Low-Level Programming on Linux (x86-64)

Low-Level Cryptography

redhat

debian

8

Prepared by Ayman Alheraki

https://simplifycpp.org

Low-Level Programming on Linux (x86-64) :

Low-Level Cryptography

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Contents 2

Author's Introduction 7

1 Data Representation 13
1.1 Endianness . . . 13
1.1.1 Conceptual Definition . . . 14
1.1.2 Detecting Endianness at Runtime in Pure Assembly . . . 14
1.1.3 Endianness and Cryptographic Blocks . . . 15
1.1.4 Byte-Swap Primitives on x86-64 . . . 16
1.1.5 Manual Byte Reversal Using Shifts . . . 16
1.1.6 Explicit Endianness in Message Blocks . . . 17
1.1.7 Endianness in Multi-Precision Arithmetic . . . 17
1.1.8 Conclusion . . . 18
1.2 Word Packing and Unpacking . . . 19
1.2.1 Packing Big-Endian Words . . . 19
1.2.2 Unpacking Big-Endian Words . . . 19
1.2.3 SHA-256 Message Schedule Packing . . . 19
1.2.4 Conclusion . . . 20
1.3 Memory Alignment . . . 21

1.3.1 Alignment Requirements 21
1.3.2 Alignment with Assembler Directives 21
1.3.3 Runtime Alignment Verification 21
1.3.4 Stack Alignment 22
1.3.5 Conclusion 22

2 Primitive Operations 23
2.1 XOR & Bitwise Logic 23
2.1.1 XOR as a Fundamental Cryptographic Operator 24
2.1.2 XOR with Memory and Streaming State 24
2.1.3 AND, OR, and NOT: Boolean Control 25
2.1.4 Bit-Sliced Boolean Composition 26
2.1.5 Boolean Functions in SHA-256 27
2.1.6 Constant-Time Behavior of Boolean Logic 28
2.1.7 Streaming XOR Loop 28
2.1.8 Conclusion 29
2.2 Shift vs Rotate 30
2.2.1 Logical and Arithmetic Shifts 30
2.2.2 Rotations 31
2.2.3 Why Rotations Dominate ARX Designs 31
2.2.4 Portable Rotate Construction (Educational) 31
2.2.5 SHA-256 Shift-Based Functions 32
2.2.6 Rotate-Based Diffusion 33
2.2.7 Conclusion 33
2.3 Constant-Time Operations 34
2.3.1 Constant-Time Selection 34
2.3.2 Constant-Time Comparison 34
2.3.3 Constant-Time Mask Generation 35

2.3.4 Constant-Time Swap . . . 35
2.3.5 Constant-Time Zeroization . . . 36
2.3.6 Conclusion . . . 36

3 SHA-1 Implementation 37
3.1 Message Scheduling . . . 37
3.1.1 Loading the Initial 16 Words . . . 38
3.1.2 Expanding Words W[16] Through W[79] . . . 38
3.1.3 Combined Scheduler . . . 39
3.1.4 Constant-Time Properties . . . 40
3.1.5 Conclusion . . . 41
3.2 Compression Rounds . . . 42
3.2.1 Working Registers . . . 42
3.2.2 Boolean Functions . . . 42
3.2.3 Round Constants . . . 42
3.2.4 Compression Loop Skeleton . . . 43
3.2.5 Conclusion . . . 46
3.3 Digest Finalization . . . 47
3.3.1 Padding . . . 47
3.3.2 Digest Serialization . . . 48
3.3.3 Conclusion . . . 48

4 HMAC-SHA1 50
4.1 Inner and Outer Keys . . . 50
4.1.1 Key Normalization and Block Preparation . . . 51
4.1.2 Generating K_ipad (Inner Key Block) . . . 51
4.1.3 Generating K_opad (Outer Key Block) . . . 52
4.1.4 Constant-Time Properties . . . 53

4.1.5 Integrated Inner and Outer Key Generation 53
4.1.6 Conclusion 54
4.2 Block Padding 55
4.2.1 Padding Rules 55
4.2.2 Constant-Time Padding Routine 55
4.2.3 Conclusion 56
4.3 Test Vector Verification 57
4.3.1 Constant-Time Digest Comparison 57
4.3.2 Conclusion 58

