

# VaultOS

A Capability-Secured Nanokernel with  
Database-Centric Architecture

*Design, Algorithms, and Implementation*

GUI Desktop: Window Manager · Compositor · Applications

VaultShell: Friendly CLI · SQL REPL · Script Engine

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

<b>Preface</b>	<b>vii</b>
<b>I Foundations</b>	<b>1</b>
<b>1 Introduction and Design Philosophy</b>	<b>3</b>
1.1 The Database-as-OS Paradigm . . . . .	3
1.2 Capability-Based Security Model . . . . .	4
1.3 Threat Model . . . . .	4
1.4 System Architecture Overview . . . . .	5
1.5 Comparison with Traditional OS Designs . . . . .	5
<b>2 System Bootstrap</b>	<b>7</b>
2.1 UEFI Boot Protocol . . . . .	7
2.2 BootInfo Structure . . . . .	7
2.3 Kernel Initialization Sequence . . . . .	8
2.4 Higher-Half Kernel Design . . . . .	8
<b>II Memory Management</b>	<b>9</b>
<b>3 Physical Memory Manager</b>	<b>11</b>
3.1 Bitmap Allocator Design . . . . .	11
3.2 Algorithm: PMM-ALLOC . . . . .	11
3.3 Algorithm: PMM-FREE . . . . .	12
3.4 Complexity Analysis . . . . .	12
<b>4 Virtual Memory and Paging</b>	<b>13</b>
4.1 x86-64 Four-Level Page Tables . . . . .	13
4.2 Virtual Address Space Layout . . . . .	13
4.3 Algorithm: PAGE-MAP . . . . .	14
4.4 Algorithm: VIRT-TO-PHYS . . . . .	14

<b>5 Kernel Heap Allocator</b>	<b>17</b>
5.1 First-Fit with Coalescing . . . . .	17
5.2 Algorithm: KMALLOC . . . . .	17
5.3 Algorithm: KFREE with Coalescing . . . . .	18
 <b>III Cryptographic Primitives</b>	 <b>19</b>
<b>6 SHA-256</b>	<b>21</b>
6.1 Merkle-Damgård Construction . . . . .	21
6.2 Compression Function . . . . .	22
6.3 Block-Based Update . . . . .	23
<b>7 AES-128</b>	<b>25</b>
7.1 Rijndael Cipher Structure . . . . .	25
7.2 GF( $2^8$ ) Arithmetic and Precomputed Tables . . . . .	25
7.3 Algorithm: AES-KEY-EXPAND . . . . .	26
7.4 Algorithm: AES-ENCRYPT-BLOCK . . . . .	26
7.5 CBC Mode . . . . .	26
7.6 AES-NI Hardware Acceleration . . . . .	27
<b>8 HMAC-SHA256 and Random Number Generation</b>	<b>29</b>
8.1 HMAC Construction . . . . .	29
8.2 Pre-computed Context . . . . .	29
8.3 Constant-Time Verification . . . . .	30
8.4 Random Number Generation . . . . .	30
 <b>IV Data Structures</b>	 <b>31</b>
<b>9 B-Tree Index</b>	<b>33</b>
9.1 B-Tree Properties . . . . .	33
9.2 Node Structure . . . . .	33
9.3 Algorithm: B-TREE-SEARCH . . . . .	34
9.4 Algorithm: B-TREE-INSERT . . . . .	34
9.5 Algorithm: B-TREE-SPLIT-CHILD . . . . .	35
9.6 Algorithm: B-TREE-DELETE (Lazy) . . . . .	36
<b>10 Auxiliary Data Structures</b>	<b>37</b>
10.1 Intrusive Doubly-Linked Lists . . . . .	37
10.2 Bitmap Operations . . . . .	37
10.3 Ring Buffers . . . . .	37

<b>V Security Architecture</b>	<b>39</b>
<b>11 Capability System</b>	<b>41</b>
11.1 Capability Token Structure . . . . .	41
11.2 Algorithm: CAP-CREATE . . . . .	41
11.3 Algorithm: CAP-VALIDATE with Cache . . . . .	42
11.4 Algorithm: CAP-CHECK . . . . .	42
11.5 Delegation and Revocation . . . . .	43
11.6 Rights Model . . . . .	43
<b>12 Encrypted Database Engine</b>	<b>45</b>
12.1 Per-Table Key Derivation with Domain Separation . . . . .	45
12.2 Record Serialization . . . . .	46
12.3 Encrypt-then-MAC Pipeline . . . . .	46
12.4 Verify-then-Decrypt Pipeline . . . . .	47
12.5 Constant-Time Audit Logging . . . . .	49
12.6 Query Execution Pipeline . . . . .	49
<b>13 Query Parser</b>	<b>51</b>
13.1 SQL Subset Grammar . . . . .	51
13.2 Recursive-Descent Parser . . . . .	51
<b>VI Process Management</b>	<b>53</b>
<b>14 Processes and Scheduling</b>	<b>55</b>
14.1 Process Control Block . . . . .	55
14.2 Process State Transitions . . . . .	55
14.3 Algorithm: ROUND-ROBIN-SCHEDULE . . . . .	56
14.4 Context Switching . . . . .	56
<b>15 Inter-Process Communication</b>	<b>59</b>
15.1 Message-Passing Model . . . . .	59
15.2 Message Queue . . . . .	59
15.3 Algorithms: IPC-SEND and IPC-RECV . . . . .	59
<b>VII Hardware Abstraction</b>	<b>61</b>
<b>16 x86-64 Architecture Support</b>	<b>63</b>
16.1 GDT and TSS Configuration . . . . .	63
16.2 Interrupt Handling . . . . .	63
16.3 8259 PIC Initialization . . . . .	64
16.4 SYSCALL/SYSRET Interface . . . . .	64

16.5 CPUID Feature Detection . . . . .	64
<b>17 Device Drivers</b>	<b>65</b>
17.1 Serial Port (COM1) . . . . .	65
17.2 GOP Framebuffer . . . . .	65
17.3 PS/2 Keyboard . . . . .	65
17.4 PS/2 Mouse . . . . .	65
17.5 8×16 Bitmap Font . . . . .	66
<b>VIII User Interface</b>	<b>67</b>
<b>18 VaultShell</b>	<b>69</b>
18.1 Text User Interface . . . . .	69
18.2 Line Editor . . . . .	70
18.3 Display Formatter . . . . .	70
18.4 Algorithm: Context-Aware Tab Completion . . . . .	70
<b>19 Friendly Command Layer</b>	<b>73</b>
19.1 Design: Commands as SQL Generators . . . . .	73
19.2 Command-to-SQL Translation . . . . .	73
19.3 Table Aliases . . . . .	73
19.4 Script Engine . . . . .	74
19.5 Process Management Commands . . . . .	75
19.6 Algorithm: Friendly-Translate . . . . .	77
<b>20 Graphical Desktop</b>	<b>79</b>
20.1 Graphics Subsystem . . . . .	79
20.2 Event System . . . . .	80
20.3 Window Manager . . . . .	80
20.4 Compositor . . . . .	81
20.5 Widget Toolkit . . . . .	81
20.6 Desktop Applications . . . . .	82
20.6.1 Database Tools . . . . .	82
20.6.2 Productivity Applications . . . . .	82
20.6.3 Security Applications . . . . .	83
20.6.4 System Tools . . . . .	84
20.6.5 Taskbar and Start Menu . . . . .	84
<b>A Virtual Address Space Map</b>	<b>85</b>
<b>B Syscall Number Table</b>	<b>87</b>
<b>C Error Code Reference</b>	<b>89</b>

<b>D Performance Benchmarks</b>	<b>91</b>
<b>E VaultShell Command Reference</b>	<b>93</b>

*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.

*CONTENTS*

---

# **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  $\langle id, col_1, \dots, col_k \rangle$  in a table  $T$ , and every operation on  $r$  is expressed as a query  $Q \in \{\text{SELECT}, \text{INSERT}, \text{UPDATE}, \text{DELETE}\}$  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  $\sigma = \text{HMAC-SHA256}(K_{\text{master}}, \text{cap\_id} \parallel \text{obj\_id} \parallel \text{owner} \parallel \text{rights} \parallel \text{type} \parallel \text{parent})$  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  $K_{\text{master}}$  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.  $\square$

## 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.

**Cryptographic RNG dependency.** The security of all encryption and capability operations depends on the availability of hardware RDRAND (Intel Ivy Bridge and later). When RDRAND is unavailable, VaultOS falls back to a RDTSC-seeded xorshift128+ generator, which provides statistical but *not* cryptographic randomness. In virtualised environments where TSC values are predictable, this fallback weakens all security guarantees—IVs become predictable and capability tokens lose unforgeability. The system logs a warning at boot when operating in fallback mode.

