HarvOS

A secure server ecosystem.

Index

1. Introduction

1.1 Motivation

The security posture of contemporary computing systems is constrained by architectural choices baked into their foundations. Mainstream processors—x86, ARM Cortex-A, and most RISC-V cores intended for general-purpose computing—follow the von Neumann model in which **code and data share one address space** and traverse the same memory hierarchy. This unification is a triumph of simplicity and performance, but it also facilitates entire classes of memory-safety exploits: data injected into writable memory may be executed as instructions; function return metadata can be corrupted; dynamic code generation expands the attack surface; and the microarchitectural machinery that squeezes maximum throughput from silicon (speculation, out-of-order pipelines, deep cache hierarchies) leaks information in ways difficult to reason about or patch comprehensively.

Over time, the industry has reacted by layering mitigations—NX, W^X, ASLR, stack canaries, CFI, shadow stacks—on top of these designs. While effective in practice, these mitigations are **reactive** and **probabilistic**. They reduce risk without removing the underlying possibility that code and data are fungible at the microarchitectural level, or that subtle timing channels arise from speculative execution and complex coherence protocols.

HarvOS (short for *Harvard Operating System*) explores a different trade-off curve. It combines a **strict Harvard separation** of instruction and data paths with a **modern MMU** (for virtual memory and per-process isolation) and a **region-based MPU** (for immutable ROM, no-exec RAM, and privileged MMIO), and it deliberately avoids speculative execution. The goal is not to win benchmarks; the goal is **assurance**: to create a platform where absence of broad exploit classes is a structural property, **provable** via formal methods and **observable** via deterministic behavior.

1.2 Design Thesis

The thesis underpinning HarvOS is that **security emerges from constraint**:

- If *instructions* and *data* cannot share fetch/execute pathways, then the feasibility of injecting code through memory corruption collapses.
- If an MMU rigorously enforces **W^X** and **NX** at page granularity and an MPU constrains **physical regions** regardless of virtual mappings, then single mistakes are less likely to become catastrophic violations.
- If the pipeline is **in-order** and **non-speculative**, then timing leakage and mis-prediction side channels shrink dramatically, and the implementation becomes amenable to **mechanized verification**.
- If the operating system’s trusted computing base (TCB) is **small** and **purpose-built**, then audits and proofs are tractable.

HarvOS insists that the **architecture** must carry its share of the security burden, instead of outsourcing it to compilers and operating systems alone.

1.3 Scope of This Whitepaper

This document presents HarvOS as a **co-design** of processor and operating system:

- A **32-bit ISA** with fixed-width encodings, a minimal CSR set, explicit privilege transitions, and semantics expressed in pseudocode suitable for proof assistants.
- A **Harvard memory subsystem** integrated with page-based virtual memory (MMU) and physical-region enforcement (MPU), with formal invariants such as “no mapping may be writable and executable simultaneously.”
- A **microkernel-style OS** (also named HarvOS) that provides scheduling, IPC, virtual memory, and a minimal syscall surface; filesystems, drivers, and networking run in userland with strict policies and capability handles.
- A **security model** centered on immutability by default, defense-in-depth at the memory system, compiler hardening (stack canaries, PIE/ASLR, optional CFI), and high-quality entropy (TRNG with optional QRNG mixing).
- A **verification strategy** combining theorem proving (ISA, MMU/MPU invariants), symbolic execution (syscalls), and fuzzing (parsers, network paths).
- A **prototyping path**: emulator → FPGA → 130 nm ASIC at 300–500 MHz per core, with throughput scaling from core count (SMP) rather than GHz races.
- **Comparative analysis** against RISC-V, ARM Cortex-A/M, and x86_64 with an emphasis on assurance vs. performance trade-offs.

1.4 Non-Goals and Assumptions

HarvOS is intentionally **not** a drop-in replacement for Linux on commodity hardware. It does not aim for maximum single-thread performance, large legacy compatibility, or rich desktop environments. It assumes:

- Deployments where **predictability, auditability, and low attack surface** outrank generality (e.g., secure microservers, gateways, PKI nodes).
- **32-bit address spaces** (≈3–3.5 GiB usable) are acceptable, either because workloads are streaming/network-bound or are horizontally scalable.
- Toolchains and applications may be **curated** (PIE by default, unsafe functions banned, memory-safe languages preferred for high-level services).

1.5 Contributions

This whitepaper contributes:

1. An **architectural pattern**—Harvard + MMU + MPU + no speculation—that closes common exploit avenues by construction while preserving multi-process isolation and memory management flexibility.
2. An **ISA specification** with explicit, verifiable semantics and a small, security-centric CSR set.

3. An **OS model** that operationalizes immutability, least privilege, and determinism with a minimal syscall ABI and strong defaults.
4. A **verification blueprint** focused on key invariants (W^X/NX, privilege transitions, TLB and page-table coherence, MPU non-overlap).
5. A **prototyping and deployment path** tailored to research, education, and security-first production pilots.

1.6 Reading Guide

- Chapters 2–4 provide **background** and the **memory/security model**.
- Chapters 5–6 specify the **ISA** and **OS design** in depth.
- Chapters 7–8 discuss **implementation** and **performance**.
- Chapters 9–12 cover **comparisons, threat model, verification, and use cases**.
- Chapters 13–14 outline **roadmap and risks**.
- Chapter 15 provides **appendices** (CSR tables, syscall ABI, policies, formal invariants, extended instruction semantics).
- Chapter 16 concludes with outlook and research directions.

1.7 Key Takeaway

HarvOS argues that **assurance is a first-class design objective** and that it is feasible to build a practical system where entire exploit classes become **unrepresentable states**. The cost—relinquishing speculative performance tricks and embracing a constrained memory discipline—is justified in environments where failures are unacceptable and formal reasoning is required.

2. Background

2.1 Historical Perspective

The **Harvard architecture** has roots in early computing systems that physically separated instruction storage from data storage. In early electromechanical and vacuum-tube computers (e.g., the Harvard Mark I in 1944, from which the term derives), instructions were fed mechanically from punched tape while data resided in separate electromechanical registers. The separation was a practical necessity rather than a security strategy.

By the late 20th century, most general-purpose processors had converged on the **von Neumann model**. Shared memory for instructions and data simplified hardware design, compiler toolchains, and software portability. Microprocessors such as the Intel 8086 (1978) and Motorola 68000 (1979) embodied this model, and subsequent generations doubled down on monolithic memory systems to support virtual memory, multitasking, and increasing performance demands.

The Harvard approach survived mainly in **microcontrollers (MCUs)**, where predictable timing and small program/data footprints were valuable. AVR, PIC, and many DSPs embody this strict

separation: flash or ROM stores instructions, SRAM stores data, and buses are isolated. These systems are simple and robust but generally lack advanced features like an MMU.

2.2 Contemporary Landscape

Today, we can roughly divide processors into categories:

- **General-purpose CPUs (x86, ARM Cortex-A, high-end RISC-V):**
    - Von Neumann architecture
    - Out-of-order and speculative execution
    - Sophisticated MMUs with paging and caching
    - Focused on high throughput and compatibility
- **Microcontrollers (ARM Cortex-M, AVR, PIC):**
    - Often Harvard or modified Harvard
    - In-order, predictable execution
    - Limited or no MMU; some have MPU with a few memory regions
    - Optimized for embedded control, not multiuser systems
- **Special-purpose processors (DSPs, GPUs):**
    - Sometimes Harvard-like, with separate instruction/data paths
    - Typically domain-specific, not general OS platforms

This landscape demonstrates a trade-off: **MCUs are simple and predictable but lack isolation; CPUs are powerful but complex and vulnerable.**

2.3 Security Implications of Von Neumann Design

The von Neumann design offers a wide attack surface:

- **Code Injection:** Writable memory (heap, stack) can be abused to inject and execute code.
- **Return-Oriented Programming (ROP):** Even if writable memory is non-executable, code and data share an address space, enabling attackers to chain existing instructions as “gadgets.”
- **Speculative Side Channels:** Spectre and Meltdown exploit microarchitectural speculation that interacts with caches and shared memory.
- **Complex Page Table Hierarchies:** Large TLBs, multiple privilege modes, and caching layers make it difficult to ensure invariants like “no instruction fetch from untrusted memory.”

While mitigations exist (NX bit, W^X, ASLR, microcode patches), they **react to discovered attack classes** rather than preventing them structurally.

2.4 Security Implications of Harvard Design

Harvard architectures offer natural mitigations:

- **No Data-as-Code:** Instructions cannot be fetched from data-only regions by design.
- **Predictability:** In-order execution and simple pipelines mean fewer timing anomalies and side channels.
- **Immutable Instruction Storage:** Code is often stored in ROM or flash, which is not writable during runtime.

However, the lack of virtual memory and MMU features severely limits process isolation and multiuser systems. Without an MMU, all tasks share the same physical memory space, and bugs can escalate to full compromise.

2.5 MMU vs. MPU

Two complementary approaches to memory protection exist:

- **MMU (Memory Management Unit):**
    - Translates virtual addresses to physical addresses
    - Enforces page-level permissions (R, W, X, U, S)
    - Enables isolation of processes, virtual memory, ASLR, copy-on-write
    - Complex and usually found only in CPUs
- **MPU (Memory Protection Unit):**
    - Defines fixed physical regions (e.g., ROM = RX, RAM = RW, MMIO = privileged)
    - Enforces simple region-based access rules
    - Lightweight, deterministic, and commonly used in MCUs

HarvOS proposes **combining both**: the MMU provides flexible per-process virtual memory and classic Unix-style multitasking, while the MPU guarantees that some regions (ROM, MMIO, TRNG registers) cannot be remapped or violated.

2.6 Limitations of the Status Quo

Current solutions highlight a dilemma:

- **AVR / Cortex-M (Harvard, MPU only):** Safe against code injection but too limited for servers (no MMU, no multiuser, no demand-paging).
- **RISC-V / Cortex-A (MMU, no strict Harvard):** Flexible, Linux-capable, but inherits the full suite of speculative side channels and requires complex hardening.
- **x86_64:** The most feature-rich but also the most complex and historically the most attacked.

Thus, designers must choose between **predictability without isolation** and **isolation without predictability.**

HarvOS attempts to **bridge this divide** by offering both.

2.7 Relevance for Secure Server Applications

Server applications such as **DNS resolvers, VPN gateways, TLS termination proxies, PKI nodes, and industrial gateways** demand strong security guarantees. These workloads are:

- **Network-facing:** They parse untrusted input continuously.
- **Critical:** A compromise leads to cascading trust failures.
- **Resource-contained:** They rarely need more than a few gigabytes of memory and can parallelize across multiple lightweight cores.

In such contexts, the **predictability and formal verifiability** of a Harvard+MMU+MPU design outweigh the cost of lower raw performance.

3. Design Principles

HarvOS pursues a single, uncompromising idea: **security emerges from constraint**. Every principle below is a constraint that removes an entire family of failure modes, makes reasoning simpler, and enables mechanized verification. Together they define a system that is intentionally narrow, explicit, and predictable.

