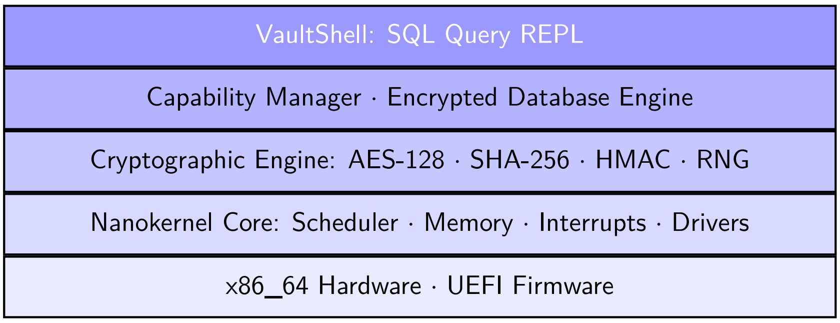
VaultOS

A Capability-Secured Nanokernel with
Database-Centric Architecture

Design, Algorithms, and Implementation


VaultShell: SQL Query REPL
Capability Manager · Encrypted Database Engine
Cryptographic Engine: AES-128 · SHA-256 · HMAC · RNG
Nanokernel Core: Scheduler · Memory · Interrupts · Drivers
x86_64 Hardware · UEFI Firmware

February

2026

Contents

Preface

VaultOS is a nanokernel operating system built on two radical premises:

1. **Everything is a database.** There is no file system, no `/proc`, no device files. Every system resource—processes, capabilities, messages, user data—is a row in an encrypted relational table, accessible only through a SQL-subset query language.

2. **All data is confidential.** Every database record is encrypted with AES-128-CBC using per-table keys derived from a master secret via HMAC-SHA256. Every access requires a cryptographically sealed capability token that proves authorization.

VaultOS deliberately rejects POSIX. There are no file descriptors, no `fork()`, no signals, no pipes, and no user/group permission bits. Instead, the system provides a small, formally analyzable interface: database queries mediated by unforgeable capability tokens.

Conventions. Algorithms are presented in the pseudocode style of Cormen, Leiserson, Rivest, and Stein (CLRS). We use $\mathcal{O}$-notation for asymptotic analysis. Diagrams use TikZ. Source code excerpts are in C or x86-64 assembly.

Part I

Foundations

Chapter 1

Introduction and Design Philosophy

1.1 The Database-as-OS Paradigm

Traditional operating systems expose heterogeneous interfaces: a file system for persistent storage, `/proc` for process metadata, signals for asynchronous notification, and sockets for communication. Each subsystem has its own access-control model, its own naming scheme, and its own failure modes.

VaultOS replaces this patchwork with a single abstraction: the *encrypted relational table*. Six system tables (Table 1.1) store all kernel state. User interaction proceeds entirely through a SQL-subset query language executed by the VaultShell REPL.

Table 1.1: VaultOS system tables.

ID	Table	Encrypted	Purpose
0	`SystemTable`	Yes	Boot metadata (OS name, version)
1	`ProcessTable`	Yes	Active processes (PID, state, capabilities)
2	`CapabilityTable`	Yes	HMAC-sealed access tokens
3	`ObjectTable`	Yes	User-defined objects (data blobs)
4	`MessageTable`	Yes	IPC message queue
5	`AuditTable`	Yes	Security audit log

Definition 1.1 (Database-as-OS). An operating system in which every named resource r has a canonical representation as a tuple $\langle \mathrm{id}, \mathrm{col}_1, \ldots, \mathrm{col}_k \rangle$ in a table T, and every operation on r is expressed as a query $Q \in \{\textsc{Select}, \textsc{Insert}, \textsc{Update}, \textsc{Delete}\}$ over T.

1.2 Capability-Based Security Model

Access control in VaultOS is based on *capabilities*: unforgeable tokens that encode a subject's rights over an object.

Definition 1.2 (Capability)**.** A capability is a tuple

$$c = (\text{cap\_id},\ \text{obj\_id},\ \text{type},\ \text{owner},\ \text{rights},\ \text{parent},\ \sigma)$$

where $\sigma = \text{HMAC-SHA256}(K_{\text{master}},\ \text{cap\_id}\|\text{obj\_id}\|\text{owner}\|\text{rights}\|\text{type}\|\text{parent})$ is a 256-bit cryptographic seal.

Rights are encoded as a 6-bit mask:

Bit	Right	Value
0	Read	2^0
1	Write	2^1
2	Execute	2^2
3	Delete	2^3
4	Grant	2^4
5	Revoke	2^5

Theorem 1.3 (Capability Unforgeability)**.** *An adversary who does not know the master key K_{master} cannot produce a valid capability c' (one whose HMAC verifies) except with probability negligible in $|K_{\text{master}}|$, assuming HMAC-SHA256 is a secure PRF.*

Proof. If an adversary could forge a capability c' with non-negligible probability, they could construct a distinguisher for HMAC-SHA256 as a PRF, contradicting the assumed security of the construction (RFC 2104). The 40-byte input domain is fixed, so no length-extension attacks apply. □

1.3 Threat Model

VaultOS protects against:

- **Unauthorized data access:** All records encrypted; queries require valid capabilities.
- **Capability forgery:** HMAC-SHA256 seal prevents fabrication.
- **Timing side-channels:** Constant-time HMAC comparison prevents timing attacks.
- **Privilege escalation:** Delegated capabilities are strict subsets of parent rights.

Out of scope (MVP): cold-boot attacks, hardware Trojans, DMA attacks, Spectre/Meltdown.

1.4 System Architecture Overview

Figure 1.1 shows the layered architecture.

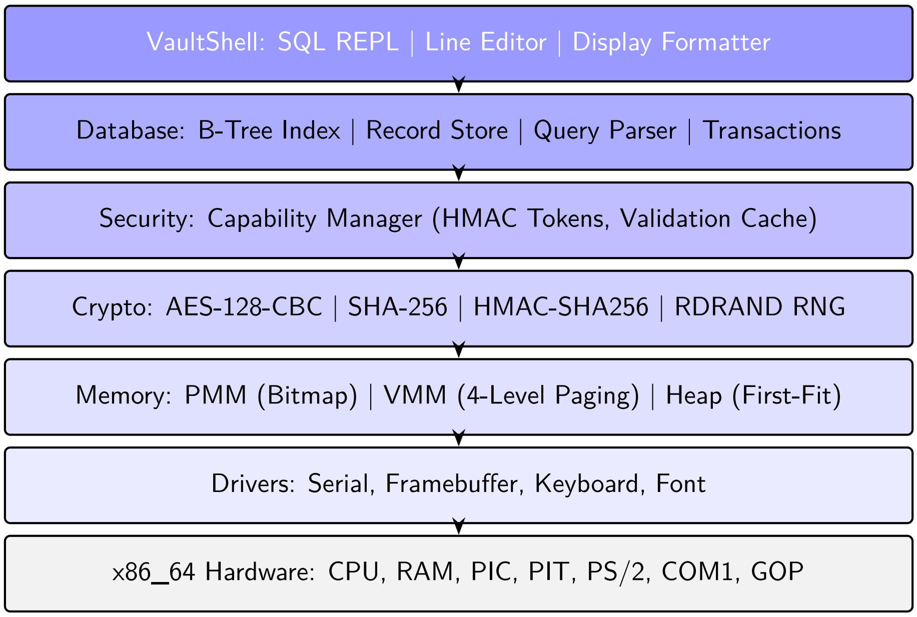

VaultShell: SQL REPL | Line Editor | Display Formatter
Database: B-Tree Index | Record Store | Query Parser | Transactions
Security: Capability Manager (HMAC Tokens, Validation Cache)
Crypto: AES-128-CBC | SHA-256 | HMAC-SHA256 | RDRAND RNG
Memory: PMM (Bitmap) | VMM (4-Level Paging) | Heap (First-Fit)
Drivers: Serial, Framebuffer, Keyboard, Font
x86_64 Hardware: CPU, RAM, PIC, PIT, PS/2, COM1, GOP

Figure 1.1: VaultOS layered architecture. Each layer depends only on the layers below it.

