Fuchsia Internals: A Fourteen-Volume Reference

Kernel, Build, Graphics, Drivers, Storage, Power, Networking, FIDL, Software Delivery, Diagnostics, Security, and the Linux Runtime — for Hardware Engineers

Systems Architecture Series

***Abstract*—Fuchsia is Google's capability-based microkernel operating system: a small kernel (Zircon) exports a uniform set of *kernel objects* reached through unforgeable *handles*, while nearly all operating-system policy—drivers, filesystems, networking, and graphics—runs in isolated user-space components. This fourteen-volume reference series documents that system bottom-up, from the block device and filesystem, through the kernel and the build that assembles it, the graphics pipeline and drivers that run on top, the power framework that sequences the whole machine into and out of suspend, the networking stack that carries its packets, the FIDL interface language and wire ABI that every service speaks, the security architecture—the capability model and the verified-boot chain of trust—that authenticates and confines all of it, and the Starnix Linux runtime that reuses the kernel's primitives to host a second operating-system personality. The volumes are unified by a handful of recurring primitives—handles and rights, Virtual Memory Objects (VMOs), FIDL messages over kernel channels, and the GN multi-toolchain build—and by a single hardware mental model in which kernel objects behave like IP blocks, handles like bus-grant tokens, and channels like AXI-Stream links. This cover document provides the series preface, a master diagram showing how the fourteen volumes partition the stack, a catalog of each volume with its sections, a recommended reading order, and a cross-volume concept index for reference-style navigation. It is aimed at electrical, RTL, and FPGA engineers who want a precise structural map before diving into any single volume.**

***Index Terms*—Fuchsia OS, Zircon, microkernel, capability system, handles, VMO, FIDL, channels, GN, Ninja, toolchains, graphics pipeline, reference series, hardware mental model.**

I. About This Series

FUCHSIA is large enough that no single document can do it justice without either drowning a newcomer in detail or skating over the mechanisms an implementer needs. This series therefore splits the system into fourteen companion volumes—the *kernel*, the *build*, the *graphics pipeline*, the *Linux runtime*, the *drivers*, the *storage stack*, the *power framework*, the *networking stack*, the *FIDL* interface language and wire ABI, and the *software-delivery* pipeline—each readable on its own, but written to share one vocabulary and one mental model. The split is deliberate: the kernel defines the primitives, the build is the orthogonal machinery that compiles and assembles everything else, graphics is a complete worked application that exercises nearly every primitive at once, and Starnix shows those same primitives reused to host an entire second OS personality.

The throughline across all three volumes is a small set of ideas. *Almost everything is a kernel object accessed through an unforgeable handle* carrying fine-grained rights. The *VMO* (Virtual Memory Object) is the universal buffer—kernel memory, executable images, and graphics framebuffers are all VMOs. *FIDL messages over Zircon channels* are the bus fabric that wires user-space components together. The *GN multi-toolchain model* is how all of that source—kernel, services, and drivers—is compiled, often many times in different contexts, into a single bootable image.

Because the intended reader is an electrical, RTL, or FPGA engineer, the same hardware analogies recur in every volume and are worth stating once: kernel objects behave like *IP blocks* with typed ports; handles are *bus-grant tokens* the requester cannot forge; a channel is an *AXI-Stream* link (typed, ordered, flow-controlled); a port is an *interrupt aggregator* collecting signals from many sources; the BTI and IOMMU together form a *bus firewall* that confines DMA-capable devices; and the scheduler is an *arbiter* granting CPU time. These mappings are not decoration—they are the fastest route into the material for someone who already thinks in synthesizable blocks and handshakes.

II. The Volumes

The fourteen volumes were numbered in the order they were written (their `No. 1`–`No. 14` designations are preserved below), but that is not the recommended reading order—see Section III. Each volume is a self-contained IEEE-style reference; here we give its exact title, length, abstract, and section list.

A. Volume I (No. 1) — Fuchsia OS: Graphics & Display Pipeline

22 pages. *Abstract:* Fuchsia decomposes its graphics pipeline into cooperating user-space services communicating via FIDL typed channels with zero-copy shared VMOs. This reference covers the complete stack—the Magma GPU driver framework, sysmem buffer negotiation, the display coordinator, the Flatland 2D compositor, the Escher Vulkan renderer, frame scheduling, and the Zircon memory model—targeted at hardware engineers familiar with RTL pipeline design and FPGA GPU implementation. *Sections:*

- Introduction; Architecture Overview
- Magma: GPU Driver Framework; sysmem: Shared Buffer Negotiation
- Display Coordinator; Flatland: Modern 2D Compositor; Scenic GFX (Legacy 3D API)
- Escher: Vulkan Renderer; Vulkan Integration
- Frame Scheduling; Memory Model
- Fuchsia Driver Framework and MSD Hosting; Input Pipeline and Display Interaction
- Coordinate Systems, DPI, and Logical Pixels; HDR, Wide Color Gamut, and Color Science
- Debugging and Observability; Vendor-Specific MSD Notes
- Putting It Together: Pixel Lifecycle; Starnix: Linux Compatibility Graphics; Conclusion

B. Volume II (No. 2) — Fuchsia OS: The Build System & Toolchains

15 pages. *Abstract:* The Fuchsia build is widely regarded as opaque on first contact, principally because of its multi-toolchain

Manuscript compiled 2026. This series cover and all fourteen volumes are compiled from the public Fuchsia OS source tree and documentation at `fuchsia.dev` and `cs.opensource.google/fuchsia`. Built with pdflatex and TikZ.