## 1.4 System Architecture Overview

Figure 1.1 shows the layered architecture.

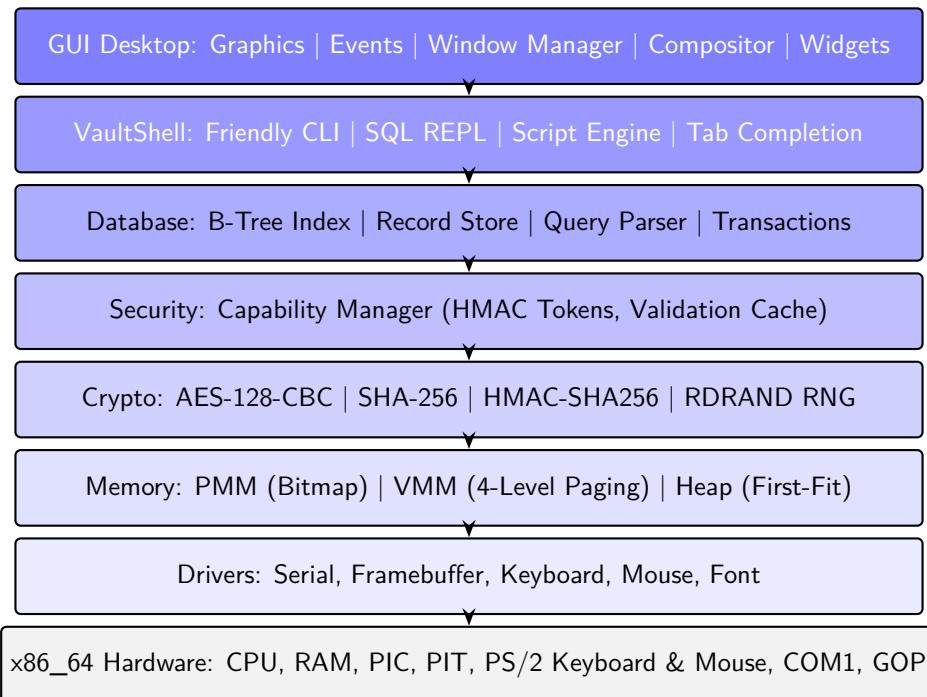


Figure 1.1: VaultOS layered architecture. Each layer depends only on the layers below it.

## 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	<code>fork/exec/wait</code>	<code>proc_create/exit</code>
Encryption	Optional (dm-crypt, etc.)	Mandatory, per-table
Audit	Syslog (optional)	Built-in AuditTable
User interface	Shell + utilities	Friendly CLI + SQL REPL + GUI Desktop

# 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  $1024 \times 768$  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).

```
1 typedef struct __attribute__((packed)) {
2     uint64_t fb_base, fb_width, fb_height, fb_pitch;
3     uint32_t fb_bpp, fb_pixel_format;
4     uint64_t mmap_base, mmap_size, mmap_desc_size;
5     uint32_t mmap_entry_count;
6     uint64_t kernel_phys_base, kernel_size;
7     uint64_t rsdp_address;
8 } BootInfo;
```

## 2.3 Kernel Initialization Sequence

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

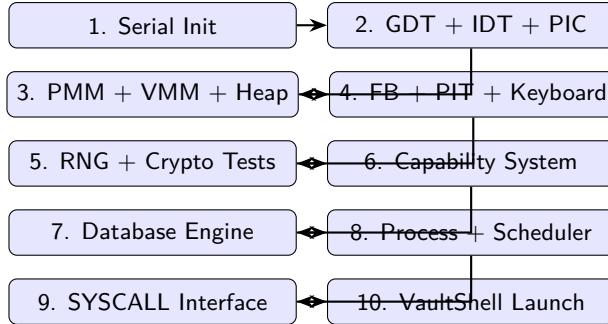


Figure 2.1: Kernel initialization sequence. Each phase depends on all preceding phases.

## 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:

0x0000000000000000–0x00007FFFFFFFFF	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  $B[0 \dots \lceil N/64 \rceil - 1]$  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  $B[w] \neq 0xFFFFFFFFFFFFFFFFF$  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 ≥ N or bit (page mod 64) of B[⟨page/64⟩] is clear then
3   | error “double free or invalid page”
4 clear bit (page mod 64) in B[⟨page/64⟩]
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.  $\square$

# 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:

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

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:

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

## 4.2 Virtual Address Space Layout

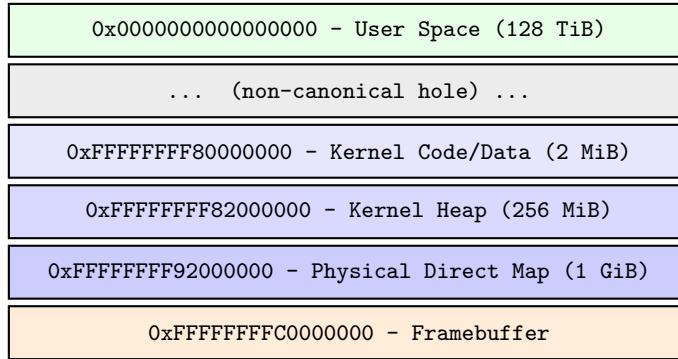


Figure 4.1: Virtual address space layout of the VaultOS kernel.

### 4.3 Algorithm: Page-Map

---

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

---

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

---

### 4.4 Algorithm: Virt-To-Phys

---

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

---

```
1 Walk PML4 → PDPT → PD → PT using index functions
2 if any level is not present then
3   | return 0                                     // page not mapped
4 if PD entry has Huge flag (2 MiB page) then
5   | return pd_phys | (virt & 0xFFFF)
6 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.*

*Proof sketch.*  $\text{PAGE-MAP}(v, p, f)$  writes the entry ( $\text{base} = p$ ,  $\text{flags} = f$ ) into the page-table entry (PTE) selected by the level-1 index of  $v$ . The four-level walk in VIRT-To-PHYS extracts exactly the same indices from  $v$  (bits 47–39, 38–30, 29–21, 20–12) and follows the same PML4→PDP→PD→PT chain. At the final level it reads the PTE base address, which is  $p$ . Since x86-64 address translation is deterministic for a given CR3 and page-table contents, the result follows. The “no intervening unmap” proviso ensures the PTE has not been cleared between the two calls.  $\square$



# 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.

```
1 typedef struct heap_block {
2     uint64_t      magic;      /* 0xDEADBEEF */
3     size_t        size;       /* usable data bytes */
4     bool          free;
5     heap_block_t *next, *prev;
6 } heap_block_t;
```

### 5.2 Algorithm: Kmalloc

---

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

---

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

# 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  $H_0, \dots, H_7$  initialized to the fractional parts of the square roots of the first eight primes:

$$H_0 = 6a09e667, \quad H_1 = bb67ae85, \quad \dots, \quad H_7 = 5be0cd19$$

## 6.2 Compression Function

---

**Algorithm 7:** SHA256-TRANSFORM( $H[0..7]$ ,  $\text{block}[0..63]$ ): 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  $\text{block}[4i \dots 4i+3]$ 
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  $a, b, c, d, e, f, g, h \leftarrow H[0], H[1], \dots, H[7]$ 
    // Compression rounds
6 for  $i \leftarrow 0$  to 63 do
7   |  $T_1 \leftarrow h + \Sigma_1(e) + \text{Ch}(e, f, g) + K_i + W[i]$ 
8   |  $T_2 \leftarrow \Sigma_0(a) + \text{Maj}(a, b, c)$ 
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  $i = 0, \dots, 7$ 

```

---

The mixing functions are defined as:

$$\text{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$\text{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\Sigma_0(x) = \text{ROTR}^2(x) \oplus \text{ROTR}^{13}(x) \oplus \text{ROTR}^{22}(x)$$

$$\Sigma_1(x) = \text{ROTR}^6(x) \oplus \text{ROTR}^{11}(x) \oplus \text{ROTR}^{25}(x)$$

$$\sigma_0(x) = \text{ROTR}^7(x) \oplus \text{ROTR}^{18}(x) \oplus (x \gg 3)$$

$$\sigma_1(x) = \text{ROTR}^{17}(x) \oplus \text{ROTR}^{19}(x) \oplus (x \gg 10)$$

### 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  $64 \times$  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  $4 \times 4$  byte matrix called the *state*.