1.5 Comparison with Traditional OS Designs

Table 1.2: VaultOS vs. POSIX-based operating systems.

Aspect	**POSIX**	**VaultOS**
Resource abstraction	Files, sockets, pipes	Database rows
Access control	uid/gid, rwx bits	HMAC-sealed capabilities
Naming	Hierarchical paths	Table + row ID
IPC	Pipes, signals, shmem	Message table + queries
Process interface	`fork/exec/wait`	`proc_create/exit`
Encryption	Optional (dm-crypt, etc.)	Mandatory, per-table
Audit	Syslog (optional)	Built-in AuditTable
User interface	Shell + utilities	SQL REPL

Chapter 2

System Bootstrap

2.1 UEFI Boot Protocol

The bootloader is a PE32+ EFI application compiled against GNU-EFI. It executes the following steps:

1. Initialize the Graphics Output Protocol (GOP) at 1024 × 768 resolution.
2. Open the FAT32 system partition and load `VAULTOS.BIN` at physical address `0x100000` (1 MiB).
3. Acquire the UEFI memory map (with retry around `ExitBootServices`).
4. Locate the ACPI RSDP from the EFI configuration tables.
5. Transfer control to `kernel_main(BootInfo *)` at the kernel physical base.

2.2 BootInfo Structure

The bootloader passes a packed structure containing framebuffer parameters, the UEFI memory map, kernel location, and runtime service pointers:

Listing 2.1: BootInfo structure (simplified).

```
typedef struct __attribute__((packed)) {
    uint64_t fb_base, fb_width, fb_height, fb_pitch;
    uint32_t fb_bpp, fb_pixel_format;
    uint64_t mmap_base, mmap_size, mmap_desc_size;
    uint32_t mmap_entry_count;
    uint64_t kernel_phys_base, kernel_size;
    uint64_t rsdp_address;
} BootInfo;
```

2.3 Kernel Initialization Sequence

The kernel initializes ten subsystems in strict dependency order (Figure 2.1).

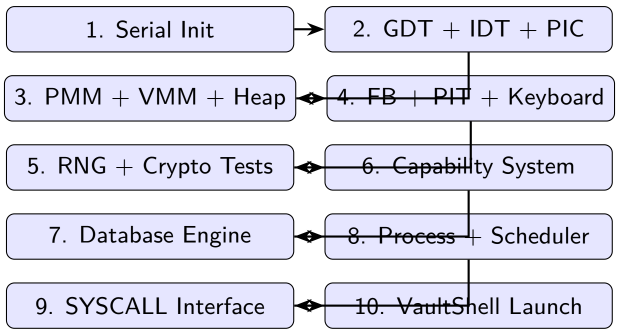


Figure 2.1: Kernel initialization sequence. Each phase depends on all preceding phases.

2.4 Higher-Half Kernel Design

The kernel is linked at virtual address `0xFFFFFFFF80000000` but loaded at physical address `0x100000`. This *higher-half* design reserves the lower 128 TiB of virtual space for user processes.

Definition 2.1 (Virtual Address Space Layout)**.** The 48-bit canonical virtual address space is partitioned as follows:

`0x0000000000000000`–`0x00007FFFFFFFFFFF`	User space (128 TiB)
`0xFFFFFFFF80000000`–`0xFFFFFFFF81FFFFFF`	Kernel code/data (2 MiB)
`0xFFFFFFFF82000000`–`0xFFFFFFFF91FFFFFF`	Kernel heap (256 MiB)
`0xFFFFFFFF92000000`–`0xFFFFFFFFBFFFFFFF`	Physical direct map (up to 1 GiB)
`0xFFFFFFFFC0000000`–`0xFFFFFFFFCFFFFFFF`	Framebuffer

Part II

Memory Management

Chapter 3

Physical Memory Manager

The physical memory manager (PMM) tracks the availability of 4 KiB physical pages using a bitmap data structure.

3.1 Bitmap Allocator Design

Definition 3.1 (Page Bitmap)**.** Let $N = \lfloor M/4096 \rfloor$ be the number of physical pages, where M is the total physical memory in bytes. The bitmap $B[0 \ldots \lceil N/64 \rceil - 1]$ is an array of 64-bit words where bit i of word $B[\lfloor i/64 \rfloor]$ is 1 if page i is allocated, and 0 if free.

For 4 GiB of RAM, $N = 2^{20}$ pages and the bitmap occupies $2^{20}/8 = 128$ KiB—a fixed overhead of 0.003%.

3.2 Algorithm: Pmm-Alloc

Algorithm 1: Pmm-Alloc(): Allocate one physical page.

```
1 for $w \leftarrow 0$ to $\lceil N/64 \rceil - 1$ do
2     if $B[w] \neq$ 0xFFFFFFFFFFFFFFFF then
3         $b \leftarrow$ index of lowest clear bit in $B[w]$
            // __builtin_ctzll(~B[w])
4         page $\leftarrow 64w + b$
5         if $page < N$ then
6             set bit $b$ in $B[w]$
7             used_pages $\leftarrow$ used_pages $+ 1$
8             return page $\times$ 4096                // physical address
9 return 0                                            // out of memory
```

3.3 Algorithm: Pmm-Free

Algorithm 2: Pmm-Free(phys_addr): Free one physical page.

```
1 page ← phys_addr/4096
2 if $page \geq N$ or $bit$ ($page$ mod 64) $of$ $B[\lfloor page/64 \rfloor]$ $is$ $clear$ then
3     error "double free or invalid page"
4 clear bit (page mod 64) in $B[\lfloor \text{page}/64 \rfloor]$
5 used_pages ← used_pages − 1
```

3.4 Complexity Analysis

Theorem 3.2 (PMM Allocation Complexity)**.** *Pmm-Alloc runs in $\mathcal{O}(N/64)$ worst-case time and $\mathcal{O}(1)$ best-case time, where N is the number of physical pages.*

Proof. The outer loop iterates over at most $\lceil N/64 \rceil$ words. Each iteration performs a constant-time comparison and bit scan. In the best case, the first word contains a free bit. In the worst case, all words must be scanned. The per-word bit scan (`ctzll`) executes in $\mathcal{O}(1)$ on x86-64 via the `BSF`/`TZCNT` instruction. □

Chapter 4

Virtual Memory and Paging

4.1 x86-64 Four-Level Page Tables

The x86-64 architecture uses a four-level radix tree to translate 48-bit virtual addresses to physical addresses.