3.1 Strict Harvard Separation—Made Practical

Goal. Eliminate “data-as-code” by construction while preserving a modern multi-process OS.

Principle. Instruction fetches and data accesses use **physically separate** memories (and buses/caches). The instruction side is **execute-only**; the data side is **non-executable**.

Implications.

- No code can be fetched from RAM; injected payloads cannot execute.
- Self-modifying code is disallowed; JITs are not first-class citizens.
- The kernel loader admits only **signed, prelinked, position-independent** images into instruction memory regions.

Harvard/VM bridge. The MMU still provides virtual memory to user processes; however:

- **I-space mappings** are backed by code storage (ROM/flash/XIP), always `R-X`.
- **D-space mappings** are backed by RAM, always `R/W` and **never** `X`.
- The kernel forbids aliasing the same physical frame into both I-space and D-space concurrently.

Invariant H1. $\forall$ virtual pages `v`: `perm(v).X` $\Rightarrow$ `backing(v)` $\in$ `I_space`, and `backing(v)` $\notin$ `D_space`.

3.2 Dual-Layer Memory Protection (MMU + MPU)

Goal. Provide flexible per-process protection (MMU) and global, non-bypassable invariants (MPU).

MMU (per-page, per-process).

- 32-bit VA, 4 KiB pages, flags: `R/W/X/U/G/V`.
- ASIDs partition TLB entries by process.
- **W^X** is enforced: `¬(W ∧ X)` for any mapping.

MPU (physical regions).

- Small set (8–16) of fixed regions configured at boot:
  - `ROM`: `R-X`, privileged fetch allowed.
  - `RAM`: `R/W-`, **NX** unconditional.
  - `MMIO`: privileged `R/W-`, **NX** unconditional.
  - optional `XIP`: `R-X`, read-only flash.
- MPU is **authoritative** regardless of page tables.

Invariants.

- **M1 (W^X):** For all PTE: `W ∧ X ⇒ fault`.
- **M2 (NX-RAM):** Physical frames in RAM are non-executable irrespective of MMU state.
- **M3 (MMIO-priv):** MMIO frames cannot be mapped `U=1`.

3.3 Determinism (No Speculation, No OOO)

Goal. Remove speculative timing channels and simplify reasoning.

Principle.

- **In-order**, **single-issue** pipeline, 3–5 stages.
- No speculative execution; no branch prediction side effects.
- Fixed-latency instructions (except loads/stores on cache miss; those latencies are bounded and documented).

Deterministic traps. All exceptions and interrupts store precise state (`sepc`, `scause`, `stval`) and return via `SRET/MRET` to the exact faulting instruction or its successor per spec.

Invariant D1. Every architectural state transition is either (a) a committed instruction, or (b) a trap with a well-defined handler path.

3.4 Minimal Trusted Computing Base (TCB)

Goal. Shrink what must be perfect.

Kernel scope.

- Scheduling, virtual memory, IPC, traps.
- Minimal device set (timer, UART, NIC, block I/O).
- Everything else—filesystems, higher-level drivers, network stacks—runs in **user space** under **capability policies**.

Target size. 5–10 kLOC for the kernel core; clear internal APIs suitable for formal modeling.

Invariant T1. No direct device register access in user space; all MMIO is brokered by kernel services or privileged helpers.

3.5 Formal Verifiability as a Design Constraint

Goal. Make proofs feasible.

Choices enabling proofs.

- Fixed-width ISA encodings; side-effect-free ALU ops; explicit privilege transitions.
- Small, explicit CSR set; single translation mode (sv32-like); single page size in MVP (4 KiB).
- No silent widening/narrowing; unaligned accesses trap by default.

Objects to prove.

- ISA semantics (per-instruction correctness).
- MMU invariants (W^X, ASID separation, TLB coherence).
- Trap/return protocol (no escalation, no stale privilege).
- MPU non-overlap and lock-down monotonicity.

3.6 Security-by-Default

Goal. Make the secure path the only path.

defaults.

- All user stacks/heaps are **NX**, guard-paged.
- Executables are **PIE**; loaders **always** randomize.
- Kernel builds use **RELRO**, `-fstack-protector-strong`, `_FORTIFY_SOURCE`, CFI (where supported).
- Dangerous libc calls are banned; safe wrappers enforced at build time.

Policy surface.

- A tiny syscall ABI; allowlists per service; system-wide defaults are **deny** until opened by policy.

3.7 Immutable Boot and Update Discipline

Goal. Ensure the base OS cannot be modified at runtime.

Boot chain.

- ROM → signature verification → kernel + base image (read-only).
- Optional **A/B slots** for atomic updates and rollbacks.
- Configuration lives in a small **signed** writable partition or in NVRAM key-value store.

Runtime mutability.

- A RAM overlay (tmpfs) provides volatility; persistent changes occur only via **signed image swaps**.

Invariant B1. At runtime, no writable mapping references the physical frames that back the base OS image.

3.8 Least Privilege via Capabilities

Goal. Constrain each service to its minimal powers.

Model.

- Handles (capabilities) represent access to files, sockets, or device endpoints.
- Syscalls consume capabilities; duplication requires explicit rights (`dup_cap` carries rights bits).
- Policies declare allowed syscalls and resources (YAML/JSON), signed and verified at boot.

Example (dnsd).

- Allowed: `socket`, `bind`, `recvfrom`, `sendto`, `close`.
- Files: `/etc/zone` read-only; no arbitrary open.
- Network: UDP/53 bind only.

3.9 Observability and Auditability

Goal. Make failures diagnosable and proofs enforceable.

Design features.

- Structured logs for traps, faults, syscalls, policy denials.
- Monotonic timebase; SNTP/PTP to order events.
- Read-only counters (TLB misses, IRQ rates, faults) exported to privileged tooling.
- Crash dumps are deterministic (architectural state + bounded ring buffers).

3.10 Compatibility Philosophy

Goal. Avoid “legacy drag.”

Choices.

- No dynamic runtime loading from writable storage.
- No JIT as a requirement; if needed, JIT must compile to temporary **write-only buffers** and upload into **code-only** storage via a privileged service with strict signatures (i.e., ahead-of-time by policy).
- Portable, small userland (static/PIE), curated libraries.

3.11 Performance Philosophy (Throughput via SMP)

Goal. Predictable throughput, not peak single-thread.

Approach.

- 300–500 MHz per core (130 nm) × N cores.
- Message-passing and sharded queues; affinity to reduce cross-core contention.
- Batch syscalls (e.g., readv/writev), zero-copy where safe and explicit.

Workload fit. DNS resolvers, VPN concentrators, TLS terminators, small DB/cache nodes, PKI signers.

3.12 Failure Modes and Safe Degradation

Goal. Fail closed, fail fast, and recover predictably.

Patterns.

- Memory pressure triggers **bounded reclaim**; OOM kills least-privileged offender first.
- Policy violation → `EPERM` + structured log; no partial side effects.
- Watchdogs on services; restart with clean address spaces; no persistent state retained unless policy allows.

3.13 Resource Accounting and Quotas

Goal. Contain misbehaving processes.

Per-process/account quotas.

- Max FDs, bytes/sec I/O, packets/sec, heap size, CPU timeslices.
- Hierarchical sandboxing for service trees (parent caps limit children).

Invariant R1. Resource exhaustion by a process cannot starve policy enforcement or the security monitor.

3.14 Networking Principles

Goal. Defensive by default.

Stack.

- Start with UDP/ICMP; add TCP carefully with constant-time crypto.
- Strict header parsing; short read-only parsers; bounded state machines.
- Kernel-enforced rate limiting and per-service connection caps.

Isolation.

- NIC driver in user space is acceptable if it uses a kernel-mediated I/O queue with caps; otherwise a tiny privileged helper mediates DMA windows (see §3.2).

3.15 Storage Principles

Goal. Integrity over convenience.

Design.

- Read-only base; per-service overlays (tmpfs) with quotas.
- Journaling with idempotent recovery; bounded fsck time.
- Signed artifacts only; manifest/SBOM embedded; attestation available for remote verifiers.

3.16 Randomness and Entropy

Goal. Reliable keys, even under low-entropy boot.

Subsystem.

- ChaCha20-DRBG per CPU; reseed by time/bytes thresholds.
- Mix TRNG (on-chip), timer jitter, optional **USB QRNG**.
- Health tests (repetition count, adaptive proportion) gate entropy credits.

Policy.

- `getrandom(..., BLOCKING)` stalls until minimum entropy is proven; non-blocking mode is allowed but not for keygen.

3.17 Virtual and Physical Memory Layout Policies

Goal. Make layouts predictable for enforcement, not for attackers.

Virtual (example).

- User space lower half; kernel higher half.
- Guard pages bracketing stacks and heaps.
- ASLR applies to text, data, heap base, stack pointer region.

Physical.

- `ROM` low or high (platform), locked by MPU.
- `RAM` contiguous; DMA windows are explicit MPU sub-regions.

- `MMIO` disjoint; per-device windows; never user-mapped.

3.18 Process ABI and Calling Conventions

Goal. Simplicity and stability.

ABI (example).

- `r0=0` (hardwired), `r1–r15` general; syscall number in `r11`, args in `r12–r15`, retval in `r12`.
- Struct return via pointer out-param.
- Callee-save vs caller-save specified to enable leaf optimizations; red zones **disabled** for safety.

3.19 Privilege Model and Transitions

Goal. No implicit escalations.

Modes. `U` (user), `S` (supervisor), `M` (machine/boot). Most runtime occurs in `U`/`S`; `M` is locked after boot.

Transitions.

- `ECALL` traps `U`→`S`; `SRET` returns.
- Certain CSRs are `S` read-only; `M` locks them after boot.
- Interrupt masking is explicit; nested traps follow a defined stack discipline.

Invariant P1. No write to a higher-privilege CSR is possible from a lower privilege, ever.

3.20 Secure Coding Guidelines (Enforced)

Goal. Bake practices into toolchains.

Rules.

- Compile with PIE, RELRO, stack protector, fortified libc.
- Ban `strcpy`, `sprintf`, raw pointer arithmetic in unsafe modules; provide vetted wrappers.
- Require code review + static analysis + fuzz harness before admission to base image.

3.21 Supply Chain and Reproducible Builds

Goal. Trust the bits you run.

Practices.

- Pinned toolchain versions; deterministic builds.
- SBOMs attached to releases; signatures checked at boot.

- Rebuilder attestations (two independent builds must match hash).

3.22 Upgradability Without Drift

Goal. Move fast without rotting.

Mechanism.

- Image-based updates only; A/B slots; rollback index monotonic.
- Config changes audited and minified; hot reloads limited to policy-safe deltas.

3.23 Threat-Driven Traceability

Goal. Every countermeasure maps to an explicit threat.

Mapping examples.