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

$$\text{Round}(s, k_r) = \text{ADDROUNDKEY}(\text{MIXCOLUMNS}(\text{SHIFTROWS}(\text{SUBBYTES}(s))), k_r)$$

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]	INV MIXCOLUMNS: $\{9\} \cdot x$
mul11[256]	INV MIXCOLUMNS: $\{b\} \cdot x$
mul13[256]	INV MIXCOLUMNS: $\{d\} \cdot x$
mul14[256]	INV MIXCOLUMNS: $\{e\} \cdot x$

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.

---

```
1  $w[0..3] \leftarrow$  32-bit words from key
2 for  $i \leftarrow 4$  to 43 do
3   | temp  $\leftarrow w[i - 1]$ 
4   | if  $i \bmod 4 = 0$  then
5     |   | temp  $\leftarrow \text{SUBWORD}(\text{ROTWORD}(\text{temp})) \oplus \text{Rcon}[i/4]$ 
6   |   |  $w[i] \leftarrow w[i - 4] \oplus \text{temp}$ 
```

---

### 7.4 Algorithm: AES-Encrypt-Block

---

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

---

```
1 ADDROUNDKEY(block, ctx.rk[0])
2 for  $r \leftarrow 1$  to 9 do
3   | SUBBYTES(block)           // S-box lookup, 16 bytes
4   | SHIFTROWS(block)         // cyclic row rotations
5   | MIXCOLUMNS(block)        // GF(28) via lookup tables
6   | ADDROUNDKEY(block, ctx.rk[r])
7 SUBBYTES(block)
8 SHIFTROWS(block)
9 ADDROUNDKEY(block, ctx.rk[10])
```

---

### 7.5 CBC Mode

---

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

---

```
1 prev  $\leftarrow$  iv
2 for off  $\leftarrow 0$  to len - 16 step 16 do
3   | for i  $\leftarrow 0$  to 15 do
4     |   | ct[off + i]  $\leftarrow$  pt[off + i]  $\oplus$  prev[i]
5     |   | AES-ENCRYPT-BLOCK(ctx, ct + off)
6   |   | prev  $\leftarrow$  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).

```

1 movdqu (%block), %xmm0      ; Load plaintext block
2 pxor 0(%rk), %xmm0        ; AddRoundKey 0
3 aesenc 16(%rk), %xmm0     ; Rounds 1-9 (9 instructions
    )
4 aesenc 32(%rk), %xmm0
5 ; ... (rounds 3-9) ...
6 aesenclast 160(%rk), %xmm0 ; Final round
7 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  $\text{ipad} = 0x36^{64}$  and  $\text{opad} = 0x5C^{64}$ .

### 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  $\text{ipad}[i] \leftarrow K'[i] \oplus 0x36$  for  $i = 0, \dots, 63$ 
6  $\text{opad}[i] \leftarrow K'[i] \oplus 0x5C$  for  $i = 0, \dots, 63$ 
7  $\text{ctx.inner\_base} \leftarrow \text{SHA256-UPDATE}(\text{SHA256-INIT}(), \text{ipad}, 64)$ 
8  $\text{ctx.outer\_base} \leftarrow \text{SHA256-UPDATE}(\text{SHA256-INIT}(), \text{opad}, 64)$ 
```

---

---

**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  $(t, q, \epsilon)$ -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  $b = 64$ , providing  $\mathcal{O}(\log_b n)$  search, insert, and delete operations.

### 9.1 B-Tree Properties

**Definition 9.1** (B-Tree of Order  $b$ ). A B-tree of order  $b$  (maximum branching factor) satisfies:

1. Every node has at most  $b$  children and  $b - 1$  keys ( $= 63$  for  $b = 64$ ).
2. Every non-root internal node has at least  $\lceil b/2 \rceil$  children ( $= 32$ ) and  $\lceil b/2 \rceil - 1$  keys ( $= 31$ ).
3. The root has at least 2 children (if non-leaf) or 0 keys (if empty tree).
4. A node with  $k$  keys has  $k + 1$  children (if internal).
5. 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.* Our order  $b$  corresponds to CLRS *minimum degree*  $t = b/2 = 32$ ; CLRS requires  $t - 1 \leq k \leq 2t - 1$  keys per non-root node, giving  $31 \leq k \leq 63$ , which matches our  $\lceil b/2 \rceil - 1 \leq k \leq b - 1$ . Some references (Knuth) define order as the maximum number of keys; ours defines it as the maximum number of children.

### 9.2 Node Structure

Listing 9.1: B-tree node structure.

```
1 typedef struct btree_node {
2     uint64_t    keys[63];           /* sorted key array */
3     void        *values[63];       /* associated record
4                                pointers */
4     btree_node *children[64];     /* child pointers */
5     uint32_t    num_keys;
6     bool        is_leaf;
7 } 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  $i < x.\text{num\_keys}$  and  $k > x.\text{keys}[i]$  do
3   |  $i \leftarrow i + 1$ 
4   | if  $i < x.\text{num\_keys}$  and  $k = x.\text{keys}[i]$  then
5     |   return  $x.\text{values}[i]$ 
6   | if  $x.\text{is\_leaf}$  then
7     |   return nil
8   return B-TREE-SEARCH( $x.\text{children}[i], k$ )
```

---

## 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  $r \leftarrow T.\text{root}$ 
2 if  $r.\text{num\_keys} = 63$  then
3   |  $s \leftarrow \text{new node (leaf = false)}$ 
4   |  $s.\text{children}[0] \leftarrow r$ 
5   |  $T.\text{root} \leftarrow s$ 
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
4 for  $j \leftarrow 0$  to 30 do
5    $z.keys[j] \leftarrow y.keys[mid + 1 + j]$ 
6    $z.values[j] \leftarrow y.values[mid + 1 + j]$ 
7 if not  $y.is\_leaf$  then
8   for  $j \leftarrow 0$  to 31 do
9      $z.children[j] \leftarrow y.children[mid + 1 + j]$ 
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  $x.keys[i] \leftarrow y.keys[mid]$ 
14  $x.values[i] \leftarrow y.values[mid]$ 
15  $x.children[i + 1] \leftarrow z$ 
16  $x.num\_keys \leftarrow x.num\_keys + 1$ 

```

---

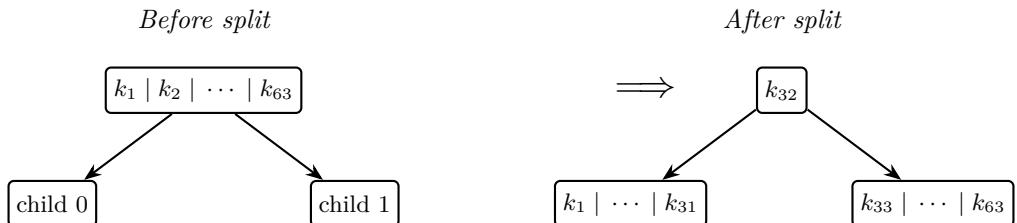


Figure 9.1: B-tree node split: the median key is promoted to the parent.

**Theorem 9.3** (B-Tree Height Bound). *A B-tree of order  $b = 64$  (CLRS minimum degree  $t = b/2 = 32$ ) containing  $n$  keys has height  $h \leq \log_t(\frac{n+1}{2}) = \log_{32}(\frac{n+1}{2})$ .*

*Proof.* The root has at least 1 key and 2 children. Every other internal node has at least  $t = 32$  children and  $t - 1 = 31$  keys. At depth  $d \geq 1$  there are at least  $2t^{d-1}$  nodes, each with at least  $t - 1$  keys. Summing:  $n \geq 1 + 2(t-1) \sum_{i=0}^{h-1} t^i = 2t^h - 1$ , yielding  $h \leq \log_t(\frac{n+1}{2})$ .  $\square$

**Corollary 9.4.** *For  $n = 10^6$  keys,  $h \leq 1 + \log_{32}(500,000) \approx 4.8$ , so the tree has at most 5 levels. For  $n = 10^3$  (typical VaultOS workload),  $h \leq 3$ .*

## 9.6 Algorithm: B-Tree-Delete (Lazy)

---

**Algorithm 18:** B-TREE-DELETE( $T, k$ ): Delete key  $k$  from B-tree  $T$  (lazy MVP).

---

```
1  $x \leftarrow T.root$ 
2 while  $x \neq \text{NULL}$  do
3    $i \leftarrow 0$ 
4   while  $i < x.\text{num\_keys}$  and  $k > x.keys[i]$  do
5      $i \leftarrow i + 1$ 
6   if  $i < x.\text{num\_keys}$  and  $k = x.keys[i]$  then
7     // Key found at position  $i$ 
8     if  $x.\text{is\_leaf}$  then
9       // Case 1: Leaf node -- shift remaining keys left
10      for  $j \leftarrow i$  to  $x.\text{num\_keys} - 2$  do
11         $x.keys[j] \leftarrow x.keys[j + 1]$ 
12         $x.values[j] \leftarrow x.values[j + 1]$ 
13       $x.\text{num\_keys} \leftarrow x.\text{num\_keys} - 1$ 
14      return TRUE
15    else
16      // Case 2: Internal node -- mark value as deleted
17       $x.values[i] \leftarrow \text{NULL}$ 
18      return TRUE
19    if  $x.\text{is\_leaf}$  then
20      return FALSE          // key not in tree
21     $x \leftarrow x.\text{children}[i]$            // descend to child
22 return FALSE
```

---

*Remark 9.5 (Lazy Delete Limitations).* Algorithm 18 is a simplified MVP implementation that does *not* maintain the B-tree minimum-occupancy invariant (Definition 9.1, property 2). Specifically:

- **Leaf nodes** may underflow below  $\lceil b/2 \rceil - 1 = 31$  keys after deletion.
- **Internal nodes** retain tombstoned entries (NULL values) rather than replacing the key with an in-order predecessor/successor.
- No merge or key redistribution between siblings is performed.