Definition 4.1 (Page Table Entry)**.** A 64-bit page table entry (PTE) encodes:

$$\text{PTE} = \underbrace{\text{phys\_addr}[51:12]}_{\text{40 bits}} \parallel \underbrace{\text{flags}[11:0]}_{\text{12 bits}}$$

where flags include Present (P), Writable (W), User (U), and No-Execute (NX, bit 63).

Definition 4.2 (Virtual Address Decomposition)**.** A 48-bit virtual address v is decomposed as:

$$\begin{aligned}
\text{PML4 index} &= (v \gg 39) \mathbin{\&} \texttt{0x1FF} \\
\text{PDPT index} &= (v \gg 30) \mathbin{\&} \texttt{0x1FF} \\
\text{PD index} &= (v \gg 21) \mathbin{\&} \texttt{0x1FF} \\
\text{PT index} &= (v \gg 12) \mathbin{\&} \texttt{0x1FF} \\
\text{Offset} &= v \mathbin{\&} \texttt{0xFFF}
\end{aligned}$$

4.2 Virtual Address Space Layout

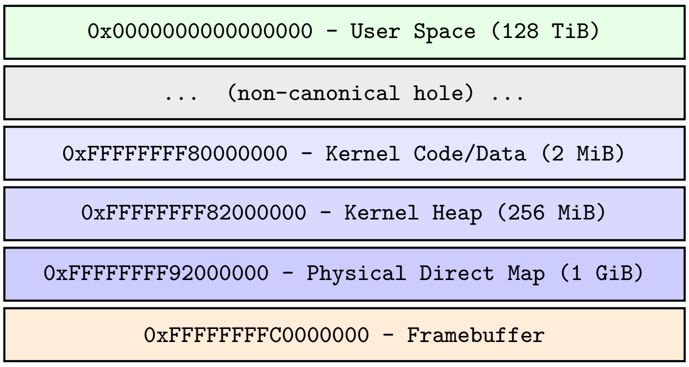

0x0000000000000000 - User Space (128 TiB)
... (non-canonical hole) ...
0xFFFFFFFF80000000 - Kernel Code/Data (2 MiB)
0xFFFFFFFF82000000 - Kernel Heap (256 MiB)
0xFFFFFFFF92000000 - Physical Direct Map (1 GiB)
0xFFFFFFFFC0000000 - Framebuffer

Figure 4.1: Virtual address space layout of the VaultOS kernel.

4.3 Algorithm: Page-Map

Algorithm 3: PAGE-MAP(pml4, virt, phys, flags): Map a virtual page to a physical frame.

```
pml4e ← pml4[PML4_INDEX(virt)]
if pml4e is not present then
    allocate a zeroed page for PDPT
    pml4[PML4_INDEX(virt)] ← pdpt_phys | P | W
pdpt ← extract address from pml4e
// Repeat for PDPT → PD → PT
// (Each level: check present, allocate if needed, descend)
pt[PT_INDEX(virt)] ← phys | flags
```

4.4 Algorithm: Virt-To-Phys

Algorithm 4: VIRT-TO-PHYS(pml4, virt): Translate virtual to physical address.

```
Walk PML4 → PDPT → PD → PT using index functions
if any level is not present then
    return 0                                  // page not mapped
if PD entry has Huge flag (2 MiB page) then
    return pd_phys | (virt & 0x1FFFFF)
return pt_phys | (virt & 0xFFF)
```

Theorem 4.3 (Translation Correctness)**.** *For any virtual address v mapped via* PAGE-MAP *to physical address p with flags f, a subsequent call to* VIRT-TO-PHYS *returns p, provided no intervening unmap or CR3 reload invalidates the* *mapping.*

Chapter 5

Kernel Heap Allocator

5.1 First-Fit with Coalescing

The kernel heap is a doubly-linked list of blocks, each preceded by a header:

Listing 5.1: Heap block header.

```c
typedef struct heap_block {
    uint64_t      magic;     /* 0xDEADBEEF */
    size_t        size;      /* usable data bytes */
    bool          free;
    heap_block_t *next, *prev;
} heap_block_t;
```

5.2 Algorithm: Kmalloc

Algorithm 5: KMALLOC(n): Allocate n bytes from the kernel heap.

```
1  $n \leftarrow$ ALIGN-UP$(n, 16)$
2  $b \leftarrow$ head of block list
3  while $b \neq$ nil do
4      if $b.free$ and $b.size \geq n$ then
5          if $b.size - n > sizeof(header) + 32$ then
6              SPLIT($b$, $n$)          // create free block from remainder
7          $b.free \leftarrow$ false
8          return pointer to $b$'s data area
9      $b \leftarrow b.next$
10 return nil                                  // out of memory
```

5.3 Algorithm: Kfree with Coalescing

Algorithm 6: KFREE(p): Free a previously allocated block and coalesce neighbors.

```
1  $b \leftarrow$ block header preceding $p$
2  assert $b.magic =$ 0xDEADBEEF
3  $b.free \leftarrow$ true
   // Forward coalescing
4  if $b.next \neq$ nil and $b.next.free$ then
5      $b.size \leftarrow b.size +$ sizeof(header) $+ b.next.size$
6      $b.next \leftarrow b.next.next$
7      if $b.next \neq$ nil then
8          $b.next.prev \leftarrow b$
   // Backward coalescing
9  if $b.prev \neq$ nil and $b.prev.free$ then
10     $b.prev.size \leftarrow b.prev.size +$ sizeof(header) $+ b.size$
11     $b.prev.next \leftarrow b.next$
12     if $b.next \neq$ nil then
13         $b.next.prev \leftarrow b.prev$
```

Theorem 5.1 (Heap Invariant). *After any sequence of* KMALLOC *and* KFREE *operations, no two adjacent blocks in the free list are both free.*

Proof. KFREE explicitly coalesces with both the predecessor and successor blocks. If either neighbor is free, the blocks are merged. Therefore, upon return from KFREE, the freed block has no free neighbor, maintaining the invariant. KMALLOC can only split a free block into (allocated, free), which cannot create adjacent free blocks. □

Part III

Cryptographic Primitives

Chapter 6

SHA-256

SHA-256 is the foundation of VaultOS's integrity guarantees. It is used in HMAC for capability sealing and in key derivation for per-table encryption.

6.1 Merkle-Damgård Construction

SHA-256 follows the Merkle-Damgård paradigm: the message is padded to a multiple of 512 bits, then processed in 512-bit (64-byte) blocks. Each block is compressed into the running 256-bit state.

