for several years, possessed everything a computer needs except the layer that turns a processor into one: an owner of resources, a stable interface, a model of who may do what, and a place for ordinary programs to run. The absence of that layer is not felt as an absence. It is felt as friction — three wallets, a bridge, a gas token you forgot, a connect button that fails the first time — and friction is easy to mistake for the nature of the thing rather than for a missing artefact.

We have also tried to show that building it forces one genuinely new thing into view. Onchain programs do not fail mainly because their arithmetic is wrong; they fail because they act on facts that are not yet true, and no language they are written in can express the difference. Once finality is a lattice, that difference is a type, the failure is a compile error, the compensation obligation is a proof obligation, and the confirmation depth that every team hardcodes becomes a quantity you can derive. That is the part of this document we would defend hardest, and it is the part that had to wait for an operating system to have somewhere to live: a type system needs a syscall table with declared requirements, and a syscall table needs a kernel.

We have tried to show that the artefact is specifiable. The kernel is small: seven invariants, forty-one syscalls, three audit hypotheses. The formal content is standard and borrowed on purpose — capability calculus from the 1980s, conflict serializability from the 1970s, bisection games from the last decade, RISC-V from a university project that had no idea what it would be used for. The engineering is not easy but it is ordinary engineering, and every mechanism in this document runs on substrates that exist today, with parameters that are published and checkable.

What remains is to build it, and to be measured against the numbers in Chapter 36 rather than against the enthusiasm in Chapter 1. If the numbers hold, then a person will pay a friend in a token they do not hold, on a chain they have never heard of, in under a second, with one signature that they understood — and they will not know that any of this happened, which is the only real test an operating system ever passes.

PART VIII

Appendices

Normative reference material: the complete syscall table, error codes, wire formats, the osPy grammar, the OSVM instruction set, notation, deployed contract interfaces, and the works this specification builds on.

APPENDIX A

Syscall Reference

This is the normative table. Forty-one calls in eight families; no other operation may change kernel state.

A.1 Reading these entries

`osu` is the cost bound, taken as the maximum over domains (Eq. (8.1)). `level` is the commitment level at which the call's effects may be relied upon (Definition 5.2). `comp` names the compensating call, or `none` when the call is read-only or idempotent. Errors are from Appendix B; every call may additionally return `E_LIMIT`.

```
NR      NAME                                          osu · level · comp
        (signature) -> return
        errors and notes
```

A.2 Family `cap` — authority

```
0x0101  cap_grant                                   1240 · l2 · cap_revoke
        (object: Name, verbs: u32, phi: Constraint[], expiry: Time) -> CapId
        E_CAP, E_BUDGET

0x0102  cap_delegate                                 980 · l2 · cap_revoke
        (parent: CapId, subject: Id, verbs: u32, expiry: Time,
         phi: Constraint[]) -> CapId
        E_CAP — raised when the request is not an attenuation (Def. 10.2)

0x0103  cap_revoke                                          290 · l2 · none
        (cap: CapId) -> ()
        E_CAP

0x0104  cap_revoke_all                                      310 · l2 · none
        () -> Epoch
        O(1) regardless of how many capabilities are outstanding

0x0105  cap_check                                            242 · - · none
        (cap: CapId, call: Call) -> Principal
        E_CAP

0x0106  cap_list                                            180 · l1 · none
        (principal: Id, cursor: u64) -> (Cap[], cursor)
        read-only enumeration
```

A.3 Family `proc` — processes

```
0x0201  proc_spawn                                    2167 · l2 · proc_kill
        (cap: CapId, image: Hash, budget: u32, input: bytes) -> Pid
        E_CAP, E_BUDGET, E_CODE

0x0202  proc_step                                  640 · l2 · proc_compensate
        (pid: Pid, step: u16, proof: MerkleProof) -> StepResult
        E_CAP, E_BUDGET, E_CONFLICT, E_DEADLINE

0x0203  proc_status                                    90 · l1 · none
        (pid: Pid) -> Pcb

0x0204  proc_budget                                   410 · l2 · none
        (pid: Pid, add: u32) -> u32
        E_CAP, E_FUNDS

0x0205  proc_kill                                     380 · l2 · none
        (cap: CapId, pid: Pid) -> ()
        E_CAP

0x0206  proc_wait                                     120 · - · none
        (pid: Pid, level: Level) -> Outcome
        E_LEVEL is returned as "not yet", not as a failure

0x0207  proc_compensate                               890 · l2 · none
        (pid: Pid, from_step: u16) -> ()
        runs compensations in reverse causal order (Def. 7.9)
```