For the current VaultOS workload (six system tables with low churn), this is acceptable. A full CLRS-compliant delete with merge and redistribute is planned for a future session.

# 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.

```
1 typedef struct list_node {
2     struct list_node *next, *prev;
3 } list_node_t;
4
5 #define container_of(ptr, type, member) \
6     ((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( $B, n, \text{start}$ ): Find first clear bit  $\geq \text{start}$  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  $\text{head} = \text{tail}$  and full when  $(\text{head} + 1) \bmod C = \text{tail}$ . 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 19:** CAP-CREATE(*obj\_id*, *type*, *owner*, *rights*, *parent*):  
Create and seal a new capability.

---

```
1 c.cap_id  $\leftarrow$  next_cap_id; next_cap_id  $\leftarrow$  next_cap_id + 1
2 c.obj_id  $\leftarrow$  obj_id; c.type  $\leftarrow$  type
3 c.owner  $\leftarrow$  owner; c.rights  $\leftarrow$  rights
4 c.parent  $\leftarrow$  parent; c.revoked  $\leftarrow$  false
   // Seal with HMAC
5 data  $\leftarrow$  c.cap_id||c.obj_id||c.owner||c.rights||c.type||c.parent
6 c.hmac  $\leftarrow$  HMAC-COMPUTE(master_ctx, data, 40)
7 return c
```

---

### 11.3 Algorithm: Cap-Validate with Cache

---

**Algorithm 20:** CAP-VALIDATE( $c$ ): Verify capability integrity using cache.

---

```
1 if  $c.\text{revoked}$  then
2   | return false
3 if  $c.\text{expires\_at} \neq 0$  and  $\text{now} > c.\text{expires\_at}$  then
4   | return false
      // Check validation cache
5  $\text{idx} \leftarrow c.\text{cap\_id} \bmod 64$ 
6 if  $\text{cache}[\text{idx}].\text{occupied}$  and  $\text{cache}[\text{idx}].\text{cap\_id} = c.\text{cap\_id}$  and
     $\text{now} - \text{cache}[\text{idx}].\text{validated\_at} < 1000$  then
7   | return  $\text{cache}[\text{idx}].\text{valid}$                                 // cache hit
      // Cache miss: recompute HMAC
8  $c' \leftarrow c$ 
9 CAP-COMPUTE-HMAC( $c'$ )
10 valid  $\leftarrow$  HMAC-VERIFY( $c.\text{hmac}, c'.\text{hmac}, 32$ )
11  $\text{cache}[\text{idx}] \leftarrow (c.\text{cap\_id}, \text{now}, \text{valid}, \text{true})$ 
12 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  $\sim 95\%$  of HMAC recomputations.

### 11.4 Algorithm: Cap-Check

---

**Algorithm 21:** CAP-CHECK( $\text{pid}$ ,  $\text{obj\_id}$ ,  $\text{required\_rights}$ ): Check if process has required rights on object.

---

```
1 if  $\text{pid} = 0$  then
2   | return true                                         // kernel always authorized
3 for  $i \leftarrow 1$  to  $\text{next\_cap\_id} - 1$  do
4   |  $c \leftarrow \text{CAP-TABLE-LOOKUP}(i)$                   //  $\mathcal{O}(1)$  direct index
5   | if  $c = \text{nil}$  or  $c.\text{owner} \neq \text{pid}$  then
6     |   | continue
7     |   | if  $c.\text{obj\_id} \neq \text{obj\_id}$  and  $c.\text{type} \neq \text{SYSTEM}$  then
8       |     | continue
9     |     | if  $(c.\text{rights} \& \text{required\_rights}) \neq \text{required\_rights}$  then
10      |       | continue
11      |       | if CAP-VALIDATE( $c$ ) then
12        |         | return true
13 return false
```

---

## 11.5 Delegation and Revocation

---

**Algorithm 22:** CAP-REVOKE(owner\\_pid, cap\\_id): Revoke a capability and cascade to children.

---

```

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

```

---

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

*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  $\text{parent} = c.\text{cap\_id}$ , covering the entire subtree.  $\square$

## 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  $c_c.\text{rights} = R_c \cap R_p \subseteq R_p$ . A child can never possess more rights than its parent.



## Chapter 12

# Encrypted Database Engine

VaultOS enforces the principle “all data is confidential” by encrypting every record at rest using per-table AES-128-CBC with HMAC-SHA256 integrity protection. The Encrypt-then-MAC composition guarantees both confidentiality (IND-CPA) and integrity (INT-CTXT), preventing both passive observation and active tampering of stored records.

### 12.1 Per-Table Key Derivation with Domain Separation

Each table requires two independent keys: an AES encryption key and an HMAC authentication key. Deriving both from the same HMAC output would constitute *key reuse*, so we apply domain separation by prepending distinct prefixes to the HMAC input.

---

**Algorithm 23:** DERIVE-TABLE-KEYS(table\_id): Derive AES and MAC keys with domain separation.

---

```
// AES key derivation
1 domainaes ← "AES" || LE32(table_id) // 7-byte input
2  $K_{aes} \leftarrow \text{HMAC-SHA256}(K_{master}, \text{domain}_{aes})[0..15]$ 
3 AES-INIT(aes_ctx[table_id],  $K_{aes}$ )
// MAC key derivation
4 domainmac ← "MAC" || LE32(table_id) // 7-byte input
5  $K_{mac} \leftarrow \text{HMAC-SHA256}(K_{master}, \text{domain}_{mac})$  // full 32 bytes
6 HMAC-INIT(mac_ctx[table_id],  $K_{mac}$ )
7 MEMSET-ZERO( $K_{aes}, K_{mac}$ ) // zeroize intermediates
```

---

The master key  $K_{master}$  is a 256-bit value generated from RDRAND at boot time. Since VaultOS has no persistent storage (yet), keys are ephemeral and regenerated each boot cycle.

## 12.2 Record Serialization

Before encryption, a `record_t` must be converted to a contiguous byte buffer. The wire format uses little-endian encoding:

Field	Bytes	Description
<code>row_id</code>	8	Primary key (uint64)
<code>table_id</code>	4	Table identifier (uint32)
<code>field_count</code>	4	Number of fields
<i>Per-field (repeated <code>field_count</code> times):</i>		
<code>type</code>	4	Column type enum
<code>data</code>	variable	Type-dependent payload

Field data sizes by type:

Type	Data encoding
<code>COL_U64, COL_I64</code>	8 bytes (little-endian)
<code>COL_U32</code>	4 bytes
<code>COL_U8, COL_BOOL</code>	1 byte
<code>COL_STR</code>	2-byte length prefix + <code>length</code> bytes
<code>COL_BLOB</code>	4-byte length prefix + <code>length</code> bytes

Maximum serialized size is bounded by `MAX_RECORD_SIZE` (4096 bytes). Both `RECORD-SERIALIZE` and `RECORD-DESERIALIZE` run in  $\mathcal{O}(f)$  where  $f$  is the field count, with bounds checking at every step.