Definition 6.1 (SHA-256 State)**.** The state consists of eight 32-bit words $H_0, \ldots, H_7$ initialized to the fractional parts of the square roots of the first eight primes:

$$H_0 = \texttt{6a09e667}, \quad H_1 = \texttt{bb67ae85}, \quad \ldots, \quad H_7 = \texttt{5be0cd19}$$

6.2 Compression Function

Algorithm 7: SHA256-TRANSFORM($H[0..7]$, block$[0..63]$): Process one 512-bit block.

```
   // Message schedule
1  for $i \leftarrow 0$ to 15 do
2      $W[i] \leftarrow$ big-endian 32-bit word from block$[4i \ldots 4i{+}3]$
3  for $i \leftarrow 16$ to 63 do
4      $W[i] \leftarrow \sigma_1(W[i-2]) + W[i-7] + \sigma_0(W[i-15]) + W[i-16]$
   // Initialize working variables
5  $a, b, c, d, e, f, g, h \leftarrow H[0], H[1], \ldots, H[7]$
   // Compression rounds
6  for $i \leftarrow 0$ to 63 do
7      $T_1 \leftarrow h + \Sigma_1(e) + \mathrm{Ch}(e, f, g) + K_i + W[i]$
8      $T_2 \leftarrow \Sigma_0(a) + \mathrm{Maj}(a, b, c)$
9      $h \leftarrow g;\ g \leftarrow f;\ f \leftarrow e;\ e \leftarrow d + T_1$
10     $d \leftarrow c;\ c \leftarrow b;\ b \leftarrow a;\ a \leftarrow T_1 + T_2$
   // Update state
11 $H[i] \leftarrow H[i] + \{a, b, c, d, e, f, g, h\}_i$ for $i = 0, \ldots, 7$
```

The mixing functions are defined as:

$$\begin{aligned}
\mathrm{Ch}(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
\mathrm{Maj}(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= \mathrm{ROTR}^2(x) \oplus \mathrm{ROTR}^{13}(x) \oplus \mathrm{ROTR}^{22}(x) \\
\Sigma_1(x) &= \mathrm{ROTR}^6(x) \oplus \mathrm{ROTR}^{11}(x) \oplus \mathrm{ROTR}^{25}(x) \\
\sigma_0(x) &= \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus (x \gg 3) \\
\sigma_1(x) &= \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus (x \gg 10)
\end{aligned}$$

6.3 Block-Based Update

Algorithm 8: SHA256-Update(ctx, data, len): Incrementally feed data to the hash function.

```
1  buffered ← ctx.count mod 64
2  ctx.count ← ctx.count + len
3  if buffered > 0 then
4      need ← 64 − buffered
5      if len < need then
6          copy data to buffer at offset buffered
7          return
8      copy need bytes to complete buffer
9      SHA256-Transform(ctx.state, ctx.buffer)
10     advance data by need;
11     len ← len − need
12 while len ≥ 64 do
13     SHA256-Transform(ctx.state, data)        // process directly
14     advance data by 64;
15     len ← len − 64
16 if len > 0 then
17     copy remaining len bytes to ctx.buffer
```

Observation 6.2. The block-based update processes aligned 64-byte blocks directly from the input buffer, avoiding 64× per-byte overhead compared to a naïve byte-at-a-time approach. For HMAC computation of a 64-byte ipad, this processes the entire block in a single call to SHA256-Transform.

Chapter 7

AES-128

AES-128 provides confidentiality for all database records. VaultOS implements both a software path with precomputed lookup tables and a hardware-accelerated path using AES-NI instructions.

7.1 Rijndael Cipher Structure

AES-128 operates on 128-bit (16-byte) blocks using a 128-bit key, performing 10 rounds of transformations on a 4×4 byte matrix called the *state*.

Definition 7.1 (AES Round)**.** Each round (except the last) applies four transformations:

$$\text{Round}(s, k_r) = \textsc{AddRoundKey}(\textsc{MixColumns}(\textsc{ShiftRows}(\textsc{SubBytes}(s))), k_r)$$

The final round omits MixColumns.

7.2 GF(2^8) Arithmetic and Precomputed Tables

The MixColumns step requires multiplication in GF(2^8) with the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ (= `0x11B`).

Rather than computing these multiplications at runtime (8 iterations per byte), VaultOS uses six precomputed 256-byte lookup tables:

Table	Usage
mul2[256]	MixColumns: $\{2\} \cdot x$
mul3[256]	MixColumns: $\{3\} \cdot x$
mul9[256]	InvMixColumns: $\{9\} \cdot x$
mul11[256]	InvMixColumns: $\{b\} \cdot x$
mul13[256]	InvMixColumns: $\{d\} \cdot x$
mul14[256]	InvMixColumns: $\{e\} \cdot x$

This replaces $48 \times 8 = 384$ loop iterations per block with 48 table lookups.

7.3 Algorithm: AES-Key-Expand

Algorithm 9: AES-KEY-EXPAND(key[0..15]): Expand 128-bit key to 11 round keys.

```
w[0..3] ← 32-bit words from key
for $i$ ← 4 to 43 do
    temp ← $w[i-1]$
    if $i \bmod 4 = 0$ then
        temp ← SubWord(RotWord(temp)) ⊕ Rcon[$i/4$]
    $w[i] \leftarrow w[i-4] \oplus$ temp
```

7.4 Algorithm: AES-Encrypt-Block

Algorithm 10: AES-ENCRYPT-BLOCK(ctx, block[0..15]): Encrypt one 128-bit block.

```
AddRoundKey(block, ctx.rk[0])
for $r$ ← 1 to 9 do
    SubBytes(block)                        // S-box lookup, 16 bytes
    ShiftRows(block)                       // cyclic row rotations
    MixColumns(block)                      // GF($2^8$) via lookup tables
    AddRoundKey(block, ctx.rk[$r$])
SubBytes(block)
ShiftRows(block)
AddRoundKey(block, ctx.rk[10])
```

7.5 CBC Mode

Algorithm 11: AES-CBC-ENCRYPT(ctx, iv, pt, ct, len)

```
prev ← iv
for $off$ ← 0 to $len - 16$ step 16 do
    for $i$ ← 0 to 15 do
        ct[off + $i$] ← pt[off + $i$] ⊕ prev[$i$]
    AES-Encrypt-Block(ctx, ct + off)
    prev ← ct + off                        // pointer, no copy
```

Observation 7.2. The CBC encrypt implementation avoids redundant `memcpy` by maintaining a pointer to the previous ciphertext block rather than copying it to a temporary buffer. This eliminates 2 of 3 `memcpy` calls per block.

7.6 AES-NI Hardware Acceleration

When the CPU supports AES-NI (detected via `CPUID.01H:ECX[25]`), VaultOS dispatches to a hardware-accelerated path:

Listing 7.1: AES-NI encrypt (inline assembly sketch).

```
movdqu   (%block), %xmm0        ; Load plaintext block
pxor     0(%rk),   %xmm0        ; AddRoundKey 0
aesenc   16(%rk),  %xmm0        ; Rounds 1-9 (9 instructions
    )
aesenc   32(%rk),  %xmm0
; ... (rounds 3-9) ...
aesenclast 160(%rk), %xmm0      ; Final round
movdqu   %xmm0,    (%block)     ; Store ciphertext
```

For decryption, each intermediate round key must be transformed via `AESIMC` (InvMixColumns) before use with `AESDEC`.

Table 7.1: AES-128-CBC performance: 1 KiB × 100 iterations.