A.4 Family `mem` — objects and storage

```
0x0301  mem_alloc                                variable · l2 · mem_close
        (cap: CapId, ns: Name, len: u32, lease: Time) -> ObjId
        cost 620 + 8 per 32 bytes (EVM); 180 + refundable deposit (SVM)
        E_CAP, E_FUNDS, E_LIMIT

0x0302  mem_read                                       90 · l2 · none
        (obj: ObjId, off: u32, len: u32) -> bytes

0x0303  mem_write                               610 · l2 · mem_write(prior)
        (cap: CapId, obj: ObjId, off: u32, data: bytes) -> ()
        E_CAP, E_CONFLICT, E_LIMIT

0x0304  mem_close                                     340 · l2 · none
        (obj: ObjId) -> Refund
        permissionless after lease expiry; splits the deposit (§17.6)

0x0305  mem_lease                                     520 · l2 · none
        (cap: CapId, obj: ObjId, until: Time) -> ()
        E_CAP, E_FUNDS

0x0306  mem_page                                      210 · l2 · none
        (obj: ObjId, page: u32, proof: MerkleProof) -> bytes
        E_LIMIT when the proof depth exceeds the committed height
```

A.5 Family `val` — value

```
0x0401  val_balance                                          110 · l2 · none
        (principal: Id, asset: Name) -> Balance
        returns the three views of §29.1: spendable now / soon / held

0x0402  val_transfer                                1417 · l2 · reverse (< l3)
        (cap: CapId, to: Name, asset: Name, amount: u128) -> Receipt
        E_CAP, E_FUNDS, E_LEVEL, E_DOMAIN

0x0403  val_swap                                             3900 · l2 · none
        (cap: CapId, give: AssetAmount, want: Name, min_out: u128,
         band: bps) -> Receipt
        E_CAP, E_FUNDS, E_BAND, E_DEADLINE

0x0404  val_quote                                             140 · l1 · none
        (give: AssetAmount, want: Name) -> (out: u128, ttl: Time)
        advisory only; the band passed to val_swap is what binds

0x0405  val_escrow                                 1980 · l2 · val_release(ref)
        (cap: CapId, asset: Name, amount: u128, predicate: Hash,
         tau: Time) -> EscrowId
        E_CAP, E_FUNDS

0x0406  val_release                                          1240 · l2 · none
        (escrow: EscrowId, witness: bytes) -> Receipt
        E_CAP, E_DEADLINE, E_LEVEL — K5: releases or refunds exactly once
```

A.6 Family `xdom` — cross-domain

```
0x0501  xdom_open                                    2583 · l2 · xdom_refund
        (cap: CapId, give: AssetAmount, want: AssetSpec, dst: DomainId,
         tau: Time) -> IntentId
        tau must satisfy Eq. (11.1); E_CAP, E_FUNDS, E_DEADLINE

0x0502  xdom_fill                                            1850 · l2 · none
        (intent: IntentId, funds: Proof) -> FillId
        called by the filler, on the destination domain

0x0503  xdom_attest                                          1610 · l2 · none
        (fill: FillId, evidence: Attestation) -> ()
        evidence is a committee signature set, a light-client proof, or a
        succinct proof, selected by notional value (§11.5)

0x0504  xdom_claim                                           2140 · l3 · none
        (intent: IntentId, att: Attestation) -> Receipt
        releases escrow to the filler; E_LEVEL, E_DEADLINE

0x0505  xdom_refund                                          1180 · l2 · none
        (intent: IntentId) -> Receipt
        permissionless after tau; idempotent (Prop. 16.3)
```

A.7 Family `ns` — naming

```
0x0601  ns_resolve                                          150 · l1 · none
        (path: Name) -> ObjId
        O(1): the id is the hash of the path, Eq. (17.2)

0x0602  ns_bind                                        720 · l2 · ns_unbind
        (cap: CapId, path: Name, obj: ObjId) -> ()
        E_CAP, E_CONFLICT

0x0603  ns_unbind                                           290 · l2 · none
        (cap: CapId, path: Name) -> ()

0x0604  ns_list                                             190 · l1 · none
        (prefix: Name, cursor: u64) -> (Name[], cursor)
        enumeration only; never on the resolution path
```