5 ChaCha20 59
5.1 State Layout 59
5.1.1 The 4×4 ChaCha20 State Matrix 59
5.1.2 Encoding the Constants 60
5.1.3 Memory Layout of the State Buffer 61
5.1.4 State Initialization in Assembly 61
5.1.5 Endianness 63
5.1.6 Conclusion 63
5.2 Quarter Round Function 64
5.2.1 Quarter Round Definition 64
5.2.2 Register Mapping 64
5.2.3 Quarter Round in Assembly 64
5.2.4 Why These Operations 65
5.2.5 Conclusion 65
5.3 Block Output Generation 66
5.3.1 Feed-Forward Step 66
5.3.2 Feed-Forward Assembly 66
5.3.3 Keystream Semantics 67

5.3.4 Conclusion 67

6 Final Project 68
6.1 A Complete Cryptography Assembly Library 68
6.1.1 Architectural Philosophy 69
6.1.2 Directory-Level Organization 70
6.1.3 ABI, Calling Convention, and Register Contract 71
6.1.4 Digest Layer (SHA-1) 72
6.1.5 HMAC Layer (HMAC-SHA1) 73
6.1.6 ChaCha20 Stream Cipher Layer 73
6.1.7 Constant-Time Utility Layer 74
6.1.8 Secure Memory Handling 74
6.1.9 Cross-Primitive Guarantees 75
6.1.10 Example: Encryption + Authentication Pipeline 75
6.1.11 Unified Memory Layout 75
6.1.12 Conclusion 76

Conclusion 77

References 80

Author's Introduction

Cryptography is often perceived as a discipline dominated by mathematical abstraction and high-level formalism. In practice, however, the security and correctness of cryptographic systems ultimately depend on the exact machine instructions that execute them. This volume, Low-Level Cryptography, focuses on the level where cryptography becomes concrete: CPU registers, status flags, memory alignment, instruction latency, bitwise transformations, and deterministic state evolution.

The primary objective of this book is to demonstrate, step by step, how modern symmetric cryptographic primitives can be implemented directly in x86-64 assembly for Linux, using the GNU assembler (GAS) with Intel syntax. All implementations are written without reliance on compilers, external libraries, language runtimes, or abstractions that obscure the true execution behavior of the code. Every instruction shown is explicit, analyzable, and under full programmer control.

The material reflects the post-2020 understanding of secure software design, where cryptographic correctness cannot be separated from constant-time execution, microarchitectural predictability, and strict control over memory access patterns. Each chapter adheres to the principle that secure cryptography must avoid secret-dependent branching, data-indexed memory access, speculative side effects, or any form of uncontrolled control flow. All assembly listings follow GAS conventions, including Intel operand order and the use of # for comments, to ensure correctness and consistency with the Linux toolchain.

The cryptographic primitives covered in this book—SHA-1, HMAC-SHA1, and ChaCha20—are not selected solely for their contemporary strength, but because their internal structures lend themselves naturally to instruction-level analysis. Their reliance on ARX (Add-Rotate-XOR) operations and feed-forward designs allows the reader to observe how secure bitwise transformations are realized concretely within a modern CPU pipeline.

Throughout this volume, cryptographic state is represented explicitly through register-mapped variables, 32-bit words, and statically or stack-allocated aligned buffers under direct programmer control. Endianness is handled manually through register-level manipulation, ensuring that every serialized digest, message block, and keystream word accurately reflects the algorithmic specification. Message scheduling, state expansion, compression rounds, and keystream generation are all implemented as transparent assembly routines whose behavior can be followed instruction by instruction.

A low-level cryptographic programmer must understand not only the algorithmic steps of a primitive, but also how those steps interact with instruction pipelines, register dependencies, memory alignment constraints, and execution latency. The ChaCha20 quarter-round, for example, illustrates the efficiency of ARX ciphers on x86-64 hardware, while the HMAC construction demonstrates how domain separation and length binding must be preserved even when implemented manually in assembly. Similarly, SHA-1's message schedule expansion and compression rounds provide a clear example of structured bit diffusion using XOR operations, rotations, and disciplined register movement.

This book treats cryptographic implementation as a deterministic engineering discipline. All code operates without libc, without compiler intrinsics, and without dynamic memory allocation. Safety is achieved not through abstraction, but through explicit construction: constant-time comparison loops, controlled zeroization routines, aligned buffers, and manually constructed stack frames form the foundation upon which

the cryptographic logic is built. These practices are not optional; they are required to eliminate side-channel leakage, undefined behavior, and unintended toolchain interference.