Operation	**AES-NI (Haswell)**	**Software (Nehalem)**	**Speedup**
Encrypt	29,388 cycles/op	217,929 cycles/op	7.4×
Decrypt	39,023 cycles/op	284,357 cycles/op	7.3×

Theorem 7.3 (CBC IND-CPA Security)**.** *AES-128-CBC with random IVs is IND-CPA secure (indistinguishable under chosen-plaintext attack) assuming AES is a pseudorandom permutation (PRP), up to* 2^{64} *blocks (the birthday bound on 128-bit blocks).*

Chapter 8

HMAC-SHA256 and Random Number Generation

8.1 HMAC Construction

HMAC-SHA256 (RFC 2104) provides message authentication:

$$\text{HMAC}(K, m) = \text{SHA256}((K \oplus \text{opad}) \parallel \text{SHA256}((K \oplus \text{ipad}) \parallel m))$$

where ipad = $\texttt{0x36}^{64}$ and opad = $\texttt{0x5C}^{64}$.

8.2 Pre-computed Context

Since the master key K is fixed at boot, VaultOS pre-computes the SHA-256 state after processing the ipad and opad blocks:

Algorithm 12: HMAC-Init(ctx, K, klen): Pre-compute HMAC state for key K.

```
1 if $klen > 64$ then
2     $K' \leftarrow \text{SHA256}(K)$;   pad to 64 bytes with zeros
3 else
4     $K' \leftarrow K$ padded to 64 bytes with zeros
5 ipad[$i$] $\leftarrow K'[i] \oplus$ 0x36 for $i = 0, \ldots, 63$
6 opad[$i$] $\leftarrow K'[i] \oplus$ 0x5C for $i = 0, \ldots, 63$
7 ctx.inner_base $\leftarrow$ SHA256-Update(SHA256-Init(), ipad, 64)
8 ctx.outer_base $\leftarrow$ SHA256-Update(SHA256-Init(), opad, 64)
```

Algorithm 13: HMAC-Compute(ctx, data, len, mac): Compute HMAC using pre-computed context.

```
1 inner ← clone(ctx.inner_base)
2 SHA256-Update(inner, data, len)
3 SHA256-Final(inner, inner_hash)
4 outer ← clone(ctx.outer_base)
5 SHA256-Update(outer, inner_hash, 32)
6 SHA256-Final(outer, mac)
```

Observation 8.1. Pre-computing the HMAC context eliminates re-hashing the 128 bytes of ipad and opad for every HMAC computation. For capability validation (40-byte payload), this reduces SHA-256 block processing from 4 blocks to 2 blocks per HMAC—a $\sim 3\times$ speedup.

8.3 Constant-Time Verification

Algorithm 14: HMAC-Verify($a[0..n-1]$, $b[0..n-1]$): Constant-time comparison.

```
1 diff ← 0
2 for $i \leftarrow 0$ to $n-1$ do
3     diff ← diff | $(a[i] \oplus b[i])$
4 return diff = 0
```

The OR-accumulation ensures the loop runs in exactly n iterations regardless of where mismatches occur, preventing timing side-channel attacks.

8.4 Random Number Generation

VaultOS seeds its entropy pool from the `RDRAND` instruction (when available) and falls back to `RDTSC`-based seeding with an xorshift128+ PRNG.

Theorem 8.2 (HMAC Unforgeability)**.** *Under the assumption that SHA-256 is a pseudorandom function (PRF) when keyed, HMAC-SHA256 is* (t, q, ϵ)*-unforgeable: no adversary running in time* t *and making* q *queries can forge a valid MAC with probability greater than* $\epsilon + q \cdot 2^{-256}$.

Part IV

Data Structures

Chapter 9

B-Tree Index

The database engine indexes every table with a B-tree of order $t = 64$, providing $\mathcal{O}(\log_{64} n)$ search, insert, and delete operations.

9.1 B-Tree Properties

Definition 9.1 (B-Tree of Order t)**.** A B-tree of order t satisfies:

1. Every node has at most $2t - 1$ keys ($= 127$ for $t = 64$).
2. Every non-root node has at least $t - 1$ keys ($= 63$).
3. A node with k keys has $k + 1$ children (if internal).
4. All leaves appear at the same depth.

In VaultOS, BTREE_ORDER $= 64$, so nodes store up to 63 keys and 64 child pointers. Each node occupies approximately $63 \times 8 + 63 \times 8 + 64 \times 8 + 8 = 1{,}520$ bytes, fitting comfortably in L1 cache.

Remark 9.2. We use order-t notation where the maximum number of keys per node is $t - 1 = 63$, consistent with the CLRS definition. Some references use t for the maximum keys; ours uses t for the minimum degree.

9.2 Node Structure

Listing 9.1: B-tree node structure.

```
typedef struct btree_node {
    uint64_t    keys[63];        /* sorted key array */
    void       *values[63];      /* associated record
        pointers */
    btree_node *children[64];    /* child pointers */
    uint32_t   num_keys;
    bool       is_leaf;
} btree_node_t;
```

9.3 Algorithm: B-Tree-Search

Algorithm 15: B-TREE-SEARCH(x, k): Search for key k in subtree rooted at x.

```
1 $i \leftarrow 0$
2 while $i < x.num\_keys$ and $k > x.keys[i]$ do
3     $i \leftarrow i + 1$
4 if $i < x.num\_keys$ and $k = x.keys[i]$ then
5     return $x.values[i]$
6 if $x.is\_leaf$ then
7     return nil
8 return B-TREE-SEARCH($x.children[i]$, $k$)
```

9.4 Algorithm: B-Tree-Insert

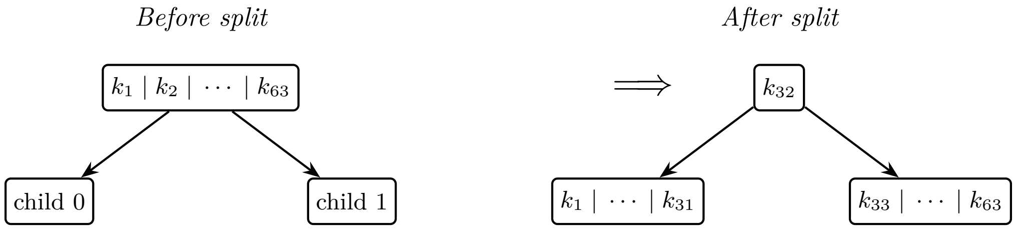
Algorithm 16: B-TREE-INSERT(T, k, v): Insert key k with value v into B-tree T.

```
1 $r \leftarrow T.root$
2 if $r.num\_keys = 63$ then
3     $s \leftarrow$ new node (leaf = false)
4     $s.children[0] \leftarrow r$
5     $T.root \leftarrow s$
6     B-TREE-SPLIT-CHILD($s$, 0)
7     B-TREE-INSERT-NONFULL($s$, $k$, $v$)
8 else
9     B-TREE-INSERT-NONFULL($r$, $k$, $v$)
```

9.5 Algorithm: B-Tree-Split-Child

Algorithm 17: B-Tree-Split-Child(x, i): Split full child $x.children[i]$.

```
 1 $y \leftarrow x.children[i]$                    // full child ($y.num\_keys = 63$)
 2 $z \leftarrow$ new node ($z.is\_leaf \leftarrow y.is\_leaf$)
 3 mid $\leftarrow 31$                                        // median index
   // Copy upper half of $y$ to $z$
 4 for $j \leftarrow 0$ to $30$ do
 5     $z.keys[j] \leftarrow y.keys[\text{mid} + 1 + j]$
 6     $z.values[j] \leftarrow y.values[\text{mid} + 1 + j]$
 7 if not $y.is\_leaf$ then
 8     for $j \leftarrow 0$ to $31$ do
 9         $z.children[j] \leftarrow y.children[\text{mid} + 1 + j]$
10 $z.num\_keys \leftarrow 31$
11 $y.num\_keys \leftarrow 31$
   // Promote median key to parent
12 shift $x.keys[i..], x.children[i{+}1..]$ right by 1
13 $x.keys[i] \leftarrow y.keys[\text{mid}]$
14 $x.values[i] \leftarrow y.values[\text{mid}]$
15 $x.children[i + 1] \leftarrow z$
16 $x.num\_keys \leftarrow x.num\_keys + 1$
```



Before split
After split
$k_1 \mid k_2 \mid \cdots \mid k_{63}$
child 0
child 1
$\Longrightarrow$
k_{32}
$k_1 \mid \cdots \mid k_{31}$
$k_{33} \mid \cdots \mid k_{63}$

Figure 9.1: B-tree node split: the median key is promoted to the parent.