A.8 Family `evt` — events

```
0x0701  evt_emit                                            100 · l1 · none
        (cap: CapId, topic: str, payload: bytes <= 512) -> Seq
        E_CAP, E_LIMIT

0x0702  evt_subscribe                          560 · l2 · evt_subscribe(off)
        (cap: CapId, filter: Filter, handler: Name) -> SubId
        E_CAP, E_LIMIT

0x0703  evt_poll                                            130 · l1 · none
        (sub: SubId, cursor: u64) -> (Event[], cursor)
        at-least-once delivery; handlers deduplicate by digest

0x0704  evt_ack                                             180 · l1 · none
        (sub: SubId, upto: u64) -> ()
```

A.9 Family `sys` — system

```
0x0801  sys_version                                           8 · - · none
        () -> (kernel: SemVer, abi: u16, families: u16[8], features: u64)

0x0802  sys_time                                             12 · - · none
        (domain: DomainId) -> (height: u64, ts: u64, level: Level)
        the domain clock of 5.1 — never the wall clock (Axiom 5.1)

0x0803  sys_rand                                             90 · l3 · none
        (beacon: Id, round: u64) -> bytes32
        refuses below l3; branching on a lower level is a compile error in
        osPy and E_LEVEL at runtime elsewhere (§19.2)
```

APPENDIX B

Error Codes

Code	Name	Condition	Caller's correct response
`-1`	`E_CAP`	Missing, expired, revoked or insufficient capability	Request a new grant; do not retry
`-2`	`E_BUDGET`	Estimated cost exceeds remaining budget	Raise the budget (`proc_budget`) or reduce the work
`-3`	`E_FUNDS`	Insufficient balance, escrow or deposit	Fund the account; do not retry blindly
`-4`	`E_BAND`	Execution price outside the authorised band	Re-quote and re-authorise, or widen the band explicitly
`-5`	`E_DEADLINE`	Deadline passed before completion	Re-issue with a new deadline; expect the refund path to have run
`-6`	`E_CONFLICT`	Write conflict on an object	Retry with exponential backoff; consider sharding
`-7`	`E_LEVEL`	Required commitment level not yet reached	Wait; the call is not a failure
`-8`	`E_DOMAIN`	Domain rejected the transaction	Driver retries with a fee bump; surfaced after n attempts
`-9`	`E_REORG`	A dependency was reorganised away	Compensations have run; re-plan from current state
`-10`	`E_LIMIT`	Capacity limit: bytes, accounts, CU/gas, steps	Split the work into more batches
`-11`	`E_CYCLES`	OSVM cycle budget exhausted	Raise `limits.cycles` in the manifest, or optimise
`-12`	`E_DETERMINISM`	A non-deterministic construct was reached	Compile-time error in practice; a runtime occurrence is a bug
`-13`	`E_UNLOGGED`	A syscall result was not present in the log φ	The receipt is invalid; the executor must re-run
`-14`	`E_VERSION`	Syscall number not implemented at this ABI level	Negotiate down via `sys_version`
`-15`	`E_CODE`	Program code hash differs from the capability's binding	Re-grant to the new code identity, deliberately

APPENDIX C

Wire Formats

C.1 TLV tags

Tag	Type	Encoding
`0x01`	`u64`	LEB128, minimal form; non-minimal is a decode error
`0x02`	`i64`	Zig-zag LEB128
`0x03`	`bytes`	Length-prefixed, no padding
`0x04`	`Name`	The 32-byte canonical hash of an `os://` name (Eq. (17.2))
`0x05`	`amount`	`(u128 value, u8 decimals, Name asset)`
`0x06`	`list`	Count-prefixed; elements in declared order
`0x07`	`struct`	Fields in declaration order; no field tags
`0x08`	`option`	`0x00` = none, `0x01` + value = some