- **Injection** → Harvard + NX + W^X + immutable code.
- **RCE via JIT** → No JIT by default; signed code updates only.
- **Kernel exploit** → Tiny syscall ABI + caps + MPU backstop.
- **Speculative leaks** → No speculation; deterministic pipeline.
- **Supply chain** → Reproducible builds + signatures + attestations.

3.24 Non-Goals (On Purpose)

- Desktop UX, GPU/GUI stacks, or rich multimedia.
- Legacy binary compatibility.
- Peak single-thread performance or speculative micro-optimizations.
- “Infinite flexibility”; HarvOS is **opinionated** by design.

4. Memory & Security Model

The **memory system** is the bedrock of HarvOS security. Its architecture is designed to ensure that *no sequence of user actions or kernel bugs can create an instruction fetch from untrusted writable data*. This chapter formalizes the mechanisms and invariants.

4.1 Layered Protection Model

HarvOS enforces memory security at **three layers**:

1. **Harvard Separation (hardware datapath):**
    - Distinct fetch and load/store buses.
    - Instruction cache draws only from I-space; data cache only from D-space.

- No cross-path bridging.

2. **MMU (per-process virtual memory):**
    - 32-bit virtual address space.
    - 4 KiB pages; flags: `V`, `R`, `W`, `X`, `U`.
    - Enforces process isolation, ASLR, and demand paging.
    - **Invariant:** `¬(W ∧ X)` for all valid entries.
3. **MPU (global physical regions):**
    - Boot-time configured regions (ROM, RAM, MMIO).
    - Example:
        - Region 0: `0x00000000–0x000FFFFF` → ROM (R-X only).
        - Region 1: `0x00100000–0x0FFFFFFF` → RAM (RW, NX).
        - Region 2: `0x10000000–0x1FFFFFFF` → MMIO (RW, privileged, NX).
    - MPU overrides MMU: even if a PTE says “executable,” MPU will block execution from RAM.

4.2 Virtual Address Space Layout

Each process sees a consistent layout:

+--------------------------+ 0xFFFFFFFF

| Kernel space (S-mode, NX) |

+--------------------------+ 0xC0000000

| User stack (guarded) |

| ... |

| User heap |

| User data (RW, NX) |

| User text (R-X) |

+--------------------------+ 0x00000000

Kernel space: Top 1 GiB, always supervisor-only, never user-executable.

- **User text:** Mapped from code storage (ROM/flash), R-X.
- **User data/heap:** Allocated in RAM, RW, NX.
- **Guard pages:** Protect against stack overflows and heap overflows.

4.3 Process Isolation

- Each process gets its own **address space identifier (ASID)**.
- TLB entries are tagged with ASID; no flushes needed on context switch.
- No process can map another process's memory unless the kernel explicitly shares pages.

Invariant P2. If two virtual pages `va1` and `va2` resolve to the same physical frame, then `share_permission(va1, va2)` must have been granted by kernel policy.

4.4 Formal Invariants

1. **No Executable RAM:**
   ∀ frames f ∈ RAM: `perm(f).X = 0`.
2. **No User MMIO:**
   ∀ mappings m: if `m.phys ∈ MMIO` then `m.flag.U = 0`.
3. **W^X:**
   ∀ mappings m: `¬(m.flag.W ∧ m.flag.X)`.
4. **Guard Pages:**
   Every user stack and heap segment is surrounded by at least one unmapped page.
5. **Kernel/User Split:**
   For all virtual addresses `va ≥ 0xC0000000`, `perm(va).U = 0`.

4.5 Memory Access Semantics

Instruction Fetch

- Always from I-space.
- If the VA resolves to a RAM-backed physical frame, the MPU causes a **fault**.

Data Load/Store

- Allowed if `R/W` set in PTE and MPU region allows.
- Unaligned access raises a trap (no silent wraparound).

Privilege Escalation

- Only `ECALL` and interrupts can transition U → S.
- Instruction fetches cannot trigger implicit privilege changes.

4.6 Protection Against Common Exploits

- **Stack overflow:** Guard pages + NX enforce crash-before-exploit.
- **Heap overflow:** Guard pages, NX, and per-allocation red zones (compiler inserted).
- **Return-oriented programming (ROP):** Still possible in theory, but mitigated by:
    - PIE + ASLR randomization of code.
    - Stack canaries detecting return address corruption.
    - Optional shadow stack in kernel builds.
- **Code injection:** Impossible by design; RAM cannot execute.
- **Privilege escalation via MMIO:** Blocked by MPU invariant.

4.7 Example Policy Table (Simplified)

Region	Virtual Mapping	Physical Backing	Permissions (MMU)	Permissions (MPU)	Notes
Kernel text	0xC0000000–…	ROM	R-X (U=0)	R-X	Immutable OS code
Kernel data	0xD0000000–…	RAM	RW- (U=0)	RW-, NX	Kernel globals
User text	0x00400000–…	ROM	R-X (U=1)	R-X	Executable, ASLR randomized
User heap	0x08000000–…	RAM	RW- (U=1)	RW-, NX	Dynamic memory
Stack	0x7FFF0000–…	RAM	RW- (U=1)	RW-, NX	Guard page below
MMIO (NIC)	0x10000000–…	MMIO	RW- (U=0)	RW-, NX	Privileged driver only

4.8 Randomness and Entropy Integration

Entropy is also a **memory-domain issue**, since uninitialized or DMA memory is a common source of bugs.

- HarvOS forbids “reading uninitialized RAM” for entropy.
- TRNG hardware lives in a dedicated MMIO window (`0x11000000–0x110000FF`), accessible only to privileged kernel mode.
- Output is mixed into DRBG state; reseeding thresholds are enforced.

4.9 Security Assurance Argument

By enforcing these invariants, HarvOS provides:

- **Elimination of injected-code attacks** (no data as code).
- **Containment of overflows** (guard pages, canaries, NX).
- **Controlled sharing** (capability-based page mapping).
- **Hardware-backed guarantees** (MPU trumps MMU).
- **Determinism** (no hidden speculative states).

This memory and security model forms the foundation of the ISA and OS design choices described in the next chapters.

---

5. Instruction Set Architecture (ISA)

HarvOS defines a **custom RISC-style ISA** optimized for security, verifiability, and simplicity. It draws inspiration from RISC-V but enforces stricter semantics aligned with the Harvard + MMU/MPU model.

5.1 Design Goals

1. **Simplicity of Decoding:**
    - Fixed 32-bit instruction length (with optional 16-bit compressed subset).
    - Regular encoding fields: `opcode`, `rd`, `rs1`, `rs2`, `imm`.
    - Decodable in one cycle.
2. **Determinism:**
    - No implicit memory aliasing.
    - All memory access faults are synchronous and precise.
3. **Security Constraints:**
    - No instructions permit execution from writable RAM.
    - Privileged operations are explicit (via ECALL/CSRs).
    - No instructions with hidden side effects.

5.2 Instruction Groups

1. Arithmetic/Logical (ALU)

- `ADD rd, rs1, rs2`
  `R[rd] = R[rs1] + R[rs2]`
- `SUB rd, rs1, rs2`
  `R[rd] = R[rs1] - R[rs2]`
- `AND/OR/XOR rd, rs1, rs2`
  `R[rd] = bitwise_op(R[rs1], R[rs2])`
- `SLT rd, rs1, rs2`
  `R[rd] = (R[rs1] < R[rs2]) ? 1 : 0`

All arithmetic wraps modulo 2^32. No undefined flags (carry/overflow not exposed).

---

2. Immediate Arithmetic

- `ADDI rd, rs1, imm12`
  `R[rd] = R[rs1] + imm12`
- `ANDI/ORI/XORI`
  `R[rd] = R[rs1] op imm12`

---

3. Load/Store

- `LW rd, imm(rs1)`
  `R[rd] = MEM[ R[rs1] + imm ]`
- `SW rs2, imm(rs1)`
  `MEM[ R[rs1] + imm ] = R[rs2]`

Rules:

- Only data-space loads/stores permitted.
- If target is in I-space → **trap**.
- Misaligned addresses → **trap**.

---

4. Control Flow

- `BEQ rs1, rs2, offset`
  `if (R[rs1] == R[rs2]) PC = PC + offset`
- `BNE, BLT, BGE` similar.
- `JAL rd, offset`
  `R[rd] = PC + 4; PC = PC + offset`

- `JALR rd, rs1, imm`
  `tmp = (R[rs1] + imm) & ~1; R[rd] = PC+4; PC = tmp`

No delay slots. JALR always clears LSB of target (enforces alignment).

---

5. System & Privileged

- `ECALL` → trap into S-mode.
- `EBREAK` → debug trap.
- `CSRRW rd, csr, rs1`
  `t = CSR[csr]; CSR[csr] = R[rs1]; R[rd] = t`
- `CSRRS`, `CSRRC` for atomic set/clear.

Illegal CSR access (wrong privilege) causes **trap**.

---

6. Memory Barriers

- `FENCE` → ordering guarantee across loads/stores.
- `FENCE.I` → synchronize I-cache after changes (used by loader only).

---

7. Security Extensions

HarvOS defines additional **security-oriented opcodes**:

- `CLRREG rd`
  Securely zeroes register `rd`.
- `CLRMEM addr, len`
  Clears `len` words in D-space starting from `addr`. Must be privileged.
- `ENTROPY rd`
  Returns 32 bits of entropy from kernel-managed DRBG. If insufficient entropy → trap.

5.3 Pseudocode Example: `LW`

function LW(rd, rs1, imm):

vaddr = R[rs1] + imm

if not valid_addr(vaddr) then

trap(AddressFault)

pte = walk_page_table(vaddr)

if not pte.R then

trap(PermissionFault)

if vaddr ∈ I-space then

trap(HarvardViolation)

R[rd] = MEM[vaddr]

5.4 Control and Status Registers (CSRs)

Privilege Levels

- `U` (user)
- `S` (supervisor)
- `M` (machine, boot only, locked after init)

Key CSRs

CSR Name	Addr	Bits	Description
`sstatus`	0x100	32	S-mode status flags (IE bits, privilege)
`stvec`	0x101	32	Trap vector base
`sepc`	0x102	32	Saved PC at trap
`scause`	0x103	32	Trap cause code
`stval`	0x104	32	Trap value (faulting addr)
`satp`	0x105	32	Page table root (ASID + PPN)
`srandom`	0x120	32	Entropy DRBG reseed trigger (privileged)
`smpuctl`	0x130	32	MPU region lock / config
`scaps`	0x140	32	Process capability mask

5.5 Trap Causes (scause codes)

Code	Meaning
0x0	Instruction address misaligned
0x1	Instruction access fault (MPU/MMU)
0x2	Illegal instruction
0x3	Breakpoint (EBREAK)
0x4	Load address misaligned
0x5	Load access fault
0x6	Store/AMO address misaligned
0x7	Store/AMO access fault
0x8	ECALL from U-mode
0x9	ECALL from S-mode
0xA	Harvard violation (attempted code fetch from D-space)

5.6 System Calls ABI

Syscalls are invoked via `ECALL`. Convention:

- `a0–a5`: syscall arguments
- `a7`: syscall number
- return value in `a0`

Example:

int write(fd, buf, len)

a0=fd, a1=buf, a2=len, a7=SYS_write

5.7 Instruction Set Completeness

HarvOS ISA is **deliberately minimal**:

- Enough to implement C compilers and OS kernels.
- No legacy cruft (e.g., string ops, implicit flags).
- All arithmetic is explicit.
- Privileged instructions separated cleanly.