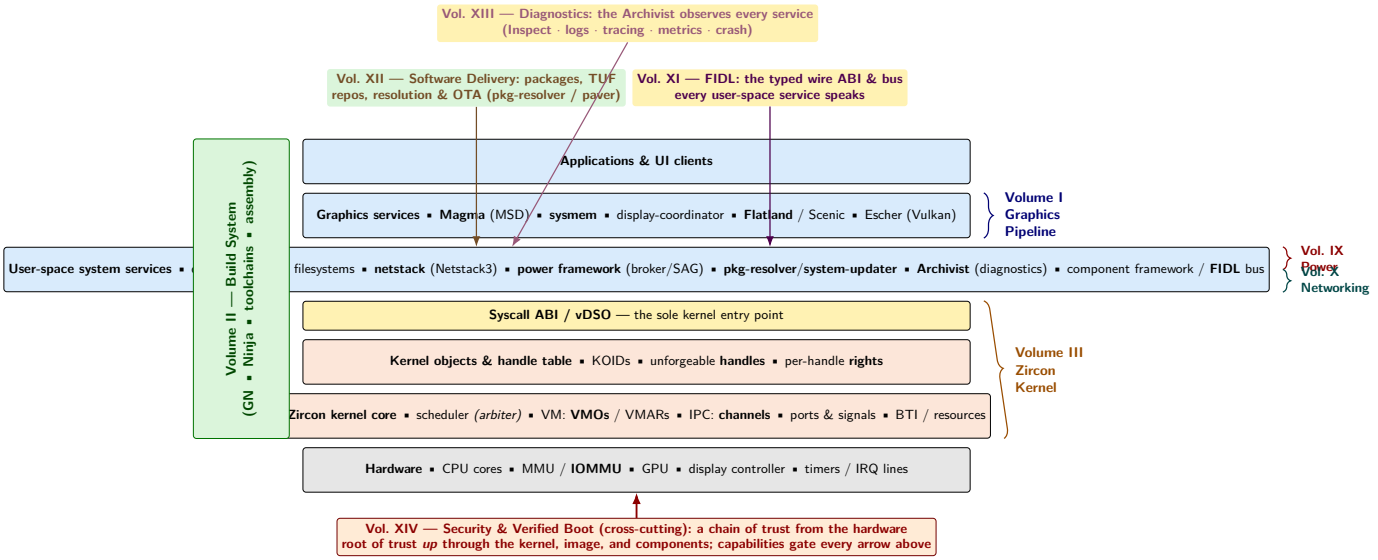

Fig. 1. The Fuchsia stack as horizontal layers, hardware at the bottom and applications at the top, annotated with which volume of this series covers which layers. **Volume III (Zircon Kernel)** owns the kernel core (scheduler, VM, IPC, ports, BTI/resources), the kernel-object and handle table, and the syscall / vDSO ABI that mediates all hardware access. **Volume I (Graphics)** covers the user-space graphics services—Magma, sysmem, display-coordinator, Flatland/Scenic, Escher—that sit on the kernel. **Volume II (Build)** is drawn as an orthogonal band rather than a layer: GN/Ninja, the compiler toolchains, and product assembly are how *every* box above the hardware is compiled and stitched into the running image. **Volume IV (Starnix)** is not drawn as its own band: it is a Linux OS personality that lives in the user-space services layer, running Linux user code on the CPU via restricted mode and reaching sideways into the Volume I graphics services. **Volume IX (Power)** likewise lives in the services layer—the Power Broker and System Activity Governor are ordinary components—but reaches down into the kernel's suspend path and timers and out to every driver's power element. **Volume X (Networking)** is the netstack in that same services layer: Netstack3 sits on the kernel's sockets and VMOs, drives NIC drivers (Volume VII) through the network-device descriptor rings, and serves the socket API to applications and to Starnix. **Volume XI (FIDL)** is not a layer at all but the *contract* on the arrows: the typed interface language and its byte-and-handle wire ABI that every user-space service (the **FIDL bus** in the services layer) speaks, and the bindings generated from it. **Volume XIII (Diagnostics)** is the **Archivist** and the observability plane in that same services layer: it collects Inspect, logs, and traces from every box above the hardware and serves them under selectors, so it observes the whole stack rather than occupying one layer of it. **Volume XIV (Security & Verified Boot)** is drawn as a cross-cutting annotation beneath the stack: it is not one layer but two disciplines that run through all of them—an integrity *chain of trust* rising from the hardware root of trust up through the ZBI, kernel, and verified base packages, and *capabilities* (handles + rights, routed by the component framework) that gate every arrow in the diagram. Warm fill = kernel; cool fill = user space; grey = hardware.

model: a single source target may be compiled many times, once per toolchain context, each producing distinct outputs. This reference dissects the whole stack—`jiri` for source, `fx` as workflow wrapper, GN as the meta-build/elaboration engine, Ninja as the executor, the prebuilt Clang and Rust toolchains, in-build code generation (`fidlc`, `cmc`), and Product Assembly that stitches packages into a bootable image—and makes the *relationships* between those components unambiguous. It is aimed at hardware engineers fluent in RTL elaboration and synthesis: GN is an elaboration pass instantiating a design unit in multiple contexts, Ninja is the build runner, and toolchains are cross-compilation cell libraries. *Sections:*

- Introduction; The Tool Stack & Source Layout
- GN Fundamentals; The Toolchain Model; Toolchain Variants
- The Compiler Toolchains: Clang & Rust; Zircon & Kernel Toolchains
- Code Generation in the Build; Packages, Components & Assembly
- From `fx set` to a Running Image; A Complete Worked Example
- Common Pitfalls & FAQ; Tests in the Build; Drivers in the Build
- The SDK / IDK and Out-of-Tree Builds; The Bazel Migration
- Build Performance & Caching; Debugging the Build
- How the Build Got Here; Glossary; Conclusion

C. Volume III (No. 3) — The Zircon Kernel: Architecture, Subsystems, and the User API

43 pages. *Abstract:* Zircon is the microkernel at the core of Fuchsia. Rather than a monolithic kernel exporting hundreds of ad-hoc syscalls, it exposes a small, uniform set of kernel objects manipulated through unforgeable handles carrying fine-grained rights; almost all OS policy lives in user space. This volume is a detailed tour: the object/handle/rights model and the vDSO-based syscall ABI; the job/process/thread hierarchy and the fair scheduler; the virtual- and physical-memory subsystems (VMOs, VMARs, page queues, the user-space pager); the IPC primitives (channels, sockets, FIFOs, ports, futexes); interrupts, exceptions, and time; the hardware-access objects (resources, MMIO, interrupts, BTIs, the IOMMU); secure boot hand-off; restricted mode and the security model; and diagnostics and multi-architecture support. It targets engineers comfortable with hardware and RTL. *Sections (grouped as in the volume):*

- **Part I — Foundations:** Introduction: The Zircon Microkernel; System Architecture and the Kernel Boundary; Kernel Objects and Handles; The System Call ABI and the vDSO; Capabilities: Rights and the Security Model
- **Part II — Tasks & scheduling:** Tasks: Jobs, Processes, and Threads; Thread Scheduling; Synchronization Primitives
- **Part III — Memory:** Virtual Memory: VMOs and VMARs; Physical Memory and Demand Paging; Kernel Memory, Per-CPU State, and SMP

- **Part IV — IPC, signals, interrupts, time:** Inter-Process Communication: Channels; Streaming and Lightweight IPC (Sockets, FIFOs, Streams, Eventpairs); Signals, Waiting, and Ports; Interrupts and Exceptions; Time, Timers, and Clocks
- **Part V — Platform:** Hardware Access: Resources, MMIO, BTIs, and the IOMMU; Boot and User-Space Bootstrap; The Security Model: Policy, Sandboxing, and Restricted Mode; Diagnostics, Tracing, and Debugging; Multi-Architecture Support and Virtualization; Conclusion
- **Appendices:** A — Syscall Reference; B — Glossary

D. Volume IV (No. 4) — Starnix: Running Unmodified Linux on Zircon

8 pages. *Abstract:* Starnix is Fuchsia's Linux runtime—a Linux-compatible OS personality re-implemented in Rust as an ordinary user-space component. Rather than booting a Linux guest in a virtual machine, it runs unmodified Linux/Android *user* code directly on the CPU via Zircon *restricted mode* and emulates the Linux system-call surface, virtual file system, signals, and IPC in userspace, bridging files to native Fuchsia filesystems through `fuchsia.io` and passing the GPU through to the graphics stack. It builds directly on Volume III. *Sections:*