C.2 Digest preimages

```
  intent      "OS_INTENT_V1"  ‖ chain_id ‖ canonical(Intent)
  session     "OS_SESSION_V1" ‖ chain_ids ‖ k_s ‖ verbs ‖ Φ ‖ t₀ ‖ t₁ ‖ nonce
  capability  "OS_CAP_V1"     ‖ subject ‖ object ‖ verbs ‖ expiry ‖ epoch ‖ nonce ‖
parent
  cross-domain "OS_XDOM_V1"   ‖ chain_id_a ‖ chain_id_b ‖ intent_id ‖ nonce
  rotation    "OS_ROTATE_V1"  ‖ domain ‖ new_key ‖ binding_nonce
  binding     "OS_BIND_V1"    ‖ domain ‖ account ‖ binding_nonce
  receipt     "OS_RECEIPT_V1" ‖ image ‖ input ‖ log_root ‖ out ‖ state_root ‖ cycles
```

Every preimage begins with a domain-separating tag and includes a chain identifier, so no signature is valid in two contexts. This is the mechanical basis for the replay claims of §11.7 and Proposition 16.3.

C.3 Constraint encoding

```
constraint (TLV, ≤ 4 per capability on the hot path)
┌──────┬──────────────────────┬──────────────────────────────────────────┐
│ 0x10 │ amount_max   u128    │ per-invocation cap                       │
│ 0x11 │ total_max    u128    │ cumulative cap, counted at the ROOT      │
│ 0x12 │ rate         (u32, u32)│ count per window (window in seconds)   │
│ 0x13 │ recipients   Name[]  │ allow-list (≤ 8 entries inline,          │
│      │                      │ else a merkle root with proofs)          │
│ 0x14 │ price_band   (Name,u16)│ oracle name, tolerance in bps          │
│ 0x15 │ asset        Name    │ restrict to one asset or class           │
│ 0x16 │ domain       u16 bitset│ restrict to specific domains           │
└──────┴──────────────────────┴──────────────────────────────────────────┘
```

APPENDIX D

osPy Grammar and Determinism Rules

D.1 Grammar

The accepted grammar is a proper subset of Python 3.12's. Productions marked † are accepted syntactically and rejected by the determinism pass with `E_DETERMINISM` and a source location.

```
module       ::= { import | funcdef | classdef | decorated | stmt }
import       ::= "from" dotted "import" names        (* osfun.* and local *)
               | "import" dotted                     (* osfun.* and local *)
decorated    ::= decorator { decorator } (funcdef | classdef)
decorator    ::= "@" dotted [ "(" [arglist] ")" ] NEWLINE

funcdef      ::= [ "async" ] "def" NAME "(" [params] ")" [ "->" type ] ":" suite
params       ::= param { "," param }
param        ::= NAME [ ":" type ] [ "=" expr ]
classdef     ::= "class" NAME [ "(" [ NAME ] ")" ] ":" suite   (* single base *)

suite        ::= NEWLINE INDENT stmt+ DEDENT | simple_stmt
stmt         ::= simple_stmt | compound_stmt
simple_stmt  ::= expr_stmt | assign | return | raise | pass | break
               | continue | assert | global†  | del
compound_stmt::= if | while | for | try | with | match | funcdef | classdef

if           ::= "if" expr ":" suite { "elif" expr ":" suite } [ "else" ":" suite ]
while        ::= "while" expr ":" suite [ "else" ":" suite ]
for          ::= [ "async" ] "for" target "in" expr ":" suite [ "else" ":" suite ]
try          ::= "try" ":" suite ( except+ [ else ] [ finally ] | finally )
match        ::= "match" expr ":" NEWLINE INDENT case+ DEDENT
with         ::= [ "async" ] "with" withitem { "," withitem } ":" suite

expr         ::= ternary | lambda | await | comprehension | primary
await        ::= "await" expr
primary      ::= atom { trailer }
atom         ::= NAME | NUMBER | STRING | FSTRING | tuple | list | dict | set
               | "(" expr ")" | "None" | "True" | "False"
trailer      ::= "(" [arglist] ")" | "[" subscript "]" | "." NAME

type         ::= NAME [ "[" type { "," type } "]" ]
```

Rejected at parse time: `eval`, `exec`, `compile`, `__import__`, `globals`, `locals`, `id`, `open`, `input`, `print` (use `osfun.log`), `import` of any module outside `osfun.*` and the program's own package, `class` with more than one base, metaclass arguments, `float` literals, `complex`, `__del__`, `threading`, `multiprocessing`, `socket`, `subprocess`, `ctypes`.