## 12.3 Encrypt-then-MAC Pipeline

Each record stored in the B-tree is wrapped in an `encrypted_record_t`:

```
1 typedef struct {
2     uint8_t    iv[16];           /* Random IV for AES-CBC */
3     uint8_t    *ciphertext;      /* Encrypted serialized
4                                record */
4     uint32_t   ciphertext_len;  /* Length (PKCS7-padded) */
5     uint8_t    mac[32];         /* HMAC-SHA256(IV ||
6                                ciphertext) */
6     uint64_t   row_id;          /* Plaintext key for B-tree
7                                */
7     uint32_t   table_id;        /* Table this belongs to */
8 } encrypted_record_t;
```

---

**Algorithm 24:** RECORD-ENCRYPT(rec, table\_id): Encrypt a record for storage.

---

```

// Step 1: Serialize
1 plain[0..n-1] ← RECORD-SERIALIZE(rec)

// Step 2: PKCS7 pad to AES block boundary
2 m ← ⌈n/16⌉ × 16
3 PKCS7-PAD(plain, n, m)

// Step 3: Random IV
4 iv[0..15] ← RANDOM-BYTES(16)

// Step 4: AES-CBC encrypt (see §??)
5 ct[0..m-1] ← AES-CBC-ENCRYPT(aes_ctx[table_id], iv, plain, m)

// Step 5: Authenticate (Encrypt-then-MAC)
6 σ ← HMAC-COMPUTE(mac_ctx[table_id], iv || ct)

// Step 6: Build encrypted record
7 enc ← { iv, ct, m, σ, rec.row_id, table_id }
8 B-TREE-INSERT(index[table_id], rec.row_id, enc)
9 MEMSET-ZERO(plain)           // zeroize plaintext buffer

```

---

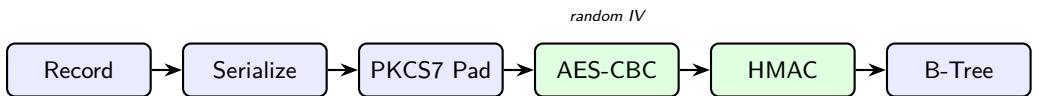


Figure 12.1: Encrypt-then-MAC pipeline for record insertion. Green nodes indicate cryptographic operations.

## 12.4 Verify-then-Decrypt Pipeline

On read, the MAC is verified *before* any decryption occurs. This is essential: attempting to decrypt tampered ciphertext could produce chosen-plaintext oracle attacks if error messages leak information.

---

**Algorithm 25:** RECORD-DECRYPT( $\text{enc}$ ,  $\text{table\_id}$ ): Verify MAC then decrypt.

---

```
// Step 1: Recompute MAC
1  $\sigma' \leftarrow \text{HMAC-COMPUTE}(\text{mac\_ctx}[\text{table\_id}], \text{enc.iv} \parallel \text{enc.ct})$ 
// Step 2: Constant-time verification (see §??)
2 if  $\neg \text{HMAC-VERIFY}(\text{enc.mac}, \sigma', 32)$  then
3   log "MAC verification failed for row enc.row_id"
4   return NULL // reject tampered record
// Step 3: AES-CBC decrypt
5 padded[0.. $m-1$ ]  $\leftarrow$ 
  AES-CBC-DECRYPT(aes_ctx[table_id], enc.iv, enc.ct,  $m$ )
// Step 4: Remove PKCS7 padding
6  $n \leftarrow \text{PKCS7-UNPAD}(\text{padded}, m)$ 
7 if  $n = 0$  then
8   return NULL // invalid padding
// Step 5: Deserialize
9 rec  $\leftarrow \text{RECORD-DESERIALIZE}(\text{padded}[0..n-1])$ 
10 MEMSET-ZERO(padded) // zeroize decrypted buffer
11 return rec
```

---

**Theorem 12.1** (Encrypt-then-MAC Security). *Let  $\mathcal{E}$  be AES-128-CBC (IND-CPA secure) and  $\mathcal{M}$  be HMAC-SHA256 (SUF-CMA secure). The Encrypt-then-MAC composition  $\Pi = (\mathcal{E}, \mathcal{M})$  achieves both IND-CPA confidentiality and INT-CTXT ciphertext integrity, provided the encryption and MAC keys are independent (guaranteed by domain-separated derivation, Algorithm 23).*

The decrypt buffer is a static kernel variable, safe because VaultOS is single-threaded. All plaintext buffers are zeroed after use to limit the window of exposure in physical memory.

*Remark 12.2* (Single-Threaded Precondition). The use of a static decrypt buffer in Algorithm 25 relies on the invariant that VaultOS never preempts a thread while a database operation is in progress. If future versions introduce preemptive multitasking or kernel threads, the decrypt pipeline must be modified to use per-context buffers (e.g., allocated on the caller's stack or from a per-CPU slab), or protected by a spinlock. Violating this precondition would allow a concurrent query to overwrite a partially-filled plaintext buffer, leading to data corruption or information leakage across security domains.

## 12.5 Constant-Time Audit Logging

The audit subsystem writes to `AuditTable` for every INSERT, DELETE, and UPDATE operation. Naive string handling leaks information about the action and result strings through execution timing. VaultOS pads all audit strings to the fixed maximum length before serialization:

---

**Algorithm 26:** AUDIT-LOG(pid, action, target\_id, result): Constant-time audit write.

---

```

1 padded_action[0..MAXSTR] ← 0MAXSTR+1
2 padded_result[0..MAXSTR] ← 0MAXSTR+1
3 MEMCPY(padded_action, action, min(|action|, MAXSTR))
4 MEMCPY(padded_result, result, min(|result|, MAXSTR))
5 rec ← NEW-RECORD(AUDITTABLE)
6 rec.fields ←
    [row_id, timestamp, pid, padded_action, target_id, padded_result]
7 RECORD-ENCRYPT(rec, TABLE_ID_AUDIT)
8 MEMSET-ZERO(padded_action, padded_result)

```

---

By always writing  $\text{MAXSTR} = 255$  bytes regardless of actual string length, the serialization, padding, encryption, and MAC computation all operate on identically sized inputs, eliminating timing side-channels in the audit path.

## 12.6 Query Execution Pipeline

With Encrypt-then-MAC active, the query pipeline includes cryptographic operations at both ends:

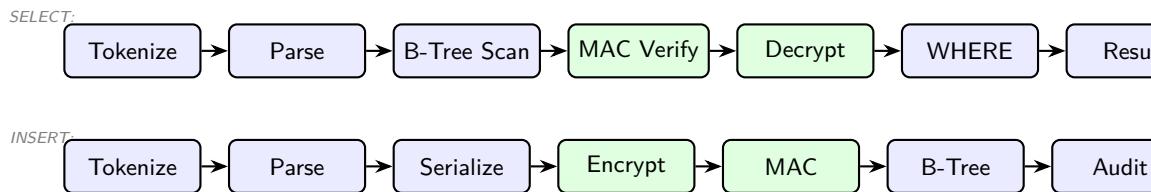


Figure 12.2: Query execution pipelines for SELECT and INSERT. Green nodes indicate cryptographic operations. UPDATE combines both paths: decrypt→modify→re-encrypt with a fresh IV.

For UPDATE queries, the pipeline decrypts matching records, applies field modifications in plaintext, then re-encrypts with a fresh random IV and recomputed MAC. This ensures that updating a record produces completely different ciphertext even if the plaintext change is minimal (semantic security).



# 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> [&ndlt;cond>]*
<cond>     ::= <ident> <op> <value>
<op>       ::= '=' | '!=> | '<' | '>' | '<=' | '>='
```

*Remark 13.1 (No DDL Support).* VaultOS does not support Data Definition Language statements (CREATE TABLE, DROP TABLE, ALTER TABLE). The six system tables are defined at compile time in `db_init_system_tables()` and cannot be created or destroyed at runtime. This is by design: the table schemas are part of the kernel's security invariants, and allowing runtime schema modification would undermine the static guarantees provided by the capability system.

### 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 13.2** (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).

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

### 14.2 Process State Transitions

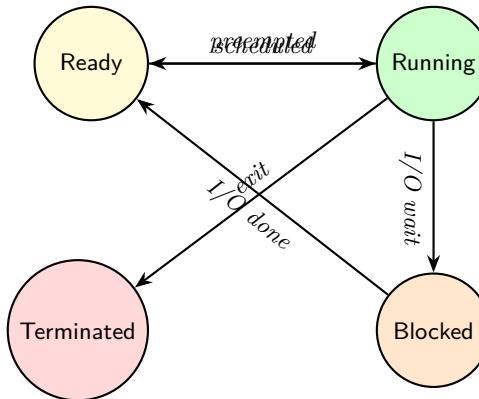


Figure 14.1: Process state transition diagram.

### 14.3 Algorithm: Round-Robin-Schedule

---

**Algorithm 27:** 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).

---

```
1 context_switch:  
2     ; Save old context (rdi = &old_ctx)  
3     mov [rdi+0x00], rbx  
4     mov [rdi+0x08], rbp  
5     mov [rdi+0x10], r12  
6     ; ... save r13-r15, rsp, rip, cr3 ...  
7  
8     ; Load new context (rsi = &new_ctx)  
9     mov cr3, [rsi+0x48]          ; switch page tables  
10    mov rbx, [rsi+0x00]  
11    mov rbp, [rsi+0x08]  
12    mov rsp, [rsi+0x38]  
13    ; ... restore r12-r15 ...  
14    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 28:** IPC-SEND(src, dst, type, payload, len): Send a message.