- Introduction; Architecture Overview; The Restricted-Mode Execution Loop
- Task and Process Model; Namespaces and Credentials; Memory Management
- System-Call Dispatch; The Virtual File System; Bridging to Fuchsia: RemoteFs and `fuchsia.io`
- Signals; Networking; Android Support: Binder, ashmem, and the GPU
- Devices, Containers, and Boot; Diagnostics, `/proc`, and `ptrace`
- Performance and Limitations; Putting It Together: Life of a Linux Process; Conclusion

E. Volume V (No. 5) — Graphics FIDL APIs in Practice

13 pages. *Abstract:* Where Volume I explains the graphics *architecture*, this volume is a practitioner's guide to the wire-level contract: for each graphics FIDL protocol it shows the actual (abridged) FIDL declarations, a timing/sequence diagram of a real usage flow, and the *effects*—what each call causes, and when those effects become visible (latch points, fences, vsync). Covers `fuchsia.ui.composition` (Flatland, Allocator, capture), `fuchsia.sysmem2`, `fuchsia.ui.views` / `fuchsia.element`, `fuchsia.hardware.display`, and `fuchsia.gpu.magma`. *Sections:*

- Introduction; FIDL in Five Minutes, Graphics Edition; Getting a View on Screen
- Allocating Shared Memory: sysmem2; Flatland: Building and Presenting a Scene
- Images: Wiring a Collection into the Scene; A Minimal Client, Annotated
- Direct to the Display Coordinator; GPU Work: `fuchsia.gpu.magma`
- Reading Back: Screen Capture; Ordering Across Channels; Choosing Your Layer
- End to End: One Frame's Life on the Wire; Debugging and Common Failure Modes
- Conclusion; Appendix: Protocol Quick Reference

F. Volume VI (No. 6) — The Component Framework: Realms, Capabilities, and Runners

10 pages. *Abstract:* The connective tissue the other volumes kept pointing at: how Fuchsia assembles isolated processes into a system without ambient authority. Component manager elaborates a tree of *component instances* (realms) from CML manifests; every capability a program can touch is an explicit *route* (use ← offer ← expose) walked lazily at access time and delivered as a namespace entry backed by a channel. Covers CML, capability types and routing semantics, namespaces and the outgoing directory, lifecycle and resolvers, collections and the `Realm` protocol, runners (ELF, Starnix) and environments, storage/config, RealmBuilder, and boot integration—with the RTL analogy made literal: components as module instances, routes as port maps, component manager as the elaborator. *Sections:*

- Introduction; The Topology: Realms, Monikers, Instances; Declaring a Component: CML
- Capabilities and Routing; From Route to Runtime: Namespaces and the Outgoing Directory
- Lifecycle: Resolve, Start, Stop; Dynamic Components: Collections and the Realm Protocol
- Runners and Environments; Storage, Config, and Data; Testing: RealmBuilder
- Boot Integration; Worked Example: Landing a Media Player; Security Posture
- The Complete Picture; Debugging and Common Failure Modes; Conclusion

G. Volume VII (No. 7) — Driver Development in Practice (DFv2)

13 pages. *Abstract:* The capstone for the hardware reader: how to land your own device on Fuchsia. A driver is not kernel code but a sandboxed user-space component in a driver host, reaching hardware only through capabilities the kernel confines. Covers the live node topology maintained by driver manager and driver index; how a driver is a component (`runner: "driver"`) that binds to a node when its `.bind` rules match the node's properties; composite nodes; the `fdf` runtime and its dispatchers; and—the heart—taking MMIO, an interrupt, and a DMA bus-transaction initiator from `fuchsia.hardware.platform.device` and turning them into register access, an ISR, and pinned DMA. Ends with a worked sensor driver, the DriverTestRealm/`ffx driver` workflow, and a debugging guide. Node topology = device tree; bind = the match step; MMIO/IRQ/BTI = the register map, interrupt, and DMA engine you built. *Sections:*

- Introduction; The Big Picture: Driver Manager, Hosts, Nodes; A Driver Is a Component
- Binding: Matching a Driver to a Node; Composite Drivers: One Device, Several Parents
- The Driver Runtime and Dispatchers; Touching Hardware: MMIO, Interrupts, DMA
- Serving a Protocol From Your Driver; Worked Example: A Minimal Sensor Driver
- Writing Drivers in Rust; Testing and Bring-up; From DFv1 to DFv2
- Debugging and Common Failure Modes; Conclusion

H. Volume VIII (No. 8) — Storage: From Block Device to Filesystem (fxfs & blobfs)

11 pages. *Abstract:* On Fuchsia, storage is a stack of user-space servers standing on a thin, DMA-friendly kernel primitive: the block driver is a driver component (Vol. VII), every filesystem is a component (Vol. VI) speaking FIDL, and the only kernel machinery is the handle, the channel, the VMO, and the pager (Vol. III). This volume traces one byte from platter to program: the `fuchsia.hardware.block` FIFO/shared-VMO protocol (a DMA descriptor ring); volume management (FVM vs. multi-volume Fxfs); "filesystems as components" and `fshost`; the `fuchsia.io` node protocol; Fxfs in depth (LSM trees, object stores, the logical journal, the allocator); *why a file read is a page fault* (pager-backed VMOs); and the content-addressed blob store (Blobfs and Fxblob) that Merkle-verifies every page before the CPU runs it. A full-width capstone follows one `open()` of an executable to verified execution. *Sections:*

- Introduction: Storage as a Layered Pipeline; The Block Device Layer
- Volume Management: FVM and Multi-Volume Fxfs; Filesystems as Components
- The `fuchsia.io` Contract; Fxfs: The Principal Filesystem
- Demand Paging: A File Read Is a Page Fault; Content-Addressed Storage: Blobfs and Fxblob
- Capstone: From `open()` to Verified Execution; Integrity, Encryption, and Power-Safety
- Legacy and Alternative Filesystems; Tooling and a Debugging Field Guide
- Conclusion; Appendix: Quick Reference

I. Volume IX (No. 9) — Power Management: The Power Framework, Suspend-to-Idle, and System Power Control

12 pages. *Abstract:* Power management on Fuchsia is a user-space *dependency graph*. The Power Framework models every manageable consumer as a *power element* with discrete *power levels* and every ordering constraint as a *dependency*; a central *Power Broker* (`fuchsia.power.broker`) holds this topology, accepts ref-counted votes called *leases*, and sequences level transitions the way a board PMIC sequences rails—up in dependency order, down in reverse, each step gated by a power-good handshake. Above it the *System Activity Governor* (`fuchsia.power.system`) owns the system's "execution state" and the suspend decision and hands out *wake leases*; below it a suspend HAL (`fuchsia.hardware.power.suspend`) idles the SoC and wake sources (`fuchsia.time.alarms`, hrtimers, interrupts) bring it back. The volume then surveys CPU performance domains and DVFS, thermal throttling and the legacy Power Manager node graph, system power-state control and reboot (including collaborative reboot), batteries and power sources, and DFv2 driver power integration. Written for the EE reader as enables, rails, votes, and sequencing. *Sections:*