D.2 Finality annotations

```
stamped_type ::= type [ "@" stamp ]
stamp        ::= level | "{" domain ":" level { "," domain ":" level } "}"
level        ::= "seen" | "included" | "confirmed" | "final" | "settled"
               | "?"                          (* gradual, §13.7 *)

settle_expr  ::= "await" "level" "." "settle" "(" expr
                 { "," domain "=" level } ")"
handler_dec  ::= "@" "on_retract" "(" NAME ")" NEWLINE funcdef
```

Stamps are inferred everywhere they are omitted (Proposition 13.4). A written stamp is checked against the inferred one and rejected if it is stronger: the language will let a program forget finality, never invent it (T-SUB).

D.3 Determinism rules

Rule	Statement
D1	No floating-point type. `Decimal` (128-bit, 6 dp) for money; `Fixed[n]` for other fractional values; soft-float `math` only if explicitly imported, and then linked into the image
D2	`hash()` is a pure function of the value, computed with SipHash-1-3 under a constant seed compiled into the runtime
D3	`set` and `dict` iterate in insertion order; `set` operations preserve left-operand order then right-operand order
D4	Object identity is not observable: `id()` is absent, and `is` is permitted only against `None`, `True`, `False`
D5	Finalisation is scope-based; `__del__` is not supported; context managers run deterministically
D6	Time and randomness come only from `osfun.time` / `osfun.rand`, with a declared commitment level
D7	Recursion depth is fixed at 512; exceeding it raises `RecursionError` at the same depth in every implementation
D8	Strings are UTF-8, normalised to NFC at construction; case mapping uses a frozen table shipped in the runtime
D9	Integers are arbitrary precision and metered per 64-bit limb
D10	`async` tasks are scheduled round-robin over the ready queue in creation order; there is no preemption and no timers other than `osfun.time`
D11	Exception messages contain no addresses, paths or timings
D11b	Every effect is checked against its syscall's finality requirement; a program that acts above its evidence does not compile (Chapter 13)
D12	The compiler is reproducible: identical sources produce identical images on any machine

APPENDIX E

OSVM Instruction Set

RV64I base plus the M extension. `imm` fields follow the RISC-V encodings; `sext` denotes sign extension. Cycles are as defined in Definition 23.2.

E.1 RV64I — integer base

Mnemonic	Opcode	funct3/7	Semantics	Cyc
`LUI`	`0x37`	—	`rd ← sext(imm[31:12] << 12)`	1
`AUIPC`	`0x17`	—	`rd ← pc + sext(imm[31:12] << 12)`	1
`JAL`	`0x6f`	—	`rd ← pc+4; pc ← pc + sext(imm)`	1
`JALR`	`0x67`	`0`	`t ← (rs1+sext(imm)) & ~1; rd ← pc+4; pc ← t`	1
`BEQ/BNE`	`0x63`	`0,1`	Branch if `rs1 == rs2` / `!=`	1
`BLT/BGE`	`0x63`	`4,5`	Signed compare branch	1
`BLTU/BGEU`	`0x63`	`6,7`	Unsigned compare branch	1
`LB/LH/LW/LD`	`0x03`	`0,1,2,3`	Signed load of 1/2/4/8 bytes	2
`LBU/LHU/LWU`	`0x03`	`4,5,6`	Unsigned load	2
`SB/SH/SW/SD`	`0x23`	`0,1,2,3`	Store 1/2/4/8 bytes	2
`ADDI`	`0x13`	`0`	`rd ← rs1 + sext(imm)`	1
`SLTI/SLTIU`	`0x13`	`2,3`	Set less than, signed/unsigned	1
`XORI/ORI/ANDI`	`0x13`	`4,6,7`	Bitwise with immediate	1
`SLLI/SRLI/SRAI`	`0x13`	`1,5`	Shift by `imm[5:0]`	1
`ADD/SUB`	`0x33`	`0` / f7 `0x00,0x20`	`rd ← rs1 ± rs2`, wrapping	1
`SLL/SRL/SRA`	`0x33`	`1,5`	Shift by `rs2[5:0]`	1
`SLT/SLTU`	`0x33`	`2,3`	Set less than	1
`XOR/OR/AND`	`0x33`	`4,6,7`	Bitwise	1
`ADDIW/SLLIW/…`	`0x1b`	—	32-bit ops, result sign-extended	1
`ADDW/SUBW/…`	`0x3b`	—	32-bit register ops	1
`ECALL`	`0x73`	`0`	Trap to the kernel (§19.3)	declared
`EBREAK`	`0x73`	`0`	Halt with `Break`	1
`FENCE`	`0x0f`	`0`	No-op (single hart)	1