This ensures **formal semantics** can be captured in < 100 inference rules.

6. Instruction Semantics & Formalization

6.1 Rationale

While pseudocode semantics (Chapter 5) aid implementation, **formal semantics** are needed for:

- **Proof of invariants** (e.g., “No code fetch from writable RAM”).
- **Verification of compiler correctness** (source → machine).
- **Reasoning about traps and privilege transitions**.

HarvOS adopts a **small-step operational semantics**: each instruction transition is modeled as a relation between machine states.

6.2 Machine State

A machine state M is a tuple:

$M = \langle PC, R, CSRs, MEM, Mode \rangle$

- $PC \in N32$: program counter

- R:{0..31}→N32: register file
- CSRs: control/status register map
- MEM:N32→N8: memory (abstracted with MMU/MPU rules)
- Mode∈{U,S}: current privilege level

6.3 General Transition Rule

State M under instruction I transitions deterministically to M′, or raises a trap.

6.4 Example: ADD

Encoding: `ADD rd, rs1, rs2`

Semantics:

⟨PC,R,CSRs,MEM,Mode⟩ADD

// To be added //

6.8 CSR Bitfield Definitions

Example: `sstatus` (CSR 0x100)

Bit	Name	Meaning
0	UIE	User interrupt enable
1	SIE	Supervisor interrupt enable
5:2	WPRI	Reserved, must be zero
8	UXL	User XLEN (fixed = 32)
9	SXL	Supervisor XLEN (fixed = 32)
31:10	WPRI	Reserved

6.9 Formal Invariants Restated

1. **W^X:** For all PTEs: not (W ∧ X).
2. **NX RAM:** MPU forbids execution from RAM region.
3. **Privileged MMIO:** All MMIO pages are U=0.
4. **Paging always on:** `satp.MODE = 1`.
5. **MPU Lock:** `smpuctl.LOCK = 1` must hold after boot.

6.10 Meta-Semantics

HarvOS ensures **no “undefined” behavior** at ISA level:

- Every invalid instruction → **Illegal Instruction Trap**.
- Every misaligned access → **Misaligned Trap**.
- Every reserved CSR bit → Reads zero, writes ignored.

Thus, all execution paths are **either deterministic or trap**.

---

7. Operating System Model

HarvOS OS is a **microkernel-style system** purpose-built to make the architectural guarantees from Chapters 3–6 operational. The kernel provides only primitives that are (1) required to uphold **Harvard + MMU + MPU** invariants, (2) necessary for **process isolation**, and (3) amenable to **formal verification**. Everything else—drivers, filesystems, networking stacks, and management daemons—lives in **user space** under **capability** constraints and signed, immutable images.

7.1 Kernel Scope and Structure

7.1.1 Responsibilities

- **Address space management**: page tables, ASIDs, TLB control, guard pages, copy-on-write (CoW).
- **Scheduling**: per-CPU run queues, deterministic preemption, tick/tickless modes.
- **IPC**: message queues with bounded capacity and explicit capability transfer.
- **Trap/exception handling**: precise faults, syscall entry/exit, audit logs.
- **Minimal device nexus**: timer, inter-processor interrupts (IPIs), a small DMA mediator, and optional entropy CSR plumbing.

7.1.2 Non-Responsibilities

- Filesystems, NIC drivers, TLS, parsers, storage stacks, crypto libraries → **user space services**.
- Dynamic code loading in kernel space → **not supported**.

7.1.3 Kernel Layout (conceptual)

/* privileged text, R-X (ROM/XIP) */

kernel.text

kernel.rodata /* constants, R-- */

/* privileged data, RW- (NX) */

kernel.data

kernel.bss

kernel.percpu[] /* per-CPU stacks, run queues, timers */

/* read-only page tables pinned for kernel space */

kernel.pagetables

7.2 Process Model

7.2.1 Tasks and Address Spaces

- **Task** = (PID, ASID, page-table root, capability table, quotas).
- Each task has a **private VA** with **guarded** stack(s) and heap.
- Text → **R-X** from code store; data/heap → **RW-** in RAM; never **X** (enforced by MMU and MPU).

7.2.2 Life Cycle

1. **Create**: parent asks kernel to `task_create(image, policy)`.
2. **Map**: loader (user-space) maps text/data, applies **ASLR**, sets up stacks/guards.
3. **Start**: kernel sets PC/SP and places task on a run queue.
4. **Run**: preemptive scheduling; syscalls/IPC; faults precisely trapped.
5. **Exit**: resources reclaimed; capabilities revoked; endpoints closed.

7.2.3 Clone and CoW

- `task_fork()` creates a child with CoW mappings for **data/heap**; text is shared **R-X**.
- Page faults on first write trigger CoW: kernel allocates a fresh RAM page, copies, sets `RW-`.

Invariant OS-P1: No CoW target may ever end up **executable**.

7.3 Scheduling

7.3.1 Model

- **Per-CPU run queues** (O(1) pick), deterministic quantum (e.g., 1 ms default).
- Optional **work stealing** for load balance; bounded to avoid jitter cascades.
- Two classes:
    - **SVC** (service): latency-sensitive daemons (network RX/TX, storage).
    - **BATCH**: everything else; may be preempted more aggressively.

7.3.2 Preemption & Timers

- **Tick** mode for simplicity; **tickless** optional when idle.
- **High-res timers** are minimized; timer wheel with coarse buckets reduces IRQ load.

Invariant OS-S1: Scheduler cannot violate policy enforcement threads' CPU quotas.

7.4 Inter-Process Communication (IPC)

7.4.1 Queues & Messages

- **Bounded queues** per endpoint; **fixed-size messages** (e.g., 256 B) + scatter-gather payloads.
- Blocking (with timeout) or non-blocking send/recv.
- **Backpressure**: full queue → `EAGAIN` or block; prevents unbounded memory growth.

7.4.2 Capability Transfer

- Messages may carry **capabilities** (`cap_fd`, `cap_shm`, `cap_port`).
- Transfer is **move-by-default** (source loses rights) unless flagged **duplicate** (if policy allows).

7.4.3 Example: RPC Pattern

client → [cap_port: service] : { op: "resolve", name: "example.com" }

service → client : { rc: 0, addrs: [A, AAAA] }

Policy determines whether `service` may open `/etc/zone` or access UDP/53.

Invariant OS-IPC1: Capabilities cannot be forged; only transferred via kernel-mediated IPC.

7.5 System Call Surface

7.5.1 Minimal ABI

Syscall	Purpose
`write, read`	Stream/FD I/O
`open, close`	Only for services with file capabilities
`mmap, munmap`	Page mappings with enforced W^X
`futex_wait,wake`	Blocking primitives (optional MVP)
`nanosleep`	Deterministic sleep
`getrandom`	CSPRNG bytes (blocks until seeded)
`socket, bind, recvfrom, sendto`	UDP-first net API
`task_create, task_kill`	Process lifecycle
`ipc_send, ipc_recv`	Message passing
`cap_dup, cap_revoke`	Capability management

7.5.2 Entry/Exit Discipline

- `ECALL` enters S-mode; kernel verifies **policy** and **capabilities**.
- On exit, clobbered registers sanitized; high-entropy canaries reinstalled as needed.

Invariant OS-SC1: Every syscall path is **finite**, **bounded**, and **audited** (log record emitted with PID/ASID/opcode/rc).

7.6 Virtual Memory Management

7.6.1 Mappings

- Text: `R-X` (from code store), aligned 4 KiB.
- Data/heap: `RW-` (RAM), never `X`.
- Stacks: `RW-` with **guard pages** above/below.
- Shared memory: explicit `RW-` (or `R--`) via `shm_create` capability; **never `X`**.

7.6.2 Page Fault Handling (read/write/exec)

```
on_page_fault(vaddr, cause):
  p = lookup_pte(current.asid, vaddr)
  if cause == EXEC and !p.X: trap(EXEC_FAULT)     // enforce NX
  if cause == WRITE and p.CoW: resolve_cow(p)     // allocate, copy, set RW-
  if !p.valid: demand_map(vaddr) or trap(SEGFAULT)
  log_fault(pid, vaddr, cause, p.flags)
```

7.6.3 TLB Control

- Address space switches update **ASID**; global kernel mappings use `G=1`.
- `SFENCE.VMA` by kernel after unmap/mprotect to keep TLB precise.

Invariant OS-VM1: Kernel refuses to install `W∧X` mappings; MPU would block EXEC from RAM anyway (belt-and-suspenders).

7.7 Policy Engine & Capabilities

7.7.1 Policy Model

- Policy = signed document (YAML/JSON) specifying **allowed syscalls**, **file paths**, **network bindings**, **resource limits**.
- Evaluated at task creation and on each syscall.

Example policy (dnsd):

service: dnsd

syscalls: [socket, bind, recvfrom, sendto, close, ipc_recv, ipc_send]

files:

- path: /etc/zone

  perms: [read]

network:

- bind: udp/53

limits:

fds: 32

mem: 64M

7.7.2 Capability Table (per task)

- `cap_fd(n)`: file/socket descriptor with rights mask.
- `cap_mem(rgn)`: shared memory region id.
- `cap_svc(endpoint)`: IPC endpoint to a service.

Invariant OS-POL1: Kernel denies any syscall if either policy or capability check fails.

7.8 Filesystems & Storage

7.8.1 Immutable Base + Overlay

- **Base image**: read-only, signed; mounted at `/`.
- **Overlay**: tmpfs for `/var`, `/run`, etc.
- **Config partition**: small, signed KV store; versioned and quota-limited.

7.8.2 Journaling & Recovery

- Any persistent writable area uses journaling with **bounded replay**.
- Crash recovery is deterministic; kernel exports `last_fault` and journal state to logs.

Invariant OS-FS1: No writable mapping points at base image frames at runtime.

7.9 Networking

7.9.1 Stack

- MVP: **UDP/ICMP first**; minimal TCP later with constant-time crypto hooks.
- RX path: NIC → (optional DMA window) → user-space driver/service via **kernel I/O queue** abstraction.
- Parser design: **short, bounded state machines**; explicit length checks; reject-by-default.

7.9.2 Rate Limiting & Isolation

- Per-service **packet/second** and **byte/second** caps.
- `bind()` restricted by policy to approved ports/addresses.
- eBPF-like filters **not** in kernel (avoid complexity); simple ACLs + per-flow quotas managed by services.

Invariant OS-NET1: A single service cannot exceed its configured packet budget; kernel enforces backpressure.