- Introduction: Power as a Dependency Graph; The Power Framework at a Glance
- Power Broker: Elements, Levels, Dependencies; Orchestration: Leases, Required Levels, Sequencing
- System Activity Governor; Suspend-to-Idle; Wake Sources and Alarms
- CPU Power and Performance; Thermal Management; The Legacy Power Manager
- System Power-State Control and Reboot (incl. Collaborative Reboot)
- Batteries and Power Sources; Driver Power Integration
- Observability and Debugging; Capstone: Keeping the System Awake to Finish a Job
- Design Rationale and a Translation Table; Conclusion; Appendices: Quick Reference, Glossary

J. Volume X (No. 10) — Networking: The Netstack3 Stack, Sockets, and the Network Device Protocol

13 pages. *Abstract:* A network stack is a packet-processing pipeline, and on Fuchsia it is an ordinary user-space one. Its heart is **Netstack3**, a from-scratch *Rust* rewrite split—like a synthesizable design—into a portable *Core* of protocol state machines (IP, TCP, UDP, ICMP, device, filter) and a Fuchsia *Bindings* layer that supplies the async runtime, the FIDL surface, and lock ordering. Applications reach it through **fdio**, which bridges POSIX `socket()` into `fuchsia.posix.socket` and carries payload on a Zircon socket; the stack drives NIC drivers through the **network-device** protocol (`fuchsia.hardware.network`)—rx/tx FIFOs of descriptor indices over a VMO buffer pool, a NIC's DMA descriptor rings and the sibling of Volume VIII's block protocol. The volume covers interfaces and *netcfg*, SLAAC and DAD, the routing table (a longest-prefix match), the neighbor (ARP/NDP) cache, DNS and DHCP, the netfilter-style packet-filter/NAT engine, HTTP and reachability, and networking in Starnix, before a full-width capstone follows one packet down the tx path and up the rx path. Written for the EE reader as rings, classifiers, CAMs, and refresh cycles. *Sections:*

- Introduction: The Stack as a User-Space Pipeline; The Stack at a Glance
- The Socket API and fdio; Netstack3: Core versus Bindings (+ Netstack2 vs Netstack3)
- Transport and Demultiplexing (the 5-tuple flow classifier; IP forwarding)
- The Network Device Protocol (sessions, FIFOs, the VMO buffer pool, offload)
- Interfaces, Addresses, and netcfg; Routing and Neighbors (LPM, ARP/NDP)
- Name Resolution and DHCP (the DORA handshake); Packet Filtering and NAT
- Higher-Level Services and Starnix; Capstone: The Life of a Packet
- Observability and Debugging; Design Rationale; Conclusion; Appendices: Quick Reference, Translation Table

K. Volume XI (No. 11) — FIDL: The Interface Definition Language, its Wire ABI, and its Bindings

14 pages. *Abstract:* Every arrow in this series' diagrams is a FIDL message crossing a channel; this volume specifies FIDL *itself*—distinct from Volume V, which *uses* graphics FIDL. FIDL is three things at once: a typed *language* of protocols and data; a stable *wire ABI* (a byte-and-handle packet format); and per-language *bindings*. For the hardware reader the wire format is a *bus packet format*—a fixed transactional header whose *ordinal* is an opcode and whose *txid* is an AXI-style transaction tag, 8-byte-aligned little-endian objects (an inline primary object plus out-of-line secondaries), and *envelopes* that make tables and flexible unions a forward/backward-compatible *versioned register map*. Handles never serialize: they are replaced inline by presence markers and shipped out-of-band on the channel's handle array—a sideband descriptor bus. Encode/decode is a *SerDes* object-graph walk; the `fidlc` → JSON IR → `fidlgen`

toolchain is an elaboration/synthesis flow. Covers the type system, protocols/methods, `@available` versioning, the v2 wire format in byte-level detail (with a worked encoding), the C++ (natural vs. wire) and Rust bindings, ABI/API evolution and soft transitions, persistent and driver-transport FIDL, and `fidlcat`. *Sections:*

- Introduction: One Contract, Three Faces; The Language: Types
- Protocols, Methods, and Events; Libraries, Attributes, and Versioning
- Wire Format I: The Message and the Inline Object; Wire Format II: Out-of-Line Objects, Envelopes, and Handles (with a worked encoding)
- Encoding, Decoding, and Messaging; The Toolchain: `fidlc`, the JSON IR, and `fidlgen`
- Bindings I: C++ (Natural vs. Wire); Bindings II: Rust; Common FIDL Patterns
- Evolution and Compatibility (ABI vs. API, soft transitions); FIDL Beyond Channels (persistent, driver transport)
- Capstone: The Life of a FIDL Call; Observability and Debugging (`fidlcat`); How FIDL Got Here
- Conclusion; Quick Reference (wire cheat-sheet); Glossary

L. Volume XII (No. 12) — Software Delivery & OTA: Packages, Repositories, Resolution, and the System Update

13 pages. *Abstract:* A running Fuchsia system is a set of *packages*: content-addressed, Merkle-rooted bundles whose every byte traces back to a signature. This volume follows software from a signed repository to verified execution. It opens the package—a `meta.far` (a *FAR* archive) naming blobs by Merkle root, addressed by a `fuchsia-pkg://` URL—and the base/-cache/universe package sets; specifies the repository as a *TUF* signed role hierarchy (root/targets/snapshot/timestamp) giving tamper-resistance and rollback protection; and traces *resolution* (`pkg-resolver` + `pkg-cache` turning a URL into Merkle-verified blobs and a `fuchsia.io` directory). The second half is the OTA: the *update package*, the `system-updater` driven through `fuchsia.update.installer`, checking via `omaha-client`, and the `fuchsia.paver` writing images into the *inactive* A/B/R boot slot—the FPGA golden/application fallback-boot analogy—then anti-rollback (the `epoch.json` backstop), health-checked commit/rollback, channels, and the end-to-end trust chain from a TUF signature to a per-page Merkle check. Builds directly on Volume VIII. *Sections:*

- Introduction: From a Signed Repository to Verified Execution; The Package: a Content-Addressed Unit (meta.far/FAR, blobs, the package URL)
- Package Sets and the System Image; Repositories and TUF (the signed role hierarchy, rollback protection, rewrite rules)
- Resolving a Package (`pkg-resolver`/`pkg-cache`, the `NeededBlobs` handshake); The Blob Store, Delivery Blobs, and GC
- The Update Package; The OTA Flow (full-width life of an OTA); The Paver and A/B/R Boot Slots (the boot-slot state machine)
- Anti-Rollback: the Epoch Backstop and the Commit; Channels, Omaha, and Eager Updates; Verified Execution, End to End (the trust chain)
- Tooling and a Debugging Field Guide; Capstone: From a Signed Build to a Verified Instruction; Design Rationale; How Software Delivery Got Here
- Conclusion; Appendix: Quick Reference, Glossary, Translation Table