E.2 M extension — multiply and divide

Mnemonic	funct3	Semantics (funct7 = `0x01`)	Cyc
`MUL`	`0`	Low 64 bits of the product	4
`MULH`	`1`	High 64 bits, signed × signed	4
`MULHSU`	`2`	High 64 bits, signed × unsigned	4
`MULHU`	`3`	High 64 bits, unsigned × unsigned	4
`DIV`	`4`	Signed quotient; $x/0 = -1$; overflow yields the dividend	8
`DIVU`	`5`	Unsigned quotient; $x/0 = 2^{64} - 1$	8
`REM`	`6`	Signed remainder; $x \bmod 0 = x$	8
`REMU`	`7`	Unsigned remainder; $x \bmod 0 = x$	8
`MULW/DIVW/…`	—	32-bit variants, sign-extended results	4/8

The division-by-zero results are architectural, not chosen by us, which is precisely why they are safe to rely on: they are the same in every conforming RISC-V implementation, including every zkVM.

APPENDIX F

Notation and Glossary

F.1 Symbols

Symbol	Meaning	Defined
M_c	Execution domain (abstract machine)	Def. 7.1
Σ_c	State space of domain c	Def. 7.1
δ_c	Transition function	Def. 7.1
Γ_c	Metering function, native units	Def. 7.1
F_c	Finality / commitment function	Def. 5.2
$\ell_0 \ldots \ell_4$	Commitment levels: seen, included, confirmed, final, settled	Def. 5.2
ε_c	Observation lag of domain c	Eq. (7.5)
λ_c	Reorganisation decay rate	Eq. (5.2)
κ_c	Native-unit to OSU conversion	Eq. (8.1)
π_c	Price of one OSU on domain c	Eq. (8.2)
φ_c	Physical conversion map	Eq. (8.1)
$R(t), W(t)$	Read and write access sets	Def. 7.4
G_T	Conflict graph over pending set T	Def. 9.1
Δ	Batching interval	Eq. (9.4)
μ_a	Service rate of hot object a	§8.3
ρ	Utilisation λ/μ	Eq. (9.6)
k	Shard count / bisection arity	Eq. (17.3), Eq. (27.1)
κ	A capability record	Def. 10.1
$\sqsubseteq$	Attenuation order	Def. 10.2
$[\![\kappa]\!]$	Authority denoted by κ	Def. 10.1
$E(\alpha)$	Revocation epoch of principal α	Def. 10.5
$\mathcal{L}$	The finality lattice, $\prod_c L_c$	Def. 12.2
σ	A stamp: a level per domain	Def. 12.2
$\sqcap, \sqcup$	Stamp meet and join (componentwise)	Def. 12.2
$\sqsubseteq$	Finality order: “at least as permanent as”	Def. 12.2
ρ_s	Finality required by syscall s	T-EFF, §13.2
`pc`	Program-counter stamp: meet of control-flow influences	§13.1
$p_c(d)$	Retraction hazard at depth d	Def. 14.1

d^*, ℓ^*	Optimal depth and level for a value at risk	Eq. (14.2)
τ	Cross-domain timeout	Eq. (11.1)
$f_{\min}$	Minimum viable filler fee	Eq. (11.3)
B_P	Process budget in OSU	Def. 8.2
ω	OSVM machine state	Def. 23.1
$\mathrm{C}(\omega)$	State commitment	Eq. (23.2)
$k_{\max}$	Cycle ceiling	Def. 23.2
N	Cycle count of a disputed execution	Def. 27.2
v^*	Value threshold for succinct proofs	Def. 27.3
π	Placement function	Def. 2.2
α	An OS account / principal	Def. 3.1