---

```
1 next_head ← (head + 1) mod C
2 if next_head = tail then
3   | return FULL
4 queue[head] ← (+ msg_id, src, dst, type, payload, now)
5 head ← next_head
6 return OK
```

---

---

**Algorithm 29:** 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  $8 \times 16$  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  $8 \times 16$  font, the text grid has  $\lfloor W/8 \rfloor$  columns and  $\lfloor H/16 \rfloor$  rows. At  $1024 \times 768$ : 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 PS/2 Mouse

The mouse driver handles IRQ12 via the 8042 PS/2 controller (data port 0x60, command port 0x64). Initialization enables the auxiliary port, sets sample rate to 100 Hz, and enables data reporting. Each mouse event produces a 3-byte packet:

---

**Byte    Contents**

---

0	Flags: buttons (bits 0–2), sign bits (4–5), overflow (6–7)
1	$\Delta x$ (9-bit signed, sign in byte 0 bit 4)
2	$\Delta y$ (9-bit signed, sign in byte 0 bit 5)

---

The driver accumulates bytes in a 3-byte state machine (triggered by the byte 0 alignment bit 3) and pushes completed packets into a ring buffer. The GUI event loop calls `mouse_poll()` to dequeue packets and update cursor coordinates, clamped to screen bounds.

## 17.5 8×16 Bitmap Font

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

# **Part VIII**

# **User Interface**



# Chapter 18

## VaultShell

VaultShell is the primary user interface of VAULTOS: a text-mode shell that accepts SQL queries and friendly commands, with syntax highlighting, tab completion, command history, and a structured TUI layout.

### 18.1 Text User Interface

The shell screen is divided into four regions (Figure 18.1):

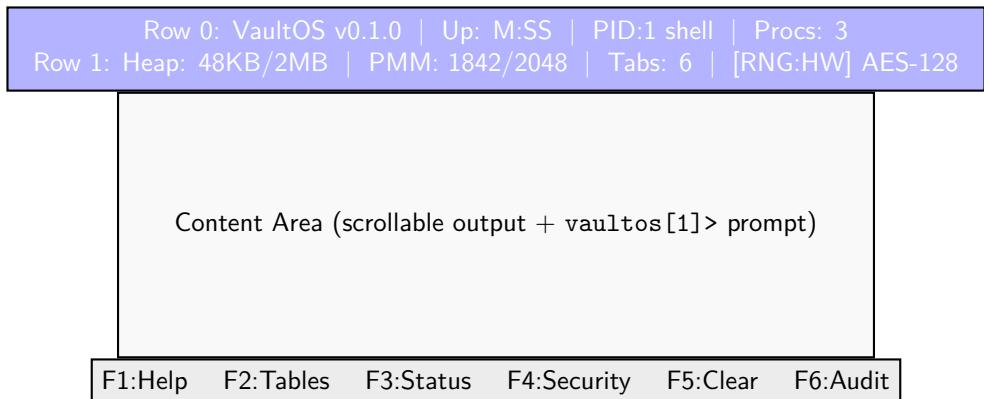


Figure 18.1: VaultShell TUI layout. The two-row status bar at the top shows system state including PID, process count, PMM pages, RNG source, and encryption mode. The prompt includes the shell's PID in brackets.

The TUI layout is computed from the framebuffer dimensions: a two-row status bar occupies rows 0–1, the F-key bar occupies the last row, and the content area fills all remaining rows. The status bar displays live metrics including the current PID, active process count, PMM free/total pages, RNG source (hardware or software), and encryption mode. The prompt shows the shell's PID: `vaultos[1]>`. F4 invokes the `security` command and F6 displays the audit log.

## 18.2 Line Editor

The line editor provides in-place editing with visual feedback:

- **Cursor movement:** Left/Right, Home/End.
- **Editing:** character insert at cursor, Backspace, Delete, Escape (clear).
- **History:** Up/Down arrows navigate a 32-entry ring buffer of previous commands. The current input is saved before entering history mode and restored when navigating past the most recent entry.
- **Syntax highlighting:** each character is colored in real time:
  - SQL keywords and friendly verbs: highlight color (gold).
  - Table aliases (`procs`, `caps`, etc.): cyan.
  - String literals (single-quoted): green.
  - Numbers: cyan. Operators (`=`, `<`, `>`, `*`): yellow.
- **Block cursor:** the character at the cursor position is rendered with inverted foreground/background colors.

## 18.3 Display Formatter

Query results are rendered as colored columnar tables. The formatter handles three cases: `SHOW TABLES` (table list with row counts), `DESCRIBE` (column schema with types and flags), and general `SELECT` results (aligned columns with type-aware formatting).

Errors are displayed with the symbolic error code name (via `vos_strerror()`) and a contextual hint—e.g., a permission error suggests “Check capabilities with `show caps`”. Success messages are styled by operation type: `GRANT/REVOKE` in gold, `INSERT/UPDATE` in green, and `DELETE` in cyan.

## 18.4 Algorithm: Context-Aware Tab Completion

Tab completion analyzes the tokens before the cursor to determine context.

---

**Algorithm 30:** COMPLETE-FIND(line, cursor\_pos): context-aware completion.

---

```
1  $w \leftarrow$  extract word fragment before cursor_pos
2 if  $|w| = 0$  then
3   return  $\emptyset$ 
4 tokens[]  $\leftarrow$  tokenize line up to start of  $w$ 
      // Context: verb + table  $\Rightarrow$  complete column names
5 if  $|tokens| \geq 2$  and is_friendly_verb(tokens[0]) and  $w$  has no = then
6   schema  $\leftarrow$  resolve_table(tokens[1])
7   if schema  $\neq$  nil then
8     for each column  $c$  in schema do
9       if  $c.name$  starts with  $w$  (case-insensitive) then
10         add_match( $c.name + "="$ )
11   if matches  $\neq \emptyset$  then
12     return matches
      // Context: "create"  $\Rightarrow$  complete object types
13 if  $|tokens| = 1$  and tokens[0] = "create" then
14   match  $w$  against {note, file, config, script, key}
      // Context: "spawn"  $\Rightarrow$  complete program names
15 if  $|tokens| = 1$  and tokens[0] = "spawn" then
16   match  $w$  against builtin programs  $\cup$  {"script:"}
      // Context: verb alone  $\Rightarrow$  complete table names + aliases
17 if  $|tokens| = 1$  and is_friendly_verb(tokens[0]) then
18   match  $w$  against table names  $\cup$  aliases
      // Default: match against all keywords, verbs, tables,
      //           aliases
19 match  $w$  against SQL keywords  $\cup$  friendly verbs  $\cup$  table names  $\cup$  aliases
20 compute_common_prefix(matches)
21 return matches
```

---

When a single match is found, the remaining characters plus a trailing space are inserted. When multiple matches exist, the longest common prefix is inserted and all candidates are displayed below the input line in cyan on dark gray.



# Chapter 19

## Friendly Command Layer

The friendly command layer provides a natural-language-inspired interface that translates user commands into SQL queries, eliminating the need to know SQL syntax for common operations.

### 19.1 Design: Commands as SQL Generators

**Definition 19.1** (Friendly Command). A friendly command  $F(v, t, \text{args})$  where  $v$  is a verb,  $t$  is a table reference, and  $\text{args}$  is a (possibly empty) sequence of key=value pairs, is a function that produces a SQL query string  $Q$  such that  $\text{db\_execute}(Q, pid)$  achieves the intended semantics of  $F$ .

This design preserves a critical invariant: *the kernel database engine is never modified*. The friendly layer is purely a shell-side string transformation, ensuring that all security properties (capability checks, encryption, audit logging) remain intact.

### 19.2 Command-to-SQL Translation

Table 19.1 lists the friendly verbs and their SQL translations.

### 19.3 Table Aliases

To reduce typing, the alias system maps short names to full table names:

Alias	Table	Alias	Table	Alias	Table
procs	ProcessTable	caps	CapabilityTable	objects	ObjectTable
msgs	MessageTable	audit	AuditTable	sys	SystemTable
config	SystemTable				

Table 19.1: Friendly commands and their SQL translations.