M. Volume XIII (No. 13) — Diagnostics & Observability: Inspect, Structured Logs, Tracing, Metrics, and Crash Reporting

14 pages. *Abstract:* You cannot debug a system you cannot see. This volume is the discipline of *observability*: the instrumentation Fuchsia builds into a running system and the machinery that reads it back out, across five pillars. **Inspect** exports a live tree of structured state into a *VMO* in the Inspect binary block format, whose generation-count lock-free read is a *seqlock* guarding a double-buffered status register. **Structured logs** are typed, timestamped records streamed over a `fuchsia.logger.LogSink` socket—an event FIFO. **Tracing** gives each provider a per-provider VMO *ring buffer* (an ARM-CoreSight trace buffer) of binary Fuchsia-Trace-Format events that `trace-manager` drains and Perfetto visualizes. **Metrics** (Cobalt) report privacy-preserving aggregates—performance counters without per-event detail. And **crash reporting** freezes a faulted process into a minidump and bundles it, with logs and Inspect, into a *snapshot*: a post-mortem scan-out. Binding the first four is the **Archivist**, serving them through `fuchsia.diagnostics.ArchiveAccessor` under *selectors*—a logic-analyzer with trigger/filter expressions—into filtered pipelines. Builds on Volume III (VMOs, exceptions) and Volume VI (components, the outgoing directory). *Sections:*

- Introduction: Reading a Running System (the five pillars + the Archivist hub)
- Inspect I: the Model (nodes, typed properties, histograms, lazy nodes); Inspect II: the VMO Binary Format and the Lock-Free Read (the seqlock, memory-layout centerpiece)
- The Archivist and the Diagnostics Pipeline (`ArchiveAccessor`, pipelines); Structured Logging (`LogSink`, the record encoding, severities)
- Selectors and Querying (`iquery`/`ffx inspect`); Instrumenting a Component (one service, three lenses)
- Tracing I: the System (`trace-manager` + providers); Tracing II: the Ring Buffer and the Fuchsia Trace Format (records, interning, Perfetto)
- Metrics: Cobalt; Persistence and Sampling (bridging Inspect→metrics and across reboot)
- Crash Reporting (full-width life of a crash report: exception→minidump→snapshot); Detect, Triage, and Snapshots
- Diagnostics Across the Series; A Debugging Workflow; Tooling and a Field Guide; Common Failure Modes
- Capstone: One Incident, Four Lenses (full-width); Design Rationale and History; Privacy and the Diagnostics Attack Surface
- Conclusion; Appendix: Quick Reference, Translation Table, Glossary

N. Volume XIV (No. 14) — Security & Verified Boot: Capabilities, Sandboxing, the Chain of Trust, and Verified Execution

13 pages. *Abstract:* The final volume, and a *synthesis*: security on Fuchsia is not a subsystem but the shape of the system, and it resolves into two orthogonal disciplines the earlier volumes kept deferring here. *Access control at runtime* is the object-capability model with *no ambient authority*—a component acts only through an unforgeable *handle* narrowed by *rights* (`ZX_RIGHT_*`), so its *namespace* (built from routed

capabilities) *is* its sandbox and the static capability graph *is* the security policy (Vols. III, VI). *Integrity from silicon to instruction* is an unbroken cryptographic chain: a hardware *root of trust* verifies the bootloader, which verifies a signed *vbmeta* (AVB) covering the *ZBI*; a monotonic *rollback index* bars downgrades; and past boot the pinned `system_image` Merkle root chains through base-package roots to per-blob Merkle trees that `blobfs`/Fxblob verifies *per page* at execution (Vols. VIII, XII). It then surveys the cryptographic services (the CPRNG, `fuchsia.kms`, the `fuchsia.tee` TrustZone secure world), data-at-rest encryption (`zxcrypt` and Fxfs), the exploit mitigations (ASLR, SafeStack, ShadowCallStack, W^X, Rust), build types, and `scrutiny`, the tool that verifies the assembled policy before it ships. For the EE reader: capabilities are per-master bus-grant tokens under SMMU/MPU bits, and verified boot is the silicon secure-boot chain checking a signed image against an eFuse key.
Sections:

- Introduction: Security as Architecture (the two orthogonal disciplines; the RTL framing; how the volume unifies the series)
- The Capability Model (ambient vs. object-capability authority; rights; rights reduction and delegation; the confused deputy, defused)
- Sandboxing: the Namespace *Is* the Sandbox; Capability Routing *Is* the Security Policy; The Trusted Computing Base (+ job policy)
- The Root of Trust and the Boot Chain (the secure-boot chain, silicon analogy); vbmeta, the ZBI, and the Rollback Index
- Verified Execution: Merkle to the Page (full-width, the unbroken chain ROM→fetch)
- Cryptographic Services (CPRNG, `fuchsia.kms`, the TEE, secure storage); Data-at-Rest Encryption (`zxcrypt`, Fxfs)
- Exploit Mitigations; Verifying the System: `scrutiny`; The Production Boundary: Build Types
- Security Across the Series; Capstone: The Unbroken Chain, End to End (full-width); Design Rationale and Threat Model; Common Failure Modes
- Conclusion (closing the fourteen-volume arc); Appendix: Quick Reference, Translation Table, Glossary

III. How to Read This Series

The volumes are numbered by creation order, but build on one another in a different order. For a *newcomer*, the productive path is:

1) **Volume III (Zircon Kernel), Parts I–II first.** The object/handle/ rights model and the memory subsystem are the foundation every other volume assumes. Read at least through scheduling before moving on; the rest of Volume III can be consulted as needed.
2) **Volume VI (Components) right after.** The kernel gives you handles; the component framework decides *who gets which handles*. Realms, CML, and capability routing are assumed context for every user-space discussion in the other volumes.
3) **Volume II (Build) next.** Once you know what a component and a VMO are, the build makes sense: it is how those components are compiled—often in several toolchains—and assembled into the bootable image.
4) **Volume I (Graphics),** as a worked application of the whole stack: it puts handles, VMOs, channels, sysmem, and the scheduler to work in a single real pipeline, and is most rewarding once those primitives are familiar.
5) **Volume V (Graphics FIDL) with or right after Volume I:** it is the wire-level companion to Volume I's architecture—read it when you are about to *write code* against Flatland, sysmem, the display coordinator, or magma.
6) **Volume VII (Drivers) when you have hardware to land.** It is the capstone for the device builder: it assumes Volume III (the kernel hardware-access objects) and Volume VI (components), and shows how to bind a driver to a node and drive real registers, interrupts, and DMA.
7) **Volume VIII (Storage) alongside Volume VII.** It stands on the same Volume III primitives (VMOs and, above all, the pager) and the Volume VI component model, and closes the loop with Volume II by showing how the packages the build produces are stored, verified, and demand-paged into execution. Read it once the kernel memory model and components are familiar.
8) **Volume IX (Power) after Volume VII.** It is a pure user-space dependency-graph story built on the Volume VI component model and the Volume III kernel (timers, the scheduler idle path); its power elements are mostly owned by the DFv2 drivers of Volume VII. Read it once components and drivers are familiar; the EE reader will find its PMIC / rail-sequencing analogy the fastest way in.
9) **Volume X (Networking) alongside Volumes VII and VIII.** It is a pure user-space pipeline standing on the Volume III kernel primitives (sockets, FIFOs, eventpairs, VMOs) and the Volume VI component model; its bottom edge is a NIC driver (Volume VII), and its network-device descriptor ring is the direct sibling of Volume VIII's block protocol, so it reads best next to both. The EE reader will find its descriptor-ring, flow-classifier, and longest-prefix-match analogies the fastest way in.
10) **Volume XI (FIDL) whenever the wire matters**—and it can be read early. It stands on the Volume III channel and the Volume VI routing model and specifies the language, the byte-level wire ABI, and the bindings that *every* other volume's services speak. Read it right after Volumes III and VI for the general contract, or alongside Volume V (which applies FIDL to graphics) when you are about to write code against a specific protocol; the EE reader will find its packet-format, ordinal/txid, envelope, and SerDes analogies the fastest way in.
11) **Volume XII (Software Delivery) alongside Volume VIII.** It stands on Volume VIII's blob store and Merkle machinery (the blobs it fetches and verifies), the Volume VI resolver that turns `fuchsia-pkg://` URLs into components, and the Volume II assembly that produces the packages it distributes. Read it once storage and components are familiar; the EE reader will find its content-addressed-CAM, secure-boot-PKI, redundant-firmware-bank (A/B/R), and anti-rollback-eFuse analogies the fastest way in.
12) **Volume XIII (Diagnostics) whenever you need to observe a running system**—and it can be read early. It stands on the Volume III VMO and exception machinery and the Volume VI component/routing model, and shows how every subsystem in the other volumes exposes itself: Inspect (a seqlock-read status VMO), structured logs, tracing (a CoreSight-style ring buffer), Cobalt metrics, and crash snapshots, all collected by the Archivist and queried by *selectors*. The EE reader will find its status-register, event-FIFO, trace-buffer, performance-counter, and scan-out analogies