F.2 Glossary

Term	Meaning
Attenuation	Producing a strictly narrower capability from a wider one. The only permitted direction of delegation
Capability	An unforgeable, scoped, expiring authority object
Commitment level	How permanent an effect is, on a five-point scale
Compensation	The kernel transition that undoes an effect executed below ℓ_3
Domain	An execution environment with its own consensus: Base, Solana
Driver	The ring-1 component that turns abstract batches into domain transactions
Filler	A capital-bearing party who satisfies a cross-domain intent from inventory
Intent	A signed, capability-bearing request for an outcome
Journal	The append-only per-principal record of everything that happened
Lease	Storage held for a time against a refundable deposit
Object	Any kernel-managed record with the 32-byte header of §17.1
OSU	The kernel's normalised resource unit
OSVM	The deterministic RV64IM process machine of Chapter 23
osPy	The deterministic Python dialect of Chapter 24
Placement	The decision of whether a function runs onchain, verified, or trusted
Principal	Any actor: human, session, program, service, filler, validator
Process	An admitted intent with a PCB, a budget and a plan
Receipt	The 168-byte commitment to a userland execution
Finality calculus	The type system of Chapter 13, in which effects may not exceed the permanence of their evidence

Stamp	A vector of levels, one per domain, attached to every value
Settle	The only operation that raises a stamp: it waits, it costs latency, and it may fail
Ring	Privilege level defined by revocation authority, -1 to 3
Session	An ephemeral key with a bounded capability
Shard	One of k objects standing in for a contended one
Syscall	One of the 41 kernel operations of Appendix A

APPENDIX G

Deployed Contract Interfaces

The revenue contracts referenced in Chapters 30 and 34 are deployed and verified. Their interfaces are reproduced here because the app-revenue path is the one part of the economic model that is already live.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

/// @title IAppRegistry
/// @notice Read interface the RevenueRouter uses to resolve an app's split
///         configuration: the protocol fee, the treasury, and per-app payees.
interface IAppRegistry {
    /// @notice Protocol fee in basis points (out of 10_000), taken before
    ///         splitting. Hard-capped at MAX_PROTOCOL_FEE_BPS = 2_000.
    function protocolFeeBps() external view returns (uint96);

    /// @notice Address that receives the protocol fee.
    function treasury() external view returns (address);

    /// @notice Owner that manages an app's payees. Not the platform.
    function appOwner(bytes32 appId) external view returns (address);

    /// @notice Whether an app id has been registered.
    function appExists(bytes32 appId) external view returns (bool);

    /// @notice The payees, their relative shares, and the total of shares.
    function getPayees(bytes32 appId)
        external
        view
        returns (
            address[] memory payees,
            uint256[] memory shares,
            uint256 totalShares
        );
}

/// @title IRevenueRouter
/// @notice Single entry point for app revenue. Takes the protocol fee, sends
///         it to the treasury, and credits the remainder to the app's payees
///         in the vault — in one transaction, holding no funds between calls.
interface IRevenueRouter {
    event RevenueCollected(
        bytes32 indexed appId,
        address indexed token,
        address indexed from,
        uint256 gross,
        uint256 protocolFee,
        uint256 net
    );
```

```
    /// @notice Pay native-asset revenue for `appId`. Send value with the call.
    function collect(bytes32 appId) external payable;

    /// @notice Pay ERC-20 revenue for `appId`. Approve the router first.
    function collectToken(bytes32 appId, address token, uint256 amount) external;
}