Command	SQL Generated	Notes
tables	SHOW TABLES	
show <table>	SELECT * FROM <T>	Aliases
info <table>	DESCRIBE <T>	
find <T> col=val	SELECT * FROM <T> WHERE col = val	Multicolumn
count <T>	SELECT * FROM <T>	Display
add <T> c=v c=v	INSERT INTO <T> (...) VALUES (...)	
del <T> col=val	DELETE FROM <T> WHERE col = val	
set <T> c=v where k=v	UPDATE <T> SET c=v WHERE k=v	
create <type> <name> [data]	INSERT INTO ObjectTable ...	
open <name>	SELECT * FROM ObjectTable WHERE name='...'	
list [type]	SELECT * FROM ObjectTable [WHERE type='...']	
rm <name>	DELETE FROM ObjectTable WHERE name='...'	
ps	SELECT * FROM ProcessTable	

Aliases are resolved before SQL generation by `friendly_resolve_alias()`, allowing commands like `show procs` instead of `SELECT * FROM ProcessTable`. Aliases are also syntax-highlighted in cyan in the line editor.

## 19.4 Script Engine

Scripts are sequences of commands stored as rows in the ObjectTable. Each line is a separate row with `type = 'script'` and the script name in the `name` field. Lines are ordered by `obj_id` (monotonically increasing from sequential inserts).

*Observation 19.2* (Script Line Ordering). The `size` field in ObjectTable cannot be used for line ordering because the query engine (line 479 of `query.c`) unconditionally overwrites it with the length of `data`. Sequential inserts produce monotonically increasing `obj_id` values in the B-tree, and B-TREE-SCAN returns keys in order, guaranteeing correct line ordering without an explicit sequence number.

---

**Algorithm 31:** SCRIPT-SAVE(name): interactive multi-line script recording.

---

```

1 DELETE FROM ObjectTable WHERE name = name AND type =
  'script'
2 line_num ← 0
3 while line_num < MAX_SCRIPT_LINES do
4   print "  N> "
5   line ← LINE-READ()
6   if line = "end" then
7     break
8   if line is empty then
9     continue
10  INSERT INTO ObjectTable (name, type, data) VALUES (name,
    'script', line)
11  line_num ← line_num + 1

```

---

**Algorithm 32:** SCRIPT-RUN(name): execute a stored script.

---

```

1 rows ← SELECT * FROM ObjectTable WHERE name = name AND type
  = 'script'
2 if rows is empty then
3   error "Script not found"
4 for each row r in rows (ordered by obj_id) do
5   cmd ← r.data
6   if FRIENDLY-TRANSLATE(cmd) succeeds then
7     sql ← translated result
8   else
9     sql ← cmd // pass through as raw SQL
10  result ← db_execute(sql, pid)
11  display result

```

---

## 19.5 Process Management Commands

Process management commands are built-in shell operations that invoke kernel process APIs directly, since they require function pointer arguments that cannot be expressed as SQL.

**spawn.** The `spawn` command creates a new process from a registry of built-in programs or from a stored script. For scripts, the entry point reads the script name from the process name field (`process_t.name`), avoiding global state and race conditions:

Listing 19.1: Script process entry point.

```
1 void script_process_entry(void) {
2     process_t *self = process_get_current();
3     const char *name = self->name + 7; /* skip "script:" */
4     */
5     script_run(name);
6     process_exit(self, 0);
7     for (;;) hlt();
}
```

**kill.** Terminates a process by PID. The shell process is protected: attempting to kill the shell's own PID is rejected with an error message.

**msg / inbox.** `msg <pid> <text>` inserts a row into `MessageTable` with the destination PID and payload. `inbox` queries `MessageTable` for messages addressed to the current process.

**monitor.** The `monitor` built-in program is a background process that runs in an infinite loop, inserting a system stats record (uptime, heap usage) into `AuditTable` every 5 seconds. It demonstrates preemptive multitasking and database access from concurrent processes.

## 19.6 Algorithm: Friendly-Translate

---

**Algorithm 33:** FRIENDLY-TRANSLATE(*input*, *sql\_buf*): convert a friendly command to SQL.

---

```
1 verb ← first token of input
2 rest ← remaining tokens
3 if verb = “tables” then
4   | sql_buf ← “SHOW TABLES”
5 else if verb = “show” then
6   | t ← resolve_alias(next token)
7   | sql_buf ← “SELECT * FROM t”
8 else if verb = “find” then
9   | t ← resolve_alias(next token)
10  | parse key=value pairs from rest into WHERE clauses with AND
11  | sql_buf ← “SELECT * FROM t WHERE ...”
12 else if verb = “add” then
13   | t ← resolve_alias(next token)
14   | parse key=value pairs into column and value lists
15   | sql_buf ← “INSERT INTO t (...) VALUES (...)”
16 else if verb ∈ {“create”, “open”, “list”, “rm”, “ps”, ...} then
17   | generate appropriate SQL for ObjectTable or ProcessTable
18 else
19   | return false // not a friendly command
20 return true
```

---



# Chapter 20

## Graphical Desktop

VAULTOS includes a graphical desktop environment built on top of the GOP framebuffer. The GUI provides windowed applications for database queries, table browsing, process management, and system monitoring. The user can switch between TUI and GUI modes with the `gui` command.

### 20.1 Graphics Subsystem

The graphics layer implements double buffering over the GOP linear framebuffer:

- **Back buffer:** a heap-allocated buffer of  $W \times H \times 4$  bytes (32-bit ARGB). All drawing operations target this buffer.
- **Flip:** `gfx_flip()` copies the back buffer to the framebuffer in a single `memcpy`. Partial flips (`gfx_flip_rect`) copy only the dirty region to minimize bandwidth.

Primitive operations include `gfx_fill_rect()`, `gfx_draw_rect()` (outline), `gfx_draw_hline()`, `gfx_draw_text()` (using the 8×16 bitmap font), and pixel-level access via `gfx_putpixel()`.

**Definition 20.1** (Double Buffer Memory). For a  $1024 \times 768$  display at 32 bpp, each buffer requires  $1024 \times 768 \times 4 = 3,145,728$  bytes ( $\approx 3$  MiB). The heap is expanded by 15 MiB upon GUI launch to accommodate the back buffer, window canvases, and widget data.

The back buffer is allocated via `kmalloc()` from the kernel heap (Chapter 5), not from a dedicated video memory pool. For the default  $1024 \times 768$  resolution this consumes  $\approx 3$  MiB of the 15 MiB GUI heap expansion. The remaining  $\approx 12$  MiB is available for window canvases, widget trees, and compositor state.

## 20.2 Event System

The event system translates hardware inputs into a unified event queue:

Event Type	Source
EVT_MOUSE_DOWN/UP/MOVE	PS/2 mouse packets
EVT_KEY_PRESS/RELEASE	PS/2 keyboard scancodes
EVT_CLOSE	Window close button click

`event_pump()` polls the mouse and keyboard drivers non-blockingly and enqueues events into a fixed-size ring buffer. The main loop calls `event_poll()` to dequeue events for dispatch.

## 20.3 Window Manager

The window manager maintains a z-ordered list of up to 16 simultaneous windows. Each window has:

Listing 20.1: Window structure (simplified).

```
1 typedef struct window {
2     uint32_t id;
3     char title[64];
4     int16_t x, y; /* screen position */
5     uint16_t w, h; /* total size */
6     uint16_t client_w, client_h; /* client area */
7     uint32_t *canvas; /* pixel buffer */
8     bool focused, dragging;
9     bool minimized; /* hidden but in
10    taskbar */
11     widget_t *widgets; /* widget linked list
12    */
13 } window_t;
```

Window operations include:

- **Focus:** clicking a window brings it to front (z-order head).
- **Drag:** clicking the title bar and moving the mouse repositions the window.
- **Minimize:** clicking the “\_” button hides the window; clicking its taskbar button restores it. Minimized windows appear with dimmed text in the taskbar.

- **Close:** clicking the “X” button sends `EVT_CLOSE` to the window’s event handler, which calls `wm_destroy_window()`.
- **Decorations:** the window manager draws a title bar (20 px), 1 px border, minimize button, and close button. The client area is below the title bar.

## 20.4 Compositor

The compositor renders the desktop each frame:

---

**Algorithm 34:** `COMPOSITOR-RENDER()`: render one frame of the desktop.

---

```
1 clear back buffer to desktop background color
2 for each window w in bottom-to-top z-order do
3   draw w's border and title bar decorations
4   call w.on_paint() to render widgets into w's canvas
5   blit w's canvas to back buffer at (w.x, w.y)
6 draw mouse cursor (white fill, black outline)
7 gfx_flip_rect(dirty region)
```

---

The cursor is a  $12 \times 19$  bitmap with a black outline and white fill, drawn on top of all windows. The compositor also renders the taskbar (28 px at screen bottom) with security badges (encryption and RNG status) and the start menu when open. The main loop runs at approximately 60 iterations per second (limited by `hlt()` idle waits).