the fastest way in.

13) **Volume XIV (Security & Verified Boot) as the capstone synthesis, and it can be read late.** It does not introduce a new subsystem; it reads the security latent in the ones already covered and ties two long diagonals through the series: *access control*—the Volume III handles/rights and the Volume VI capability routing that together make the namespace a sandbox and the topology a policy—and *integrity*—the Volume VIII Merkle blob store and the Volume XII ZBI/vbmeta chain that together verify every instruction from an eFuse key to a page fault. Read it once components, storage, and software delivery are familiar; the EE reader will find its bus-grant-token/SMMU-permission, secure-boot-chain, memory-integrity-tree, anti-rollback-eFuse, and TrustZone-secure-world analogies the fastest way in.
14) **Volume IV (Starnix) last,** for how the same primitives host a second OS personality. It leans hardest on restricted mode and the memory model of Volume III and reaches into the graphics pass-through of Volume I, so it reads best once both are familiar.

Reference-seekers need not read linearly: use the cross-volume concept index (Table I) to jump straight to the treatment of a specific primitive, and the shared vocabulary (Section IV) to settle a term.

TABLE I: Cross-volume concept index. Each recurring primitive and where it is treated across the fourteen volumes (Vol. I = Graphics, Vol. II = Build, Vol. III = Kernel, Vol. IV = Starnix, Vol. V = Graphics FIDL, Vol. VI = Components, Vol. VII = Drivers, Vol. VIII = Storage, Vol. IX = Power, Vol. X = Networking, Vol. XI = FIDL, Vol. XII = Software Delivery, Vol. XIII = Diagnostics, Vol. XIV = Security). Use this as a navigation aid.

Concept	**Where treated (volume + section pointer)**
VMO (Virtual Memory Object)	Vol. III §Virtual Memory: VMOs and VMARs defines the object, its copy-on-write children, and pager semantics; Vol. I §sysmem and §Memory Model use VMOs as the universal zero-copy framebuffer / image substrate.
Handles & rights	Vol. III §Kernel Objects and Handles and §Capabilities: Rights define the model; Vol. II §Packages, Components & Assembly routes them as component capabilities; Vol. I passes handles across every graphics service boundary.
FIDL & channels	**Vol. XI** is the dedicated treatment: the language, the v2 wire ABI (header/ordinal/txid, envelopes, out-of-band handles), and the C++/Rust bindings. Vol. III §Inter-Process Communication: Channels is the kernel transport; Vol. II §Code Generation in the Build (`fidlc`) compiles the bindings; Vol. I §Architecture Overview — every service API (Magma, sysmem, display) is FIDL over channels.
FIDL wire format / ABI	**Vol. XI** §§Wire Format I–II: the 16-byte transactional header (masked SHA-256 ordinal, txid, magic `0x01`), 8-byte-aligned little-endian objects, the 8-byte v2 envelope, table/union encoding, and handles carried out-of-band, with a worked byte-level encoding.
FIDL versioning / evolution	**Vol. XI** §§Versioning, Evolution: `@available`/API levels, ABI vs. API compatibility, and the soft-transition pattern that tables and flexible types enable. Vol. II builds against a target API level.
Graphics FIDL protocols	**Vol. V** is the dedicated wire-level guide: per-protocol FIDL listings, timing/sequence diagrams, and effects for `fuchsia.ui.composition`, `fuchsia.sysmem2`, `fuchsia.ui.views`/`fuchsia.element`, `fuchsia.hardware.display`, and `fuchsia.gpu.magma`; Vol. I gives the architecture those protocols animate.
sysmem / buffer negotiation	Vol. I §sysmem: Shared Buffer Negotiation is the constraint-based allocator; Vol. V §Allocating Shared Memory walks the `sysmem2` wire protocol; Vol. III §Virtual Memory provides the VMO substrate it hands out.
Scheduling	Vol. III §Thread Scheduling (the fair / deadline arbiter); Vol. I §Frame Scheduling (vsync-driven present deadlines) layers on top.
Vsync / fences → events	Vol. I §Frame Scheduling and §Display Coordinator use fences for present / release; Vol. III §Signals, Waiting, and Ports plus events/eventpairs are the underlying kernel mechanism.
Toolchains	Vol. II §The Toolchain Model, §Compiler Toolchains: Clang & Rust, and §Code Generation (FIDL/Go) define the GN multi-toolchain build; Vol. III §The System Call ABI and the vDSO is the syscall/ABI "cell"; Vol. I treats the Vulkan ICD as a loadable toolchain output.
Drivers & hardware access	**Vol. VII** is the developer's guide: DFv2 driver components, bind rules, the node topology, dispatchers, and taking MMIO/interrupts/BTI from the platform device. Vol. III §Hardware Access is the kernel mechanism underneath; Vol. I §Magma is a real GPU driver; Vol. II packages drivers.
Memory pressure / pager	Vol. III §Physical Memory and Demand Paging (page queues, eviction, the user-space pager); **Vol. VIII** §Demand Paging is its headline client—a file is a pager-backed VMO and a filesystem is its pager, so a file read is a page fault.
Storage & filesystems	**Vol. VIII** is the dedicated treatment: the `fuchsia.hardware.block` FIFO/VMO protocol, FVM vs. multi-volume Fxfs, `fshost`, the `fuchsia.io` node protocol, Fxfs (LSM trees, journal, allocator), demand paging, and the Merkle-verified blob store (Blobfs/Fxblob). Vol. VII supplies the block driver underneath; Vol. III the VMO/pager; Vol. II the packages the blob store holds.
Restricted mode / Starnix	Vol. III §The Security Model: Policy, Sandboxing, and Restricted Mode defines the mechanism; **Vol. IV** is the full Starnix Linux runtime built on it (execution loop, task model, VFS, Binder); Vol. I §Starnix: Linux Compatibility Graphics is the graphics-facing client.
Linux syscall emulation	**Vol. IV** §System-Call Dispatch and §The Virtual File System reimplement the Linux ABI; Vol. III provides the VMOs, threads, and channels underneath.
Binder / Android	**Vol. IV** §Android Support (in-process Binder, ashmem, GPU pass-through); Vol. I supplies the magma/sysmem graphics path it bridges to.
Components / capability routing	**Vol. VI** is the dedicated treatment: realms, CML, capability types, routing semantics, namespaces, lifecycle, collections, runners, RealmBuilder; Vol. III supplies the channel/handle substrate; Vol. II packages and assembles the components.
Packages / product assembly	Vol. II §Packages, Components & Assembly and §From `fx set` to a Running Image (Product Assembly into a bootable image); Vol. VI §Lifecycle (resolvers turn `fuchsia-pkg://` URLs into running components); **Vol. VIII** §Content-Addressed Storage is where those packages physically live and are Merkle-verified; **Vol. XII** is the dedicated treatment of how they are named, distributed, resolved, and updated.
Software delivery / resolution	**Vol. XII** is the dedicated treatment: the package (`meta.far`/FAR, `meta/contents`, blobs, the `fuchsia-pkg://` URL), the base/cache/universe package sets, TUF repositories (signed root/targets/snapshot/timestamp), and resolution (`pkg-resolver`/`pkg-cache` → a `fuchsia.io` directory). Vol. VIII stores and Merkle-verifies the blobs; Vol. VI resolves components.
OTA / system update / paver	**Vol. XII** §§The Update Package–Anti-Rollback: the update package (`packages.json`, `epoch.json`, images), the `system-updater` (`fuchsia.update.installer`), `omaha-client`, the `fuchsia.paver` writing the *inactive* A/B/R slot, and the `epoch.json` anti-rollback backstop with health-checked commit/rollback. Vol. III supplies verified boot.