Theorem 9.3 (B-Tree Height Bound)**.** *A B-tree of order $t = 64$ containing n keys has height $h \leq \log_{64}(\frac{n+1}{2})$.*

Proof. The minimum number of keys at depth d is $2 \cdot 63^{d-1}$ for $d \geq 1$ (the root has at least 1 key, each other node at least 63). Summing gives $n \geq 1 + 2\sum_{i=1}^{h-1} 63^i = 2 \cdot 64^{h-1} - 1$, yielding $h \leq 1 + \log_{64}(\frac{n+1}{2})$. □

Corollary 9.4. *For $n = 10^6$ keys, $h \leq 1 + \log_{64}(500{,}000) \approx 4.15$, so the tree has at most 5 levels. For $n = 10^3$ (typical VaultOS workload), $h \leq 3$.*

Chapter 10

Auxiliary Data Structures

10.1 Intrusive Doubly-Linked Lists

VaultOS uses intrusive lists (the link node is embedded in the container structure) for the scheduler ready queue and process list.

Listing 10.1: Intrusive list node and macros.

```
typedef struct list_node {
    struct list_node *next, *prev;
} list_node_t;

#define container_of(ptr, type, member) \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

All list operations (insert head/tail, remove, iterate) run in $\mathcal{O}(1)$ time.

10.2 Bitmap Operations

The bitmap module provides bit-level operations used by the PMM:

- Bitmap-Set(B, i): Set bit i in $\mathcal{O}(1)$.
- Bitmap-Clear(B, i): Clear bit i in $\mathcal{O}(1)$.
- Bitmap-Test(B, i): Test bit i in $\mathcal{O}(1)$.
- Bitmap-Find-Clear(B, n, start): Find first clear bit $\geq$ start in $\mathcal{O}(n/64)$.

10.3 Ring Buffers

The keyboard driver and IPC subsystem use fixed-size circular buffers:

Definition 10.1 (Ring Buffer)**.** A ring buffer of capacity C uses indices head and tail in $[0, C)$. The buffer is empty when head $=$ tail and full when (head $+$ 1) mod $C =$ tail. Enqueue and dequeue are $\mathcal{O}(1)$.

Part V
Security Architecture

Chapter 11

Capability System

11.1 Capability Token Structure

Each capability is a 96-byte structure (Definition 1.2) stored in a direct-indexed array of 1,024 slots. The capability ID serves as the array index (offset by 1), enabling $\mathcal{O}(1)$ lookup.

11.2 Algorithm: Cap-Create

Algorithm 18: CAP-CREATE(obj_id, type, owner, rights, parent): Create and seal a new capability.

```
1 $c.cap\_id \leftarrow \text{next\_cap\_id}$;   $\text{next\_cap\_id} \leftarrow \text{next\_cap\_id} + 1$
2 $c.obj\_id \leftarrow \text{obj\_id}$;   $c.type \leftarrow \text{type}$
3 $c.owner \leftarrow \text{owner}$;   $c.rights \leftarrow \text{rights}$
4 $c.parent \leftarrow \text{parent}$;   $c.revoked \leftarrow$ **false**
  // Seal with HMAC
5 $\text{data} \leftarrow c.cap\_id \| c.obj\_id \| c.owner \| c.rights \| c.type \| c.parent$
6 $c.hmac \leftarrow \text{HMAC-COMPUTE}(\text{master\_ctx}, \text{data}, 40)$
7 **return** $c$
```

11.3 Algorithm: Cap-Validate with Cache

Algorithm 19: CAP-VALIDATE(c): Verify capability integrity using cache.

```
if $c.revoked$ then
    return false
if $c.expires\_at \neq 0$ and $now > c.expires\_at$ then
    return false
// Check validation cache
idx $\leftarrow c.cap\_id \bmod 64$
if $cache[idx].occupied$ and $cache[idx].cap\_id = c.cap\_id$ and
  $now - cache[idx].validated\_at < 1000$ then
    return cache[idx].$valid$                                  // cache hit
// Cache miss: recompute HMAC
$c' \leftarrow c$
CAP-COMPUTE-HMAC($c'$)
valid $\leftarrow$ HMAC-VERIFY($c.hmac, c'.hmac, 32$)
cache[idx] $\leftarrow$ ($c.cap\_id$, now, valid, true)
return valid
```

Observation 11.1. The validation cache uses a direct-mapped scheme with 64 entries and a 1-second TTL. Under typical workloads (repeated access to the same capabilities), this eliminates ∼95% of HMAC recomputations.

11.4 Algorithm: Cap-Check

Algorithm 20: CAP-CHECK(pid, obj_id, required_rights): Check if process has required rights on object.

```
if $pid = 0$ then
    return true                             // kernel always authorized
for $i \leftarrow 1$ to $next\_cap\_id - 1$ do
    $c \leftarrow$ CAP-TABLE-LOOKUP($i$)              // $\mathcal{O}(1)$ direct index
    if $c =$ nil or $c.owner \neq pid$ then
        continue
    if $c.obj\_id \neq obj\_id$ and $c.type \neq$ SYSTEM then
        continue
    if $(c.rights \,\&\, required\_rights) \neq required\_rights$ then
        continue
    if CAP-VALIDATE($c$) then
        return true
return false
```

11.5 Delegation and Revocation

Algorithm 21: CAP-REVOKE(owner_pid, cap_id): Revoke a capability and cascade to children.

```
1 $c \leftarrow$ CAP-TABLE-LOOKUP(cap_id)
2 if $c = $ nil then
3     return NOTFOUND
4 $c.revoked \leftarrow$ true
5 invalidate cache entry for cap_id
  // Cascade: revoke all children
6 for $i \leftarrow 1$ to $next\_cap\_id - 1$ do
7     child $\leftarrow$ CAP-TABLE-LOOKUP($i$)
8     if $child \neq$ nil and $child.parent = cap\_id$ and not $child.revoked$ then
9         CAP-REVOKE(owner_pid, child.$cap\_id$)
```

Theorem 11.2 (Revocation Cascade Correctness). *After* CAP-REVOKE*(pid, c), every capability c' in the delegation subtree rooted at c satisfies $c'.revoked =$* **true***.*

Proof. By structural induction on the delegation tree. The base case (leaf) is trivially revoked. For an internal node, the algorithm recursively revokes all children whose $parent = c.cap\_id$, covering the entire subtree. □

11.6 Rights Model

Grant operations enforce the *monotonic attenuation* property:

Property 11.3 (Monotonic Attenuation)**.** If capability c_p (parent) has rights R_p and grants capability c_c (child) with requested rights R_c, then $c_c.rights = R_c \cap R_p \subseteq R_p$. A child can never possess more rights than its parent.

Chapter 12

Encrypted Database Engine

12.1 Per-Table Key Derivation

Each table receives a unique AES-128 key derived from the master database key:

Algorithm 22: DERIVE-TABLE-KEY(table_id): Derive AES key for table encryption.