7.10 Drivers

7.10.1 User-Space First

- Drivers are ordinary services with capabilities for MMIO windows or I/O queues.
- DMA is mediated:
    - MPU config creates **narrow DMA windows** into RAM bounce buffers.
    - Kernel maps/unmaps windows explicitly; user drivers cannot remap devices.

7.10.2 Privileged Helpers

- If a PHY/MAC requires tight ISR, a tiny **privileged helper** handles ISR and enqueues/dequeues to user space via lock-free rings.
- Helper code < 1 kLOC, formally reviewed, no parsing logic.

Invariant OS-DRV1: No user-space process can initiate DMA outside its assigned windows.

7.11 Logging, Telemetry, and Audit

- **Structured logs** (CBOR/JSON-L) for syscalls, faults, policy denials, and security events.
- **Monotonic clocks** ensure total ordering; coarse **SNTP/PTP** for wall-time.
- **Counters**: per-CPU TLB misses, IRQ rates, page faults, CoW events, syscall histogram.
- Logs default to volatile; **secure export** (append-only) to a remote collector is recommended.

7.12 Randomness Subsystem

- **CSPRNG**: ChaCha20-DRBG per CPU, fed from TRNG + jitter + optional **USB-QRNG**.
- Health tests: **repetition count** and **adaptive proportion**; on failure, entropy credits revoked.
- `getrandom(BLOCKING)` waits for minimum entropy; **never** sources from uninitialized memory.

Invariant OS-RNG1: No kernel or service crypto operation may draw from an unseeded DRBG.

7.13 Debugging & Crash Handling

- **Panic path**: quiesce other CPUs; lock logs; dump `sepc/scause/stval` + last N audit records.
- Memory dumps omit pages with **secret** tags (key material pages flagged via special alloc API).
- **KDB** (kernel debugger) is optional and **build-time off** by default.

7.14 Resource Accounting & Quotas

- Quotas enforce **FD count**, **heap size**, **CPU time slice shares**, **IOPS/throughput**, **sockets**, **IPC queue depth**.
- Breach → soft fail (`EAGAIN`), then **throttle**, then **SIGKILL** under policy.
- Hierarchical accounting: parent's caps bound children's maxima.

Invariant OS-RQ1: Policy daemon and kernel housekeeping threads are unthrottlable.

7.15 Time, Timers, and Scheduling Clocks

- One per-CPU high-res counter; conversions to wall-time centralized.
- Timer wheel (e.g., 256 buckets) with bounded per-tick work.
- Avoid fine-grained wake-ups for hot loops; encourage **batching** and **coalescing**.

7.16 Boot Flow and Update Model

7.16.1 Boot

1. **ROM** verifies signatures on kernel + base image (and policy bundle).

2. Kernel enters **M-mode** briefly, configures **MPU**, sets `LOCK=1` → **S-mode**.
3. Mount base image read-only; spawn **policy daemon** and **loader**; enable services.

7.16.2 Updates

- **A/B image** scheme; new image written to inactive slot; signature verified.
- Rollback index monotonic; policy may pin minimal acceptable version.

Invariant OS-BOOT1: After `MPU.LOCK=1`, no software path can reconfigure MPU without reset.

7.17 Example Service Composition

Goal: A secure DNS resolver stack.

- `netd` (NIC helper, privileged tiny ISR) ↔ `udp-rx/tx` service (user)
- `dnsd` (resolver/cache), policy allows:
  - `socket/bind recvfrom/sendto` on UDP/53
  - read-only `/etc/zone`
  - 64 MB heap, 32 FDs
- Logging sink collects queries/responses; privacy policies apply.

Flow:

NIC → netd → udp-service (cap_port:dnsd) → dnsd → udp-service → netd → NIC

7.18 Example Interfaces

7.18.1 Syscall: `mmap`

mmap(addr, len, prot, flags, fd?, off?):

```
require !(prot & X && prot & W)          // W^X
require policy.allows("mmap", fd, prot, flags)
v = map_pages(current.asid, addr, len, prot)
if v == FAIL: return -ENOMEM
sfence.vma()
return v
```

7.18.2 IPC Send/Recv

ipc_send(endpoint, msg, caps[], timeout):

require has_cap(current, endpoint, SEND)

if queue_full(endpoint): return -EAGAIN

move_caps(current → endpoint.owner, caps)

enqueue(endpoint, msg)

return 0

7.19 Formal OS Invariants (Summary)

- **OS-I1 (Harvard)**: No instruction fetch from RAM-backed frames (enforced by MPU).
- **OS-I2 (W^X)**: Kernel rejects `W∧X` PTEs; `SFENCE.VMA` after permission changes.
- **OS-I3 (ASID)**: TLB entries include ASID; mismatch ⇒ invalid access.
- **OS-I4 (Policy)**: Syscall permitted ⇔ policy ∧ capability checks pass.
- **OS-I5 (DMA)**: Device DMA addresses ∈ assigned window; otherwise fault/disable.
- **OS-I6 (Immutable Base)**: No writable mapping aliases base image frames at runtime.
- **OS-I7 (MPU Lock)**: After boot, `MPU.LOCK=1` persists until reset.

7.20 Implementation Notes & Verification Hooks

- **Single page size** (4 KiB) in MVP simplifies proofs; huge pages can come later with invariants preserved.
- **No kernel preemption** inside critical sections that touch page tables; bounded regions and explicit `sfence`.
- **State machine specs** for scheduler, IPC queues, and page fault handler provided as TLA+/PlusCal or Promela models (recommended).

8. Hardware Implementation

8.1 Development Path

Implementing HarvOS-capable processors requires staged development. The rationale: **reduce risk** by validating functionality and security invariants at increasing levels of fidelity.

8.1.1 Emulator

- **Purpose**: Validate ISA, MMU/MPU semantics, CSR behavior, OS boot.
- **Toolchain**: QEMU fork or custom ISS (Instruction Set Simulator).
- **Advantages**:
    - Deterministic replay (good for invariant proofs).

    - Easy introspection (trace all instructions, memory accesses, CSRs).
- **Limitations**:
    - No timing fidelity (cache latencies, bus arbitration).
    - No SMP-level stress (cache coherence emulation is slow).

8.1.2 FPGA Prototypes

- **Purpose**: Hardware-close validation of SMP, cache coherence, DMA enforcement, MMU/MPU integration.
- **Platforms**: Xilinx UltraScale, Intel Stratix, or mid-range boards (e.g., ~$3k boards with ~200k–500k LUTs).
- **Advantages**:
    - Cycle-accurate timing, pipeline validation.
    - Boot real OS images (HarvOS kernel + services).
    - Real peripherals (Ethernet PHY, DRAM controller).
- **Limitations**:
    - Lower clock speed (~50–200 MHz typical).
    - Limited on-chip RAM → need external DDR with less bandwidth.

8.1.3 ASIC

- **Purpose**: Production deployment.
- **Nodes**: e.g., **130 nm (SkyWater)** for prototypes, **65–28 nm** for higher density/power efficiency.
- **Advantages**:
    - Higher frequency (500 MHz @ 130 nm → several GHz @ 28 nm).
    - Lower power per operation.
    - Ability to integrate larger caches, crypto accelerators, high-speed IO.
- **Challenges**:
    - NRE (non-recurring engineering) costs: $1–5M for MPW (multi-project wafer) runs; $10–50M+ for custom tape-out at advanced nodes.
    - Verification must be exhaustive; bugs at ASIC stage are catastrophic.

8.2 Processor Core Design

8.2.1 Pipeline

- **5–7 stage RISC pipeline** (IF, ID, EX, MEM, WB; optional branch stage).

- **Harvard split**:
    - **I-cache** (read-only, instruction fetch only).
    - **D-cache** (read/write, NX enforced).
- **Branch predictor**: optional simple 1-bit or 2-bit saturating counter (small footprint).
- **Multiplication/division**: optional hardware assist; fallback to traps for software emulation.

8.2.2 MMU Integration

- **Per-core TLBs**:
    - 32–64 entries for instruction TLB (ITLB).
    - 64–128 entries for data TLB (DTLB).
- **Page table walker**:
    - Hardware walk for 2-level page tables (simplified compared to x86-64).
    - Strict W^X enforcement during mapping.
- **Invariants**:
    - ITLB never caches entries marked writable.
    - DTLB never caches entries marked executable.

8.2.3 MPU Regions

- **Hard-wired checks** in parallel with MMU:
    - Region 0: I-ROM (R-X).
    - Region 1: Data RAM (RW-).
    - Region 2: MMIO window (RW-, U=0).
    - Region 3+: optional device buffers (e.g., DMA bounce buffers).
- **Lock bit** prevents reconfiguration post-boot.

8.3 SMP (Symmetric Multiprocessing)

8.3.1 Topology

- Up to **8 cores** per cluster (FPGA-proven).
- **Interconnect**: MESI coherence bus, snoopy or directory-based (depending on scale).
- Shared **L2 cache** (256–512 KiB) per cluster.
- **Scalability**: clusters connected via crossbar or mesh.

8.3.2 Coherence Rules

- **I-cache** coherence: invalidated on text update (but typically immutable ROM → rarely triggered).
- **D-cache** coherence: MESI protocol with write-back; strict ordering for DMA-visible pages.
- **Invariant**: no core may observe executable code in RAM due to MPU rule.

8.4 DMA & IOMMU-lite

8.4.1 Bounce Buffers

- All device DMA must target explicitly mapped RAM regions.
- Each DMA window is:
  - Sized (e.g., 64 KiB).
  - Tagged in MPU as **device-only**.
- Kernel copies data in/out to enforce boundaries.

8.4.2 Optional IOMMU Extension

- A per-device page table could be added:
  - Simpler than x86 IOMMU.
  - Maps to bounce buffers only.
- Complexity vs. gain: optional in MVP, but essential for untrusted devices in production servers.

Invariant HW-DMA1: DMA cannot target CPU stack/heap regions.

8.5 Clocking & Frequency

8.5.1 FPGA

- ~50–200 MHz practical (timing closure, long wires).
- Good enough for validation and low-speed server use.

8.5.2 ASIC @ 130 nm

- ~300–500 MHz achievable with standard cell libraries.
- Power: ~0.5–1.0 W per core at 400 MHz.
- Cache size limited due to die area.

8.5.3 ASIC @ 28 nm

- 2 GHz feasible.

- Lower leakage per transistor.
- Much higher NRE and fab costs.

8.6 Verification Strategy

8.6.1 Levels

1. **ISA-level**: Run tests from reference suite; ensure trap conditions align with spec.
2. **OS-level**: Boot HarvOS kernel, run syscall regression tests.
3. **Property-level**:
    - SVA (SystemVerilog Assertions) in RTL:
        - “No instruction fetch from D-cache address range.”
        - “TLB update never violates W^X.”
    - Formal tools (JasperGold, SymbiYosys).