## 20.5 Widget Toolkit

Four widget types are provided:

---

Widget	Behavior
Button	Click callback, hover highlight, text label
Label	Static text with configurable foreground/background
TextBox	Single-line text input with cursor, character insert/delete
ListView	Scrollable item list with selection highlight, click-to-select, optional <code>on_select</code> callback

---

Widgets are stored as a linked list per window (static pool of 128 slots). `widget_dispatch()` routes mouse and keyboard events to the appropriate widget based on hit testing. `widget_draw_all()` renders all widgets into a window’s canvas using three helper

macros: `CPIX` (bounds-checked pixel write), `CHLINE` (horizontal line), and `CTEXT` (text rendering via the  $8 \times 16$  font bitmap).

Applications that require custom rendering beyond standard widgets (e.g., the text editor, data grid, calculator, hex viewer) define these same macros locally in their paint handlers and draw directly to `win->canvas`.

## 20.6 Desktop Applications

The GUI provides twelve applications organized into a start menu with category separators. All applications are defined in `desktop.c` using static data buffers, event handlers, and paint callbacks.

### 20.6.1 Database Tools

**Query Console.** A SQL REPL with a textbox for query input, “Execute” and “Template” buttons, and a listview for results. Templates cycle through common queries (`SHOW TABLES`, `SELECT * FROM SystemTable`, etc.).

**Table Browser.** A two-pane interface: the left pane lists all system tables (with Refresh), and the right pane shows either the table schema (click a table name) or query results (“View All” button). A search box supports `column=value` filtering.

**Data Grid.** A spreadsheet-like table viewer with custom canvas rendering. The left pane contains a table selector listview; the right pane renders a grid with column headers in gold, row numbers, and type-colored cells: numbers in light blue (0xFF66CCFF), strings in teal (0xFF00DDAA), and booleans in gold (0xFFFFCC00). Rows alternate between two background shades, and the selected row is highlighted. Column widths are auto-calculated from header name length. Supports keyboard navigation (Up/Down, PgUp/PgDn) and mouse click-to-select.

### 20.6.2 Productivity Applications

**VaultPad Editor.** A multi-line plain-text editor (up to 128 lines  $\times$  255 characters per line). Because the `TextBox` widget is single-line only, the editing area is entirely custom-rendered in the paint handler using the `CPIX/CTEXT` canvas macros. Features include:

- Cursor navigation: arrow keys, Home/End, PgUp/PgDn.
- Line editing: character insert, Enter (split line), Backspace (join lines), Delete (forward delete), Tab (4 spaces).

- Line numbers in a left gutter with a separator line.
- Gold 2-pixel cursor bar on the current position.
- Status bar showing line number, column, modified/saved state, and line count.
- Keyboard shortcuts: Ctrl+S (save), Ctrl+Q (close), Ctrl+N (new document).
- Mouse click to position cursor in the editing area.
- Open dialog with a listview showing all saved documents.

Documents are stored in ObjectTable using the same multi-row pattern as scripts: one row per line with `type='document'` and a shared `name` field. Saving deletes all existing rows for that document name and re-inserts each line. Single quotes in content are escaped by doubling (' → ").

**Calculator.** An integer-only calculator (`int64_t` arithmetic) with zero widget usage—the entire interface (display area and  $4 \times 5$  button grid) is custom-rendered in the paint handler. Button colors are context-dependent: digits in dark blue, operators in gold, Clear (C) in red, and equals (=) in teal. Supports both keyboard input (0–9, +-\*//, Enter, Backspace, C) and mouse click on the grid via coordinate hit-testing. Division by zero displays an error message.

**Object Inspector.** A dual-pane object viewer. The left pane shows a filterable listview of all ObjectTable entries (formatted as `[type] name`). Selecting an object displays:

- **Metadata section:** ID, Name, Type, Owner PID, and creation timestamp with gold headers and teal values.
- **Hex dump:** traditional format with 8 bytes per row—offset in gold, hex bytes in teal, and ASCII representation (printable characters or ‘.’ for non-printable). PgUp/PgDn scrolls the hex view.

### 20.6.3 Security Applications

**Security Dashboard.** A  $500 \times 380$  window showing encryption status (all six system tables marked `[SEALED]`), the encryption algorithm (AES-128-CBC), HMAC-SHA256 key derivation, active capability token count, RNG source (hardware/software), and audit event count. A “View Audit Log” button opens the Audit Log viewer.

**Audit Log Viewer.** Displays all entries from AuditTable with a text filter, Refresh button, and a listview. Each entry is formatted as [MM:SS] ACTION PID:N result.

**Capability Manager.** Lists all capability tokens with decoded rights (Read, Write, eXec, Delete, Grant, reVoke) and status (active/REVOKEd in red). A “Revoke” button executes REVOKE `cap_id` via SQL.

**Object Manager.** Lists all objects in ObjectTable with type, name, and data preview. A “Delete” button removes the selected object via SQL DELETE.

#### 20.6.4 System Tools

**Process Manager.** Lists all processes from ProcessTable with Refresh, “Spawn Monitor” (creates a background monitoring process), and “Kill” buttons. The Kill button protects the shell process from self-termination.

**System Status.** A  $420 \times 340$  window displaying live metrics: uptime, heap usage (used/free), PMM pages (free/total), table count, active processes, capability count, RNG source, encryption mode, and CPU architecture.

#### 20.6.5 Taskbar and Start Menu

**Taskbar.** The taskbar occupies the bottom 28 pixels: a “VaultOS” menu button on the left, window buttons in the center (with dimmed text for minimized windows), and security badges plus a status line on the right. The security badges show “ENC” in green (always active) and “RNG:HW” or “RNG:SW” colored green or yellow depending on `random_hw_available()`.

**Start Menu.** The start menu contains 12 application entries organized with separator lines:

- 
- Query Console / Table Browser / Data Grid
  - VaultPad Editor / Calculator / Object Inspector
  - Security Dashboard / Audit Log / Capability Manager / Object Manager
  - Process Manager / System Status
  - Exit to TUI
-

## Appendix A

# Virtual Address Space Map

Virtual Address Range	Purpose
0x0000000000000000 – 0x00007FFFFFFE000	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
0xFFFFFFFFFC0000000 – 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_REVOKER	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.



## Appendix E

# VaultShell Command Reference

Category	Command	Description
<i>Query Commands</i>		
	<code>tables</code>	List all system tables
	<code>show &lt;table&gt;</code>	Display all rows
	<code>info &lt;table&gt;</code>	Show table schema
	<code>find &lt;T&gt; col=val</code>	Search rows
	<code>count &lt;table&gt;</code>	Count rows
<i>Mutation Commands</i>		
	<code>add &lt;T&gt; col=val ...</code>	Insert a new row
	<code>del &lt;T&gt; col=val</code>	Delete matching rows
	<code>set &lt;T&gt; c=v where k=v</code>	Update matching rows
<i>Object Commands</i>		
	<code>create &lt;type&gt; &lt;name&gt;</code>	Create an object
	<code>open &lt;name&gt;</code>	View an object
	<code>list [type]</code>	List objects
	<code>rm &lt;name&gt;</code>	Delete an object
	<code>cat &lt;name&gt;</code>	Display contents
<i>Script Commands</i>		
	<code>save &lt;name&gt;</code>	Record a script
	<code>run &lt;name&gt;</code>	Execute a script
	<code>scripts</code>	List all scripts
<i>Process Commands</i>		
	<code>ps</code>	List processes
	<code>spawn &lt;program&gt;</code>	Launch a program
	<code>spawn script:&lt;name&gt;</code>	Run script as process
	<code>kill &lt;pid&gt;</code>	Terminate a process
	<code>msg &lt;pid&gt; &lt;text&gt;</code>	Send a message
	<code>inbox</code>	View messages
<i>Security &amp; Diagnostics</i>		
	<code>security / crypto</code>	Show full crypto dashboard
	<code>audit [N]</code>	Show last N audit entries (default 20)
<i>System &amp; SQL Commands</i>		
	<code>help / status / clear</code>	Built-in utilities
	<code>gui</code>	Launch GUI desktop
	<code>SELECT / INSERT / ...</code>	Raw SQL pass-through
	<code>GRANT / REVOKE</code>	Capability management
<i>F-Key Shortcuts</i>		
	<code>F1</code>	Help
	<code>F2</code>	Tables
	<code>F3</code>	Status
	<code>F4</code>	Security dashboard
	<code>F5</code>	Clear screen
	<code>F6</code>	Audit log