continued on next page

TABLE I: Cross-volume concept index *(continued)*.

Concept	**Where treated (volume + section pointer)**
Power management / suspend	**Vol. IX** is the dedicated treatment: the Power Framework (Power Broker's elements/levels/dependencies and leases), the System Activity Governor and suspend-to-idle, wake alarms, and system power-state control. Vol. VII drivers own most power elements; Vol. VI routes the capabilities; Vol. III supplies the timers and scheduler idle path.
Thermal, DVFS & reboot	**Vol. IX** covers CPU performance domains / DVFS (`fuchsia.hardware.cpu.ctrl`), thermal throttling and the legacy Power Manager node graph, and `statecontrol.Admin` reboot/poweroff/mexec (with the shutdown-shim and collaborative reboot). Vol. III is the scheduler/idle underneath.
Networking / sockets	**Vol. X** is the dedicated treatment: Netstack3 (Rust Core + Bindings), the `fuchsia.posix.socket` API and `fdio`, transport/5-tuple demux, and the zero-copy Zircon-socket data path. Vol. III supplies the sockets, eventpairs, and channels; Vol. IV (Starnix) is served by this stack.
Network device (netdevice)	**Vol. X** §The Network Device Protocol: the `fuchsia.hardware.network` rx/tx FIFOs and VMO buffer pool—a NIC DMA descriptor ring, and the direct sibling of Vol. VIII's `fuchsia.hardware.block` protocol. Vol. VII supplies the NIC driver underneath.
Routing, neighbors, DHCP, filter	**Vol. X** covers the routing table (longest-prefix match), the neighbor (ARP/NDP) cache, DNS and DHCPv4/v6, and the netfilter-style filter/NAT engine, plus *netcfg*. Vol. VI routes the capabilities that gate route/address edits.
Diagnostics / Inspect / observability	**Vol. XIII** is the dedicated treatment: *Inspect* (a live status tree in a VMO whose generation-count read is a seqlock), structured *logs* (`fuchsia.logger.LogSink`), *tracing* (`trace-manager` + per-provider VMO ring buffers + the Fuchsia Trace Format), *Cobalt* metrics, and *crash* snapshots, all collected by the **Archivist** and queried via `fuchsia.diagnostics.ArchiveAccessor` under *selectors*. Vol. III supplies the VMO and exception machinery; Vol. VI the routing that reaches every emitter; every other volume's subsystem exposes itself through these pillars.
Security / capabilities / verified boot	**Vol. XIV** is the dedicated synthesis: the object-capability model with no ambient authority (handles + `ZX_RIGHT_*`, Vol. III), the namespace-as-sandbox and capability routing as the access-control policy (Vol. VI), the Trusted Computing Base, the verified-boot chain of trust (hardware root of trust → vbmeta/AVB → the ZBI + rollback index, Vol. XII) and verified execution to the page (the pinned `system_image` Merkle root → per-blob Merkle trees, Vol. VIII), the cryptographic services (CPRNG, `fuchsia.kms`, the `fuchsia.tee` TEE), data-at-rest encryption (`zxcrypt`/Fxfs), the exploit mitigations, and `scrutiny`.

IV. Shared Vocabulary

A handful of terms cut across all three volumes; the volume that defines each fully is noted. (Vol. I = Graphics, Vol. II = Build, Vol. III = Kernel.)

Handle An unforgeable, process-local reference to a kernel object—the bus-grant token of the system. *Vol. III.*

KOID Kernel Object IDentifier: a globally unique number naming an object for the lifetime of the system. *Vol. III.*

Rights Per-handle permission bits (read, write, map, duplicate, ...) that gate every operation through that handle. *Vol. III.*

VMO Virtual Memory Object: the universal buffer—pageable, shareable, copy-on-write capable kernel memory. *Vol. III; used everywhere in Vol. I.*

VMAR Virtual Memory Address Region: the address-space tree into which VMOs are mapped. *Vol. III.*

Channel A bidirectional, message-oriented kernel IPC primitive that also transfers handles; the AXI-Stream of the system. *Vol. III.*

FIDL Fuchsia Interface Definition Language: the typed protocol layer and its versioned byte-and-handle wire ABI, carried over channels; `fidlc` generates per-language bindings. *Vol. XI (the language, wire format, and bindings in full); Vol. III (channel transport); Vol. II (codegen in the build); Vol. V (graphics protocols in practice).*

vDSO The virtual dynamic shared object: the only gateway into the kernel, defining the stable syscall ABI. *Vol. III.*

Port / signal A signal is a state bit on an object; a port aggregates asynchronous signal notifications—the interrupt aggregator. *Vol. III.*

BTI / IOMMU Bus Transaction Initiator and I/O MMU: the bus firewall that confines a device's DMA to explicitly pinned memory. *Vol. III; exercised by Vol. I drivers.*

GN toolchain A GN *label-with-toolchain* names a build context; the same target built in two toolchains yields two outputs. *Vol. II.*