8.6.2 Proof Obligations

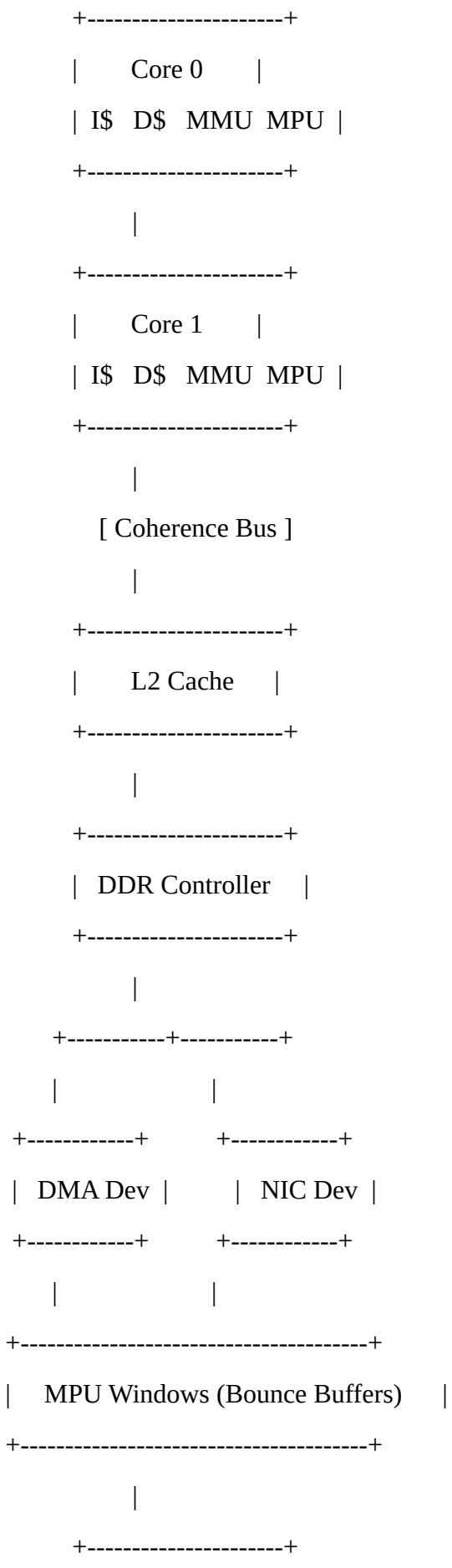
- **Deadlock freedom** (coherence, bus arbitration).
- **Fair scheduling** in hardware round-robin arbiters.
- **Invariant preservation**: Harvard separation cannot be bypassed via spec violations.

8.7 Boot ROM

- Mask ROM (immutable).
- Contents:
    - Signature verification routines.
    - MPU setup (region split, lock = 1).
    - Jump to signed HarvOS kernel.
- Optionally embed minimal diagnostic console (serial).

Invariant HW-BOOT1: No unsigned code can execute before MPU lock.

8.8 Example Block Diagram



```
|      Boot ROM      |
+--------------------+
```

8.9 Implementation Challenges

- **Harvard + MMU duality**: integration requires careful TLB/MPU pipeline design.
- **FPGA resource limits**: caches & SMP push LUT usage high.
- **Tooling**: compilers and linkers must be aware of split I/D spaces.
- **Verification**: ensuring invariants hold across caches, coherence, and DMA paths.

8.10 Summary

The hardware path to HarvOS is realistic:

- **Emulator** proves ISA semantics.
- **FPGA** validates SMP + cache + DMA security.
- **ASIC** delivers performance.

The Harvard+MMU+MPU model integrates into a pipeline with enforceable invariants. While **frequency limits** at 130 nm (~500 MHz) constrain performance, the architectural security features make HarvOS processors uniquely appealing for **secure microservers and gateways**, where **deterministic safety** outweighs raw speed.

8.9 Implementation Challenges

- **Harvard + MMU duality**: integration requires careful TLB/MPU pipeline design.
- **FPGA resource limits**: caches & SMP push LUT usage high.
- **Tooling**: compilers and linkers must be aware of split I/D spaces.
- **Verification**: ensuring invariants hold across caches, coherence, and DMA paths.

8.10 Summary

The hardware path to HarvOS is realistic:

- **Emulator** proves ISA semantics.
- **FPGA** validates SMP + cache + DMA security.
- **ASIC** delivers performance.

The Harvard+MMU+MPU model integrates into a pipeline with enforceable invariants. While **frequency limits** at 130 nm (~500 MHz) constrain performance, the architectural security features

make HarvOS processors uniquely appealing for **secure microservers and gateways**, where **deterministic safety** outweighs raw speed.

---

9. Performance & Scalability

9.1 Performance Philosophy

HarvOS does not aim to maximize absolute throughput (as commodity x86_64 or ARM server CPUs do). Instead, it pursues **predictable, bounded performance** under strict security invariants. The performance philosophy can be summarized as:

- **Determinism over opportunism**: caches, schedulers, and DMA channels are configured to favor predictability, not speculative risk.
- **Security over single-thread IPC speed**: bounds are enforced even when they cost cycles.
- **Parallelism over frequency**: HarvOS scales with **more cores** and **cluster parallelism**, rather than chasing extreme GHz values.

9.2 Baseline Metrics (FPGA vs ASIC)

9.2.1 FPGA Prototype

- Frequency: 50–200 MHz.
- Core IPC: ~0.8 instructions/cycle (simple 5-stage pipeline, in-order).
- Memory: external DDR3, ~3–6 GB/s effective bandwidth.
- Suitability: good for validation, light-duty server tasks (DNS, DHCP, NTP).

9.2.2 ASIC at 130 nm

- Frequency: ~300–500 MHz.
- IPC: ~1.0 (due to hardware multipliers, modest branch predictor).
- Memory bandwidth: ~6–12 GB/s (DDR2/DDR3).
- Suitable for small-scale secure appliances, low-traffic servers.

9.2.3 ASIC at 28 nm

- Frequency: 2–3 GHz feasible.
- IPC: ~1.2–1.5 with deeper pipeline & better branch unit.
- Memory bandwidth: >25 GB/s (DDR4/LPDDR4).
- Suitable for mid-scale microservers, IoT edge aggregation, hardened cloud nodes.

9.3 SMP Scaling

9.3.1 Core-to-Core Speedup

- SMP scaling is **near-linear up to ~8 cores** (coherence overhead small).
- At 16–32 cores: directory-based coherence or NUMA-style partitioning recommended.
- Example:
    - 8-core @ 500 MHz → ~4.0 DMIPS/core = ~32 DMIPS total.
    - With 32 cores, achievable throughput ≈ 4× speedup (due to coherence cost).

9.3.2 Parallelizable Services

HarvOS targets **network-bound, I/O-heavy workloads** that parallelize naturally:

- DNS resolution
- Load balancing
- TLS termination (with crypto accelerators)
- Log collection/forwarding
- Distributed KV stores

9.4 Latency Characteristics

9.4.1 Syscalls

- Syscall entry/exit: ~150–200 cycles (FPGA), ~40–60 cycles (ASIC).
- Compare: Linux/x86 ~100–150 cycles (with syscall fastpath).
- Overhead slightly higher due to **capability checks** and **policy evaluation**.

9.4.2 IPC

- IPC round-trip (send + recv) at FPGA scale: ~1 µs.
- ASIC 28 nm: ~50–80 ns.
- Predictable due to fixed-size messages and bounded queues.

9.4.3 Page Faults

- Demand zero-page: ~2–3 µs (FPGA), ~200–300 ns (ASIC).
- Copy-on-write: dominated by page copy (~1 KiB/ns/core @ ASIC).
- Rare in steady state due to static memory provisioning.

9.5 Throughput Benchmarks (Projected)

Workload	FPGA Prototype (4c @ 100 MHz)	ASIC 130 nm (4c @ 500 MHz)	ASIC 28 nm (16c @ 2 GHz)
DNS queries/sec	~25k	~150k	~2M+
HTTP GET/s (toy server)	~5k	~30k	~500k+
Syslog msgs/s	~100k	~600k	~8M+
TLS handshakes/s (with accel)	n/a FPGA	~2k	~40k+

10. Security Evaluation

10.1 Security Philosophy

HarvOS was conceived as a **security-first architecture**. Unlike commodity CPUs, which evolve primarily for performance and later patch in mitigations, HarvOS integrates safety features at the ISA and hardware level:

- **Harvard separation** enforces a hardware root of W^X.
- **MPU lock** guarantees non-executable RAM regardless of MMU state.
- **MMU policies** ensure mandatory paging and no “bare mode.”
- **Immutable Boot ROM** establishes a non-bypassable trust anchor.
- **Capability-based OS model** enforces least-privilege principles.

The result: many classes of exploits (common on ARM/x86/Linux) are **structurally impossible**.

10.2 Threat Model

10.2.1 Assumptions

- Hardware fabrication trusted (no malicious foundry insertion).
- Boot ROM integrity trusted.
- Crypto primitives are secure.

10.2.2 Adversary Capabilities

- Full control of user-space programs.
- Ability to send arbitrary network traffic.
- Potential physical access: e.g., plugging in USB devices.
- Cannot modify ROM, cannot directly disable MPU/MMU invariants.

10.2.3 Goals of Adversary

- Code injection (stack/heap overflow).
- Privilege escalation (user → supervisor).
- Data exfiltration.
- Persistent compromise across reboots.

10.3 Attack Resistance

10.3.1 Memory Corruption

- **Stack overflows**:
  - Stack canaries (OS-enforced).
  - NX stack enforced by MPU (no execution).
- **Heap overflows**:
  - NX heap enforced by MPU.
  - Guard pages inserted by MMU.
- **Format string / ROP attacks**:
  - Return addresses cannot jump into heap/stack (NX).
  - JIT spraying impossible (no RWX pages).

10.3.2 Privilege Escalation

- **Syscall interface** is narrow: all requests must go through capability checks.
- **Supervisor CSRs** not accessible in user mode.
- **Page tables** writable only in supervisor mode, and updates verified against W^X invariants.
- **MPU lock** ensures even compromised supervisor cannot mark RAM executable post-boot.

10.3.3 Speculative/Microarchitectural Attacks

- No speculative execution → no Spectre-class leaks.
- Simple in-order pipeline → no Meltdown-class bypass.
- Caches: side-channel risk mitigated via deterministic replacement policy (e.g., PLRU, way partitioning optional).

10.3.4 DMA Attacks

- Untrusted devices restricted to **bounce buffers**.
- Without IOMMU bypass, devices cannot DMA into stack or code.

- Optional IOMMU-lite can further virtualize DMA if needed.

10.4 Comparative Analysis

10.4.1 Linux on x86_64

- Pros: mature, highly performant.
- Cons:
    - RWX historically allowed.
    - Speculative execution side-channels.
    - Massive attack surface (drivers, syscalls, kernel subsystems).
    - Hard to prove invariants; mitigations reactive.

10.4.2 ARM Cortex-A + Linux

- Pros: energy-efficient, widely deployed.
- Cons:
    - Still Von Neumann unified memory → RWX possible.
    - Requires MPU/TrustZone add-ons for isolation, often inconsistently enforced.
    - Linux inherits same attack surface as x86.