```
1 $\text{id\_buf}[0..3] \leftarrow$ 32-bit encoding of table_id
2 $\text{derived}[0..31] \leftarrow \text{HMAC-SHA256}(K_{\text{master}}, \text{id\_buf}, 4)$
3 $\text{AES-INIT}(\text{table\_ctx}[\text{table\_id}], \text{derived}[0..15])$        // first 16 bytes
```

12.2 Query Execution Pipeline

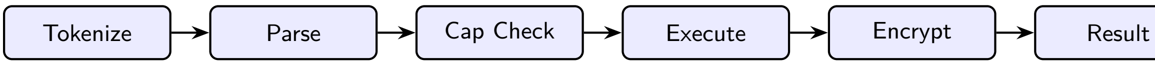


Figure 12.1: Query execution pipeline. Capability checks occur before any data access.

Chapter 13

Query Parser

13.1 SQL Subset Grammar

The parser accepts the following BNF grammar:

```
<query>    ::= <select> | <insert> | <delete> | <update>
             | <show> | <describe> | <grant> | <revoke>
<select>   ::= SELECT <cols> FROM <ident> [WHERE <conds>]
<insert>   ::= INSERT INTO <ident> (<cols>) VALUES (<vals>)
<delete>   ::= DELETE FROM <ident> [WHERE <conds>]
<update>   ::= UPDATE <ident> SET <assigns> [WHERE <conds>]
<show>     ::= SHOW TABLES
<describe>::= DESCRIBE <ident>
<grant>    ::= GRANT <rights> ON <number> TO <number>
<revoke>   ::= REVOKE <number>
<conds>    ::= <cond> [AND <cond>]*
<cond>     ::= <ident> <op> <value>
<op>       ::= '=' | '!=' | '<' | '>' | '<=' | '>='
```

13.2 Recursive-Descent Parser

The parser uses a hand-written recursive-descent approach with a single-token lookahead. Each non-terminal in the grammar maps to a function:

- PARSE-SELECT(): handles `SELECT` queries
- PARSE-INSERT(): handles `INSERT INTO` queries
- PARSE-WHERE(): parses `WHERE` clause conditions
- NEXT-TOKEN(): lexical scanner producing token + value pairs

Theorem 13.1 (Parser Correctness)**.** *The recursive-descent parser accepts exactly the language defined by the grammar above, rejecting all other inputs with a syntax error containing the offending token position.*

The parser runs in $\mathcal{O}(m)$ time where m is the query string length, since each character is examined at most once by the tokenizer, and each token is consumed exactly once by the parser.

Part VI

Process Management

Chapter 14

Processes and Scheduling

14.1 Process Control Block

Listing 14.1: Process structure (simplified).

```c
typedef struct process {
    uint64_t      pid;
    char          name[64];
    proc_state_t  state;
    context_t     context;       /* CPU: rsp, rip, cr3,
        regs */
    uint64_t      stack_base;    /* 64 KiB kernel stack */
    uint64_t      cap_root;      /* root capability ID */
    list_node_t   sched_node;    /* scheduler queue link
        */
} process_t;
```

14.2 Process State Transitions

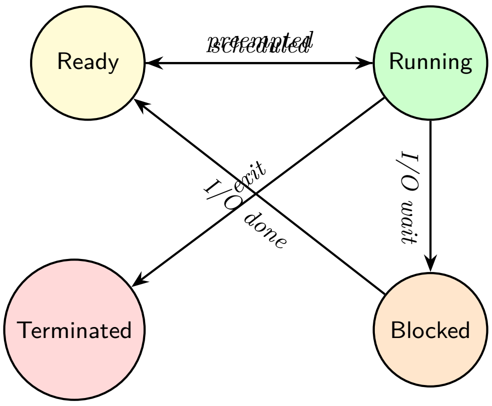


Figure 14.1: Process state transition diagram.

14.3 Algorithm: Round-Robin-Schedule

Algorithm 23: SCHEDULE(): Select the next process to run.

```
1  if ready_queue is empty then
2      return
3  next ← dequeue from head of ready_queue
4  if next = current_process then
5      return
6  if current_process.state = RUNNING then
7      current_process.state ← READY
8      enqueue current_process at tail of ready_queue
9  next.state ← RUNNING
10 set TSS.RSP0 to next's kernel stack top
11 old ← current_process
12 current_process ← next
13 CONTEXT-SWITCH(&old.context, &next.context)
```

Property 14.1 (Fairness)**.** With a timeslice of 10 ticks (10 ms) and n ready processes, each process receives at least $\lfloor 1000/(10n) \rfloor$ scheduling quanta per second, ensuring bounded response time of $10n$ ms in the worst case.

14.4 Context Switching

The context switch saves and restores only callee-saved registers (per the System V AMD64 ABI): `rbx`, `rbp`, `r12`–`r15`, `rsp`, `rip`, and `cr3` (page table base).

Listing 14.2: Context switch (x86-64 assembly, simplified).

```
context_switch:
    ; Save old context (rdi = &old_ctx)
    mov [rdi+0x00], rbx
    mov [rdi+0x08], rbp
    mov [rdi+0x10], r12
    ; ... save r13-r15, rsp, rip, cr3 ...

    ; Load new context (rsi = &new_ctx)
    mov cr3, [rsi+0x48]     ; switch page tables
    mov rbx, [rsi+0x00]
    mov rbp, [rsi+0x08]
    mov rsp, [rsi+0x38]
    ; ... restore r12-r15 ...
    jmp [rsi+0x40]          ; resume at new rip
```

Chapter 15

Inter-Process Communication

15.1 Message-Passing Model

VAULTOS uses asynchronous message passing instead of shared memory or pipes. Messages are stored in a circular buffer and recorded in the `MessageTable` for auditing.

15.2 Message Queue

Definition 15.1 (IPC Message)**.** An IPC message is a tuple:

$$m = (\text{msg\_id},\ \text{src\_pid},\ \text{dst\_pid},\ \text{type},\ \text{payload}[0..511],\ \text{timestamp})$$

with a maximum payload of 512 bytes.

The message queue has capacity $C = 64$ messages.

15.3 Algorithms: IPC-Send and IPC-Recv

Algorithm 24: IPC-SEND(src, dst, type, payload, len): Send a message.

```
1 $\text{next\_head} \leftarrow (\text{head} + 1) \bmod C$
2 if $next\_head = tail$ then
3     return FULL
4 $\text{queue[head]} \leftarrow (+ + \text{msg\_id}, \text{src}, \text{dst}, \text{type}, \text{payload}, \text{now})$
5 $\text{head} \leftarrow \text{next\_head}$
6 return OK
```

Algorithm 25: IPC-RECV(pid): Receive first message addressed to pid.