Ninja The low-level build executor GN generates files for. *Vol. II.*

Product assembly The final step that stitches packages and configuration into a bootable system image. *Vol. II.*

Component / package The unit of software distribution (package) and the sandboxed unit of execution with routed capabilities (component). *Vol. II; consumed by Vol. I services.*

sysmem The system-wide buffer allocator that negotiates VMO constraints among producers and consumers. *Vol. I.*

Flatland Fuchsia's modern retained-mode 2D scene-graph compositor API. *Vol. I.*

Magma The GPU driver framework; its Magma System Driver (MSD) maps the GPU's registers and command rings. *Vol. I.*

Restricted mode A Zircon facility letting one thread alternate between trusted host code and untrusted "restricted" code, bouncing back on each guest syscall or fault; the basis of Starnix. *Vol. III.*

Starnix Fuchsia's Linux runtime: a Linux OS personality in user space that runs unmodified Linux/Android binaries via restricted mode and emulates the Linux kernel. *Vol. IV.*

Block protocol The `fuchsia.hardware.block` interface: a lightweight FIFO control plane carrying fixed-size command descriptors over a pre-registered shared-VMO data plane—a DMA descriptor ring. *Vol. VIII.*

Fxfs Fuchsia's principal filesystem (Rust): a log-structured merge engine over object stores, made crash-consistent by a logical journal; also the volume manager and blob host (Fxblob) in modern products. *Vol. VIII.*

Blob / content addressing A write-once, read-often file named by the root of a Merkle tree over its contents, so every page is cryptographically verified on read. Packages and executables are blobs. *Vol. VIII.*

fuchsia.io The node protocol every directory and file speaks: connections over channels, `Open3`, and `GetBackingMemory` exposing a file as a pager-backed VMO. *Vol. VIII.*

Package / `fuchsia-pkg://` A content-addressed, Merkle-rooted bundle of files; a `meta.far` (a FAR archive) names its blobs, and the `meta.far`'s Merkle root is the package identity. *Vol. XII.*

TUF The Update Framework: the signed root/targets/snapshot/-timestamp metadata that makes a package repository tamper-resistant and rollback-protected. *Vol. XII.*

A/B/R & the paver The redundant boot slots (A, B, Recovery) and the `fuchsia.paver` that writes the *inactive* slot during an OTA—the FPGA golden/application fallback-boot analogue. *Vol. XII.*

Inspect A component's live tree of named, typed state, published in a VMO whose generation-count read is a seqlock; read with `iquery/ffx inspect`. *Vol. XIII.*

Archivist The diagnostics platform component that ingests logs, Inspect, and lifecycle events and serves them via `fuchsia.diagnostics.ArchiveAccessor` under *selectors*. *Vol. XIII.*

Selector `moniker:node:property`—the query/probe expression that names which diagnostics to return (the logic-analyzer trigger spec). *Vol. XIII.*

Trace / Cobalt / snapshot The Fuchsia-Trace-Format ring-buffer tracing plane (`trace-manager`), the privacy-preserving Cobalt metrics pipeline, and the crash `snapshot` bundling a minidump with logs and Inspect. *Vol. XIII.*

Capability / object-capability Authority as an unforgeable, communicable *handle* to a kernel object, narrowed by *rights*, with *no ambient authority*: a component acts only through the capabilities routed to it, so its namespace is its sandbox and the routing graph is the security policy. *Vol. XIV (with Vols. III, VI).*

Verified boot / verified execution The unbroken chain of trust: a hardware root of trust → `vbmeta`/AVB (with a rollback index) → the ZBI → the pinned `system_image` Merkle root → per-blob Merkle trees verified *per page* at execution—integrity from silicon to instruction fetch. *Vol. XIV (with Vols. VIII, XII).*

Colophon

All fourteen volumes and this cover are typeset with pdflatex and TikZ from public Fuchsia sources at `fuchsia.dev` and `cs.opensource.google/fuchsia`. They are mutually cross-referencing companions: Volume III (Zircon Kernel) supplies the primitives, Volume II (Build) explains how the system is compiled and assembled, Volume I (Graphics) applies the whole stack, Volume VII (Drivers) and Volume VIII (Storage) land hardware and persistence on the kernel, Volume IX (Power) sequences the whole machine into and out of suspend, Volume X (Networking) carries its packets, Volume XI (FIDL) specifies the language and wire ABI in which all of them talk, Volume XII (Software Delivery) packages, signs, distributes, and over-the-air-updates the whole system, Volume XIII (Diagnostics) instruments and observes the whole running system through the Archivist, Volume XIV (Security & Verified Boot) is the closing synthesis—the capability model and the verified-boot/execution chain of trust that confine and authenticate everything the other volumes build, and Volume IV (Starnix) reuses it all to host the Linux world. A combined PDF—this cover followed by the fourteen volumes in series order—is distributed as `fuchsia_internals_complete.pdf`.

References

[1] Google, "Fuchsia Documentation," 2024–2026. [Online]. Available: https://fuchsia.dev/

[2] *Fuchsia OS: Graphics & Display Pipeline*, Systems Architecture Series, Vol. 1, No. 1, 2026.

[3] *Fuchsia OS: The Build System & Toolchains*, Systems Architecture Series, Vol. 1, No. 2, 2026.

[4] *The Zircon Kernel: Architecture, Subsystems, and the User API*, Systems Architecture Series, Vol. 1, No. 3, 2026.

[5] *Graphics FIDL APIs in Practice*, Fuchsia Internals Series, Vol. 1, No. 5, 2026.

[6] *Driver Development in Practice (DFv2)*, Fuchsia Internals Series, Vol. 1, No. 7, 2026.

[7] *The Component Framework: Realms, Capabilities, and Runners*, Fuchsia Internals Series, Vol. 1, No. 6, 2026.

[8] *Starnix: Running Unmodified Linux on Zircon*, Fuchsia Internals Series, Vol. 1, No. 4, 2026.

[9] *Storage: From Block Device to Filesystem (fxfs & blobfs)*, Fuchsia Internals Series, Vol. 1, No. 8, 2026.

[10] *Power Management: The Power Framework, Suspend-to-Idle, and System Power Control*, Fuchsia Internals Series, Vol. 1, No. 9, 2026.

[11] *Networking: The Netstack3 Stack, Sockets, and the Network Device Protocol*, Fuchsia Internals Series, Vol. 1, No. 10, 2026.

[12] *FIDL: The Interface Definition Language, its Wire ABI, and its Bindings*, Fuchsia Internals Series, Vol. 1, No. 11, 2026.

[13] *Software Delivery & OTA: Packages, Repositories, Resolution, and the System Update*, Fuchsia Internals Series, Vol. 1, No. 12, 2026.

[14] *Diagnostics & Observability: Inspect, Structured Logs, Tracing, Metrics, and Crash Reporting*, Fuchsia Internals Series, Vol. 1, No. 13, 2026.
[15] *Security & Verified Boot: Capabilities, Sandboxing, the Chain of Trust, and Verified Execution*, Fuchsia Internals Series, Vol. 1, No. 14, 2026.