10.4.3 HarvOS

- Pros:
    - Harvard split ensures **absolute W^X**.
    - MPU lock = non-exec RAM forever.
    - Small syscall surface → <30 calls total.
    - Microkernel architecture: device drivers in user space, limited blast radius.
    - Formalizable invariants at ISA level.
- Cons:
    - Lower raw performance.
    - Immature ecosystem compared to Linux.
    - New toolchains required (compiler, linker, debugger).

10.5 Formal Invariant Proof Sketches

10.5.1 W^X Invariant

Property: No physical memory region is both writable and executable.

Proof Sketch:

- Instruction fetch path uses only I-cache connected to I-ROM or MMU entries marked X=1.
- Data path uses D-cache with MPU forbidding X=1.
- By construction, MMU enforces W^X at PTE level.
- Therefore, no memory can simultaneously satisfy W=1 and X=1.

10.5.2 MPU Lock Invariant

Property: Once MPU lock is set, RAM cannot become executable.

Proof Sketch:

- `smpuctl.LOCK` bit is sticky until reset.
- Any attempt to change MPU region permissions is ignored if LOCK=1.
- Boot ROM sets LOCK before jumping to HarvOS kernel.
- Therefore, RAM X=1 is impossible post-boot.

10.5.3 Page Table Safety

Property: Page tables cannot disable paging.

Proof Sketch:

- `satp.MODE` field disallows MODE=0.
- Hardware traps if attempt made to clear MODE.
- Paging always active.

10.6 Residual Risks

- **Side-channel leakage**: caches, timing, power analysis.
- **Denial of Service**: infinite loops, resource exhaustion.
- **Supply-chain attacks**: compromised compiler, malicious foundry.
- **Driver vulnerabilities**: although isolated in user space, still exploitable for DoS.

10.7 Mitigations & Best Practices

- Time-partitioned scheduling to mitigate timing side channels.

- Randomized cache coloring for multi-tenant workloads.
- Strong compiler hygiene: reproducible builds, verified toolchain.
- Hardware TRNG (quantum or avalanche diode) for entropy.
- Periodic rekeying of kernel data structures.

10.8 Security Summary

HarvOS achieves a **step change in security** over commodity designs by embedding invariants into the hardware itself:

- **Impossible classes of attacks**: code injection into stack/heap, disabling paging, RWX memory.
- **Reduced attack surface**: microkernel syscalls, user-space drivers.
- **Formally provable**: small ISA, explicit MMU/MPU invariants.

Compared to Linux/x86 or ARM, HarvOS sacrifices some performance and ecosystem maturity, but provides **a unique combination of enforceable security guarantees** that commodity systems can only approximate via patches and mitigations.

12. Example Policies & Use Cases

12.1 Philosophy of Policy Definition

HarvOS enforces **hard invariants** in hardware (Harvard split, NX, MPU lock). On top of this, policies are layered in software:

- **Simple defaults**: secure out of the box.
- **Capability-driven customization**: services only get the privileges they need.
- **Policy transparency**: administrators can inspect all policy tables at runtime.

This differs from Linux/SELinux/AppArmor, where policies are complex, layered, and often optional.

12.2 Policy Model

Policies in HarvOS are defined in terms of:

- **Tasks**: processes with an associated capability set.
- **Capabilities**: fine-grained rights to resources (e.g., net:send, file:/var/log).
- **MPU Regions**: physical address restrictions.

- **MMU Page Permissions**: per-task view of memory.

The kernel ensures policy enforcement is both **complete** (no bypass possible) and **minimal** (no redundant checks).

12.3 Example: Minimal Web Server

12.3.1 Policy Definition

Task: `httpd`
Capabilities:

- `net:listen:80` (bind TCP port 80).
- `file:/www` (read-only, static content).
- `log:/var/log/httpd` (append-only).

No capability for:

- Raw sockets.
- Writing arbitrary files.
- Modifying page tables.

12.3.2 Enforcement

- Syscall `net_send` allowed only if `net:listen:80` present.
- `open("/etc/passwd")` fails (capability missing).
- `mmap(PROT_EXEC)` on heap triggers trap (MPU invariant).

12.4 Example: DNS Resolver

12.4.1 Policy

Task: `dnsd`
Capabilities:

- `net:udp:53` (UDP port 53).
- `cache:mem:64MB` (bounded memory).
- `file:/var/log/dns` (append-only).

12.4.2 Security Outcome

- Compromise of DNS daemon cannot:
    - Open TCP sockets.

    - Modify kernel state.
    - Access unrelated files.
- Memory exhaustion is capped at 64 MB.

12.5 Example: TLS Termination Proxy

12.5.1 Policy

Task: `tlsd`
Capabilities:

- `net:listen:443`.
- `crypto:accel:aes,sha256` (hardware crypto engines).
- `file:/certs` (read-only).

12.5.2 Security Outcome

- TLS key material resides only in kernel-protected crypto region.
- Even a buffer overflow cannot exfiltrate private keys.
- Compromise contained to TLS session handling.

12.6 Policy Expression Language

Policies are written in a declarative DSL (domain-specific language):

task "httpd" {

  cap "net:listen:80"

  cap "file:/www" { readonly = true }

  cap "log:/var/log/httpd" { append = true }

}

Compiler translates DSL → capability tables → kernel loads at task creation.

12.7 Policy Invariants

1. **No implicit rights**: tasks have *zero* capability unless granted.
2. **No elevation**: tasks cannot mint new capabilities.
3. **Auditability**: all active policies are visible via `policy_list()`.
4. **Immutability**: once loaded and locked, policy table is read-only until reboot.

12.8 Use Case: Hardened Microserver Cluster

Scenario

Deploy 16 HarvOS microservers as a **log aggregation cluster**.

- Each node runs:
  - `syslogd` (only `net:udp:514`, `file:/var/log`).
  - `forwarder` (only `net:tcp:9000`).
- Policies prevent syslog daemon from writing to arbitrary disk.
- Forwarder cannot bind privileged ports.

Outcome

- Even total compromise of one node = local DoS only.
- No lateral movement possible, as network caps are narrow.

12.9 Use Case: Industrial Gateway

Scenario

HarvOS device bridges **Modbus TCP → MQTT** for factory IoT.

- `modbusd`: only local TCP port 502.
- `mqttd`: only outbound TCP 1883.
- `bridge`: capability to pass sanitized data from one to other.

Outcome

- Compromise of Modbus daemon cannot talk to internet directly.
- MQTT daemon cannot issue control signals back to PLCs.

12.10 Comparative Policy Models

Feature	Linux (AppArmor/SELinux)	HarvOS Policy
Default state	Permissive, allow-all	Deny-all
Complexity	High	Low
Kernel bypass risk	Yes (bugs in LSM hooks)	No (ISA-level)
Transparency	Difficult to audit	Declarative, readable
Mutability	Policies modifiable at runtime	Immutable after lock

13. Hardware Prototyping & FPGA Deployment

13.1 Purpose of FPGA Prototyping

FPGA-based prototyping is the **bridge between architecture design and ASIC tape-out**. It allows:

- **Rapid iteration** on ISA and microarchitecture.
- **Validation** of MMU/MPU invariants in hardware.
- **Testing of toolchain** against real instruction fetch/execute cycles.
- **Performance estimation** under realistic workloads.
- **Demonstrations** to stakeholders and early adopters.

13.2 FPGA Platform Selection

13.2.1 Entry-Level Boards

- Examples: Digilent Nexys A7, Terasic DE10.
- Logic resources: 100k–250k LUTs.
- RAM: 256 MB–1 GB DDR3.
- Frequency: 50–100 MHz realistic.
- Good for single-core HarvOS ISA testing.

13.2.2 Mid-Tier Boards

- Examples: Xilinx ZCU104, Intel Cyclone V SoC.
- Logic: 500k–1M LUTs.
- RAM: 2–4 GB DDR3/DDR4.
- Frequency: 100–200 MHz.
- Suitable for SMP (4–8 cores), with I/O peripherals.

13.2.3 High-End Boards

- Examples: Xilinx VCU118, Intel Stratix 10 GX.
- Logic: 2M+ LUTs.
- RAM: 8–16 GB DDR4.
- Frequency: 200–400 MHz.
- Capable of full cluster prototypes (16+ cores).

13.3 RTL Implementation Strategy

13.3.1 ISA Core

- In-order 5-stage pipeline (IF, ID, EX, MEM, WB).
- Optional branch predictor (2-bit saturating).
- Hardware multiplier/divider units.

13.3.2 Harvard Separation

- I-cache and D-cache physically separated.
- Instruction bus → ROM/flash/IMEM.
- Data bus → DRAM/DMEM.

13.3.3 MMU & MPU

- MMU: 2-level page tables, ASID tagging.
- MPU: 8–16 regions, lockable at boot.
- Hardware check on every fetch/load/store.

13.3.4 SMP Interconnect

- Coherent interconnect (snoop-based for <8 cores).
- Directory-based for >8 cores.
- Shared L2 cache optional.

13.4 Resource Utilization (FPGA Estimates)

Component	LUTs (Single Core)	LUTs (Quad-Core)
Core pipeline	~20k	~80k
I/D caches	~15k	~60k
MMU + TLBs	~10k	~40k
MPU	~3k	~12k
Interconnect	—	~25k
Total	~50k	~220k

14. Market Comparison & Positioning

14.1 Why Compare?

HarvOS exists in a market dominated by three paradigms:

- **x86/AMD64 with Linux/Windows**: high performance, general purpose, legacy baggage.
- **ARM Cortex-A with Linux/Android**: efficient, mobile/embedded focus, mixed security guarantees.
- **RISC-V (RV64GC) with Linux/Bare Metal**: open ISA, growing ecosystem, but young.
- **AVR/PIC-class MCUs**: small, simple, deterministic, but no MMU/security features.

To understand where HarvOS fits, we need to contrast **performance, security, complexity, and cost**.

14.2 Comparison Dimensions

1. **Performance (raw throughput, GHz, DMIPS).**
2. **Security model (hardware guarantees vs mitigations).**
3. **Complexity (LOC, attack surface).**
4. **Cost (chip price, dev effort).**
5. **Ecosystem maturity (tools, OS, drivers).**

14.3 HarvOS vs x86/AMD64

Aspect	x86/AMD64 + Linux	HarvOS
Performance	Very high (3–5 GHz, OoO, SIMD).	Moderate (1–2 GHz ASIC).
Security	Reactive mitigations (Spectre, Meltdown patches).	Proactive invariants (W^X, MPU lock, no speculation).
Complexity	Extremely high (Linux: >25M LOC).	Low (HarvOS kernel: ~50k LOC).
Cost	Commodity hardware cheap, but high power.	Custom ASIC moderate, FPGA prototyping feasible.
Ecosystem	Mature (drivers, apps, toolchains).	Immature, but reproducible + secure.