/// @title IRevenueVault
/// @notice Pull-based accounting of amounts owed to payees. Credits are
///         written by the router; withdrawal is initiated by the payee, so a
///         hostile or reverting payee cannot block a distribution.
interface IRevenueVault {
    function credit(address payee, address token, uint256 amount) external;
    function balanceOf(address payee, address token) external view returns (uint256);
    function withdraw(address token) external;
}
```

The split arithmetic is Eq. (30.1): the protocol fee is $\lfloor v \cdot \mathrm{bps} \,/ 10^4 \rfloor$, each payee receives $\left\lfloor (\mathrm{net} \cdot s_i) / \sum_j s_j \right\rfloor$, and integer-division dust is assigned to the first payee so that the sum is exactly `net`. Pull-based withdrawal is the mechanism that keeps a single reverting payee from bricking every other payee's revenue — the standard defence, applied because K2 must hold even when a counterparty is hostile.

APPENDIX H

References

The specification is an assembly of existing results. Entries are cited from the chapters that use them; an entry listed without a citation marker is context rather than a load-bearing dependency.

[1] V. Buterin, "Ethereum: a next-generation smart contract and decentralized application platform." 2014.

[2] "Solana Improvement Documents."

[3] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2014.

[4] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, 1966.

[5] N. Hardy, "KeyKOS architecture," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 4, 1985.

[6] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger."

[7] "EIP-2929: gas cost increases for state access opcodes."

[8] "EIP-3529: reduction in refunds."

[9] "EIP-1559: fee market change for ETH 1.0 chain."

[10] "EIP-7702: set code for EOAs."

[11] "ERC-4337: account abstraction using an alternative mempool."

[12] "EIP-1153: transient storage opcodes."

[13] "EIP-2930: optional access lists."

[14] "EIP-7623: increase calldata cost."

[15] "EIP-4844: shard blob transactions."

[16] A. Yakovenko, "Solana: a new architecture for a high performance blockchain." 2018.

[17] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system." 2008.

[18] M. Rosenfeld, "Analysis of hashrate-based double spending." 2014.

[19] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1993.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.

[21] T. Roughgarden, "Transaction fee mechanism design for the Ethereum blockchain: an economic analysis of EIP-1559." 2020.

[22] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, 1979.

[23] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[24] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, 1967.

[25] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM*, vol. 21, no. 2, 1974.

[26] M. S. Miller, “Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control,” Doctoral dissertation, 2006.

[27] J. Gray, “Notes on data base operating systems,” in *Operating Systems: An Advanced Course*, Springer, 1978.

[28] D. Skeen, “Nonblocking commit protocols,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1981.

[29] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM*, vol. 32, no. 2, 1985.

[30] J. D. C. Little, “A proof for the queuing formula L = \lambda W”, *Operations Research*, vol. 9, no. 3, 1961.

[31] “ERC-7683: cross-chain intents standard.”

[32] K. J. Biba, “Integrity considerations for secure computer systems,” technical report, 1977.

[33] D. E. Denning, “A lattice model of secure information flow,” *Communications of the ACM*, vol. 19, no. 5, 1976.

[34] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, 2003.

[35] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *Journal of Computer Security*, vol. 4, no. 2–3, 1996.

[36] A. C. Myers and B. Liskov, “A decentralized model for information flow control,” in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[37] J. G. Siek and W. Taha, “Gradual typing for functional languages,” in *Scheme and Functional Programming Workshop*, 2006.

[38] R. B. Findler and M. Felleisen, “Contracts for higher-order functions,” in *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

[39] J. Garay, A. Kiayias, and N. Leonardos, “The Bitcoin backbone protocol: analysis and applications,” in *Advances in Cryptology — EUROCRYPT*, 2015.

[40] A. Tarski, “A lattice-theoretical fixpoint theorem and its applications,” *Pacific Journal of Mathematics*, vol. 5, no. 2, 1955.

[41] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Scalable, transparent, and post-quantum secure computational integrity.” 2018.

[42] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA.”

[43] J.-P. Aumasson and D. J. Bernstein, “SipHash: a fast short-input PRF,” in *Progress in Cryptology — INDOCRYPT*, 2012.

[44] W3C, “Web Authentication: an API for accessing public key credentials (WebAuthn), Level 3.”

[45] “RIP-7212: precompile for secp256r1 curve support.”

[46] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, “High-speed high-security signatures,” *Journal of Cryptographic Engineering*, vol. 2, no. 2, 2012.

[47] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, 1975.

[48] J. S. Shapiro, J. M. Smith, and D. J. Farber, “EROS: a fast capability system,” in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[49] G. Klein *et al.*, “seL4: formal verification of an OS kernel,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[50] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, “Capsicum: practical capabilities for UNIX,” in *Proceedings of the 19th USENIX Security Symposium*, 2010.

[51] D. R. Engler, M. F. Kaashoek, and J. O'Toole, “Exokernel: an operating system architecture for application-level resource management,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[52] “ERC-7579: minimal modular smart accounts.”

[53] J. Teutsch and C. Reitwießner, “A scalable verification solution for blockchains.” 2017.

[54] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, “Arbitrum: scalable, private smart contracts,” in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[55] A. Teixeira and D. Nehab, “The Cartesi machine: a RISC-V-based verifiable computation platform.” 2018.