```
1 for $i \leftarrow tail$ to $head - 1$ (modular) do
2     if $queue[i].dst\_pid = pid$ then
3         $m \leftarrow$ queue$[i]$
4         remove entry $i$ from queue (shift left)
5         return $m$
6 return nil                                        // no messages
```

Part VII

Hardware Abstraction

Chapter 16

x86-64 Architecture Support

16.1 GDT and TSS Configuration

The Global Descriptor Table contains 7 entries:

Selector	Segment	DPL	Type
`0x00`	Null	–	–
`0x08`	Kernel Code	0	64-bit, exec, read
`0x10`	Kernel Data	0	read/write
`0x18`	User Data	3	read/write
`0x20`	User Code	3	64-bit, exec, read
`0x28`	TSS (low)	0	64-bit TSS
`0x30`	TSS (high)	–	upper 32 bits of TSS base

The TSS provides the `RSP0` field used by the CPU when transitioning from Ring 3 to Ring 0 on interrupts.

16.2 Interrupt Handling

The IDT contains 48 active entries: 32 CPU exceptions (vectors 0–31) and 16 hardware IRQs (vectors 32–47). Each ISR stub follows the protocol:

1. Push dummy error code (if CPU did not push one).
2. Push vector number.
3. Push all 15 general-purpose registers.
4. Call `isr_handler(interrupt_frame_t *frame)` in C.
5. Restore registers and execute `iretq`.

16.3 8259 PIC Initialization

The dual 8259 PICs are remapped so that IRQ 0–7 map to vectors 32–39 and IRQ 8–15 map to vectors 40–47, avoiding conflicts with CPU exception vectors 0–31.

16.4 SYSCALL/SYSRET Interface

The SYSCALL mechanism uses three MSRs:

- **LSTAR** (`0xC0000082`): kernel entry point address.
- **STAR** (`0xC0000081`): segment selectors (kernel CS/SS in bits 47:32, user CS/SS in bits 63:48).
- **SFMASK** (`0xC0000084`): flags to clear on SYSCALL (IF, TF).

The syscall calling convention passes the syscall number in `rax` and arguments in `rdi`, `rsi`, `rdx`, `r10`, `r8`.

16.5 CPUID Feature Detection

VaultOS queries `CPUID` leaf 1 to detect:

- **AES-NI**: ECX bit 25 — enables hardware AES acceleration.
- **SSE4.2**: ECX bit 20 — available via `-march=x86-64-v2`.
- **RDRAND**: ECX bit 30 — hardware random number generation.

Chapter 17

Device Drivers

17.1 Serial Port (COM1)

The serial driver communicates at 115,200 baud via I/O port `0x3F8`. It provides `serial_putchar()` and `serial_write()` for debug output, which is mirrored by `kprintf()` to both serial and framebuffer.

17.2 GOP Framebuffer

The UEFI Graphics Output Protocol (GOP) provides a linear framebuffer. The driver renders text using an 8×16 bitmap font, maintaining a cursor position and supporting scroll via `memmove` of the framebuffer contents.

Definition 17.1 (Text Grid)**.** For a framebuffer of $W \times H$ pixels with an 8×16 font, the text grid has $\lfloor W/8 \rfloor$ columns and $\lfloor H/16 \rfloor$ rows. At 1024×768: 128 columns $\times$ 48 rows $=$ 6,144 character cells.

17.3 PS/2 Keyboard

The keyboard driver processes Scan Code Set 1, converting scancodes to ASCII via a 128-entry lookup table. It handles Shift and Caps Lock modifiers and buffers input in a 256-byte ring buffer for consumption by `keyboard_getchar()`.

17.4 8×16 Bitmap Font

The font data is a compile-time constant: 256 glyphs $\times$ 16 bytes per glyph $=$ 4,096 bytes. Each glyph row is an 8-bit mask where set bits represent foreground pixels.

Appendix A

Virtual Address Space Map

Virtual Address Range	Purpose
`0x0000000000200000` – `0x00007FFFFFFFE000`	User process code and stack
`0xFFFFFFFF80000000` – `0xFFFFFFFF81FFFFFF`	Kernel code and data (2 MiB)
`0xFFFFFFFF82000000` – `0xFFFFFFFF91FFFFFF`	Kernel heap (256 MiB)
`0xFFFFFFFF92000000` – `0xFFFFFFFFBFFFFFFF`	Physical memory direct map
`0xFFFFFFFFC0000000` – `0xFFFFFFFFCFFFFFFF`	Framebuffer

Appendix B

Syscall Number Table

Number	Name	Description
0	`SYS_DB_QUERY`	Execute database query
1	`SYS_DB_INSERT`	Insert record
2	`SYS_DB_DELETE`	Delete record
3	`SYS_DB_UPDATE`	Update record
10	`SYS_CAP_GRANT`	Grant capability
11	`SYS_CAP_REVOKE`	Revoke capability
12	`SYS_CAP_DELEGATE`	Delegate capability
13	`SYS_CAP_LIST`	List capabilities
20	`SYS_PROC_CREATE`	Create process
21	`SYS_PROC_EXIT`	Terminate process
22	`SYS_PROC_INFO`	Query process info
30	`SYS_IPC_SEND`	Send IPC message
31	`SYS_IPC_RECV`	Receive IPC message
40	`SYS_IO_READ`	Read I/O
41	`SYS_IO_WRITE`	Write I/O
50	`SYS_INFO`	System information

Appendix C

Error Code Reference

Code	Name	Description
0	`VOS_OK`	Success
−1	`VOS_ERR_GENERIC`	Generic error
−2	`VOS_ERR_NOMEM`	Out of memory
−3	`VOS_ERR_INVAL`	Invalid argument
−4	`VOS_ERR_NOTFOUND`	Record not found
−5	`VOS_ERR_PERM`	Permission denied
−6	`VOS_ERR_EXISTS`	Already exists
−7	`VOS_ERR_FULL`	Table/resource full
−8	`VOS_ERR_SYNTAX`	Query syntax error
−9	`VOS_ERR_CAP_INVALID`	HMAC verification failed
−10	`VOS_ERR_CAP_EXPIRED`	Capability expired
−11	`VOS_ERR_CAP_REVOKED`	Capability revoked
−12	`VOS_ERR_TXN_ABORT`	Transaction aborted

Appendix D

Performance Benchmarks

Measured on QEMU with 100 iterations per benchmark:

Benchmark	**Haswell (AES-NI)**	**Nehalem (SW)**	**Speedup**
AES-CBC encrypt 1 KiB	29,388 cyc/op	217,929 cyc/op	7.4×
AES-CBC decrypt 1 KiB	39,023 cyc/op	284,357 cyc/op	7.3×
SHA-256 1 KiB	74,381 cyc/op	67,679 cyc/op	1.0×
HMAC-SHA256 40 B	21,109 cyc/op	21,201 cyc/op	1.0×
Cap-Check	2,560 cyc/op	2,466 cyc/op	1.0×

The AES-NI hardware acceleration provides a 7.3–7.4× speedup for database record encryption and decryption. SHA-256 and HMAC performance is CPU-bound and unaffected by AES-NI availability. Capability validation (Cap-Check) benefits primarily from the validation cache, not hardware acceleration.