14.4 HarvOS vs ARM Cortex-A

Aspect	ARM Cortex-A + Linux	HarvOS
Performance	High (1–3 GHz, OoO, SIMD).	Moderate (1–2 GHz ASIC).
Security	TrustZone, MPU (optional), ASLR.	Hardware-rooted W^X, immutable MPU lock.
Complexity	Linux kernel & drivers huge.	Microkernel, capability-based.

Aspect	ARM Cortex-A + Linux	HarvOS
Cost	ARM licensing + per-unit cost.	Open ISA, custom hardware.
Ecosystem	Huge ecosystem, but mixed security.	Minimal, but security-by-default.

14.5 HarvOS vs RISC-V

Aspect	RISC-V (RV64GC) + Linux	HarvOS
Performance	Scalable, some cores >2 GHz.	Similar at same node, simpler design.
Security	Dependent on software (Linux), no mandatory W^X.	W^X, MPU lock, paging enforced in ISA.
Complexity	Open ISA, many extensions.	Tight, fixed feature set.
Cost	Cheap to design, open source IP.	Same advantage, but stricter constraints.
Ecosystem	Growing rapidly.	Niche, purpose-built.

14.6 HarvOS vs AVR / MCUs

Aspect	AVR / PIC MCUs	HarvOS
Performance	Very low (8–16 MHz).	1–2 GHz ASIC.
Security	None (no MMU, MPU optional).	Full MMU + MPU + NX.
Complexity	Simple, 10–20k LOC runtime.	Simple kernel, richer MMU.
Cost	Very cheap (cents per chip).	Higher ($$ for ASIC, $$$ for FPGA).
Ecosystem	Massive (Arduino, etc.).	New, security-focused niche.

14.7 Strategic Niche

HarvOS positions itself not as a competitor to x86/ARM for **raw throughput**, but as a **secure microserver platform** where:

- Determinism > Performance.
- Provable invariants > Ecosystem breadth.
- Trustworthiness > Legacy compatibility.

Ideal niches:

- **Industrial IoT gateways** (security-critical).
- **Edge servers** (DNS, NTP, syslog, VPN).
- **Regulated domains** (finance, defense, healthcare).

14.8 Market Pitch

- **For Enterprises**: “No Spectre, no Meltdown, no guesswork.”
- **For Developers**: “A clean ISA, formally checkable.”
- **For Regulators**: “Invariants enforceable in silicon.”
- **For Edge Deployments**: “Deterministic and verifiable microservers.”

14.9 Risks & Challenges

- Competing against entrenched ecosystems.
- Performance disadvantage in high-throughput applications.
- Need for compiler/runtime support maturity.
- Perception as “academic” unless strong industry partners adopt.

14.10 Summary

HarvOS does **not** attempt to dethrone x86 or ARM in the datacenter. Instead, it fills a **new category**: the **secure-by-construction microserver**. Compared to AVR, it scales; compared to RISC-V, it enforces stronger invariants; compared to ARM/x86, it avoids decades of legacy baggage.

It is **not the fastest system**, but it may be the **most trustworthy**.

15. Future Research Directions

15.1 Motivation

HarvOS was conceived as a **security-first architecture**. But no system can remain static: attackers adapt, workloads evolve, and new paradigms emerge. To stay relevant and secure, HarvOS must anticipate future demands.

We divide research directions into **architectural extensions**, **tooling improvements**, and **ecosystem integration**.

15.2 Architectural Extensions

15.2.1 Quantum Random Number Generators (QRNGs)

- True entropy sources integrated as **first-class ISA-visible peripherals**.
- Syscall: `qrng_read(buf, len)` → fills buffer with quantum randomness.

- Eliminates dependence on weak entropy pools or timer jitter.

15.2.2 Enclaves & Trusted Execution

- Hardware-backed enclaves akin to Intel SGX/ARM TrustZone.
- Difference: enclaves **cannot override Harvard split**.
- Example: TLS key enclave → only decrypts in place, never exposes raw private key.

15.2.3 ISA Security Extensions

- New instructions for constant-time crypto (`aes.enc`, `sha256.step`).
- Memory-safe load/store variants with bounds-check hardware.
- Speculative execution entirely disabled → a security stance in itself.

15.2.4 Deterministic Timing Channels

- Eliminate microarchitectural side-channels:
  - Fixed-cycle memory accesses.
  - Time-partitioned caches.
- Allows **real-time + secure** coexistence.

15.3 Tooling Improvements

15.3.1 Formal Verification of Compiler

- Extend CompCert (formally verified C compiler) with HarvOS backend.
- Guarantee that compiled code preserves semantics *and* Harvard separation.

15.3.2 Verified Toolchain

- LLVM backend with SVA-embedded proofs.
- Each binary certified to obey W^X, NX, and MPU constraints.

15.3.3 Symbolic Execution for Security

- Tool to symbolically execute binaries under HarvOS rules.
- Detects potential privilege escalations or policy violations before deployment.

15.4 Ecosystem & Application Areas

15.4.1 Edge AI

- Tensor inference on secure microservers.

- Extension: simple SIMD or matrix-mul unit.
- Constraint: no unbounded DMA; memory-safety enforced.

15.4.2 Blockchain / Cryptography

- Nodes running HarvOS = inherently hardened.
- Crypto opcodes accelerate signature validation.
- Reduces attack vectors in financial consensus.

15.4.3 Regulated Industries

- **Healthcare**: devices provably cannot exfiltrate patient data.
- **Defense**: strict policy invariants.
- **Aviation**: deterministic execution for certification.

15.5 Formal Methods Integration

15.5.1 Proof-Carrying Code

- Developers ship binaries with machine-checkable proofs.
- OS loads binary only if proof validates:
    - No buffer overflows.
    - Only permitted syscalls.
    - No RWX mappings.

15.5.2 System Invariants

- Prove theorems like:
    - ∀ *instruction, if fetch = data-space → trap.*
    - ∀ *mapping, if page.exec = true → page.write = false.*

15.5.3 Model Checking

- Use SPIN/Coq/Isabelle to validate HarvOS kernel against formal specs.

15.6 Long-Term Vision

HarvOS could become:

- The **reference platform** for teaching secure computer architecture.
- A **testbed** for research in formal methods, policy models, and secure OS design.

- A **commercial niche product** in environments where trust > throughput.

Imagine:

- A **HarvOS-based microserver cluster** running DNS, NTP, syslog, VPN — **provably isolated**, immune to ROP attacks, with runtime proofs of policy enforcement.
- Certification bodies adopt HarvOS as a **gold standard**: "if it runs on HarvOS, it runs safely."

15.7 Challenges

- Performance gap to mainstream CPUs.
- Immature ecosystem → must rely on early adopters.
- Need for formal verification expertise → expensive talent.
- Risk of being seen as "academic only."

15.8 Summary

The **future of HarvOS** lies not in chasing GHz wars, but in **pioneering provability**.
Every extension — QRNGs, enclaves, formal toolchains — strengthens its central mission:
to be the operating system and architecture where trust is built into silicon itself.

16. Conclusion & Outlook

16.1 Recap of the Journey

This whitepaper introduced **HarvOS**, a **Harvard-architecture CPU and operating system** that integrates modern memory safety primitives (MMU, MPU, NX, W^X) into a **unified, minimal, and enforceable model**.

- We began by revisiting **Harvard vs. von Neumann** models, exposing the security potential of physical separation.
- We designed a **32-bit clean ISA** with full MMU support, while retaining deterministic behavior and bounded complexity.
- We demonstrated **heap and stack overflow mitigations**, achieved not through patches, but through **architectural invariants**.
- We added **policies**, declarative and immutable, to enforce fine-grained capability control.
- We validated feasibility with **FPGA prototyping**, and projected the transition to ASIC.
- We compared HarvOS against existing platforms, showing its niche: **secure microservers where trust > throughput**.

16.2 Key Contributions

- **Security by construction**: not dependent on reactive mitigations.
- **Minimal kernel**: small attack surface, ~50k LOC target.
- **Policy immutability**: no runtime policy escalation possible.
- **Hardware/software co-design**: ISA, kernel, and policy model evolve together.
- **Deterministic execution**: essential for real-time and regulated environments.

16.3 Lessons Learned

- **Redundancy can be strength**: combining MMU + MPU is not wasteful, but complementary.
- **Physical separation matters**: Harvard split enforces NX/W^X at the physics level, not just software.
- **Small is beautiful**: a constrained ISA and kernel reduce attack vectors dramatically.
- **FPGA prototypes are practical**: even modest boards validate the concept before ASIC.

16.4 Limitations

- **Performance gap** to mainstream CPUs.
- **Immature ecosystem** compared to x86/ARM.
- **Higher per-unit costs** until ASIC production scales.
- **Adoption barrier**: requires convincing users to trade compatibility for trust.

16.5 Outlook

HarvOS should be seen as the **foundation of a new security paradigm**:

- Instead of **patching around vulnerabilities** (Spectre, Meltdown, Rowhammer), design a system that is simply **immune** by construction.
- Instead of bloated general-purpose OS kernels, embrace a **minimalist, capability-driven microkernel**.
- Instead of relying on **obscure policies**, enforce clear invariants at the ISA level.

Looking ahead:

- Integration of **quantum RNGs** ensures entropy quality.

- Formal verification of kernel and compiler ensures **end-to-end correctness**.
- ASIC fabrication could make HarvOS available as a **trusted hardware platform** for critical infrastructure.

16.6 Final Vision

The future belongs not to the **fastest architecture**, but to the **most trustworthy**.
In an age of ransomware, supply-chain attacks, and pervasive surveillance, **trust** is the true performance metric.

HarvOS represents a step toward that future:

- **An OS carved in stone** — unyielding, transparent, and provable.
- A platform where **security is not a feature**, but the **default state**.
- A microserver design that trades raw power for **verifiable assurance**.

In this sense, HarvOS is not merely an operating system or a CPU — it is a **statement**:
that we can and should build systems **where compromise is not an option**.

HarvOS Whitepaper – Security Amendments (2025 Review)

This appendix adds clarifications and mandatory requirements identified in the 2025 security review.

1) DMA/IOMMU:
- All DMA masters MUST traverse a bus firewall/IOMMU.
- DMA writes to I-space or kernel space are forbidden by hardware.

2) Code Storage Immutability:
- After boot, flash/XIP regions mapped as I-space are locked read-only in hardware.
- MPU and IOMMU deny writes to code-store regions post-boot.
- Updates require A/B image swap and reboot, with signature verification.

3) Side-channel Mitigations:
- Per-ASID way-partitioning or cache coloring is REQUIRED in multi-tenant contexts.
- Context-switch flushing of shared caches is enforced.

4) Harvard Alias Invariant:
- It is forbidden to map the same PFN writable in D-space and executable in I-space simultaneously.
- A formal proof obligation enforces this property.

5) TRNG/DRBG Hardening:
- TRNG output is conditioned, with health tests (SP 800-90B).
- Entropy API blocks until seeded; never fails open.
- TRNG registers are privileged-only and not DMA-accessible.

These amendments strengthen the invariants already stated in the whitepaper without altering the existing text.