As part of the broader Low-Level Programming on Linux (x86-64) series, this volume bridges the gap between cryptographic theory and complete system-level implementation. By the end of the book, the reader will not only understand the internal design of key cryptographic primitives, but will also possess the practical knowledge required to build a minimal, robust cryptographic assembly library suitable for controlled environments, bare-metal experimentation, security research, and performance-critical systems.

This book is written with the conviction that cryptography is strongest when its operation is fully visible. Implementing cryptographic algorithms directly in assembly exposes every transformation, every state transition, and every timing decision. In an era increasingly dependent on secure computation, this level of understanding is not a luxury—it is a requirement.

Stay Connected

For more discussions and valuable content about Low-Level Cryptography, I invite you to follow me on LinkedIn:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Preface

Cryptography becomes trustworthy only when its execution is understood down to the level of individual machine instructions. In contemporary systems, cryptographic libraries are commonly written in high-level languages, linked through complex and opaque toolchains, and executed behind layers of abstraction that conceal the machine behavior on which their security ultimately depends. This book, Low-Level Cryptography, adopts the opposite approach. It presents cryptographic primitives exactly as they execute in registers, through arithmetic and bitwise instructions, and across explicitly aligned memory layouts on Linux x86-64 systems.

The objective of this volume is not merely to implement cryptographic algorithms, but to expose every transformation that secure computation performs at the lowest level accessible to software. All implementations are written directly in x86-64 assembly using the GNU assembler (GAS) with Intel syntax, following strict constant-time rules and explicit memory discipline. The use of assembly eliminates ambiguity introduced by compilers, optimizers, and runtime libraries, and ensures that every operation executed by the processor is visible, analyzable, and under direct programmer control.

The decision to use pure assembly is deliberate. It forces the developer to confront the reality that cryptographic strength depends not only on algorithmic correctness, but on correct execution. A hash function is only as secure as the way its message schedule is constructed. A stream cipher is only as reliable as the predictability of its addition, rotation, and XOR operations. A keyed hash construction is only as robust as

the enforced separation between its inner and outer domains, achieved without secret-dependent branching, speculative execution effects, or compiler intervention. Writing these routines directly in assembly removes uncertainty from these critical paths and allows precise control over timing behavior, memory movement, and side effects.

This book focuses on three representative primitives whose internal structures illustrate distinct cryptographic design philosophies. SHA-1 demonstrates feed-forward compression, word expansion, and big-endian serialization implemented through disciplined ARX-style operations. HMAC-SHA1 shows how domain separation, key normalization, and block padding must be enforced explicitly rather than assumed. ChaCha20 exemplifies a modern ARX stream cipher whose quarter-round and double-round constructions are naturally suited to x86-64 architectures, where rotations, XORs, and additions execute in constant time without lookup tables or unpredictable control flow.

Each primitive is constructed incrementally throughout the text: from raw state layout and register allocation, to round functions, to complete block processing, and finally to an integrated cryptographic library. Every transformation is expressed through explicit register usage, aligned buffers, and short, deterministic instruction sequences. These implementations make it possible to reason about behavior without guesswork, revealing how algorithmic structure translates directly into machine-level execution on real hardware.

The book also emphasizes the security implications inherent in low-level programming. Correct cryptographic implementation requires more than mathematical accuracy; it demands strict discipline in constant-time design, explicit zeroization of sensitive state, careful adherence to the System V ABI, and complete avoidance of data-dependent control flow. These constraints are reflected consistently in the assembly routines presented throughout the chapters. They are not optional optimizations, but foundational requirements for secure cryptographic software.

By presenting cryptography at the instruction level, the reader gains the ability to analyze, reason about, and verify cryptographic behavior directly from the processor's perspective. Understanding cryptography in assembly enables deeper insight into performance characteristics, side-channel resilience, and the true operational behavior of primitives as they execute within a modern Linux environment.

This volume is part of the Low-Level Programming on Linux (x86-64) series and builds upon the earlier focus on instruction semantics, memory layout, system calls, process models, linkers, and runtime mechanisms. Here, that foundation is extended into the domain of secure computation, demonstrating that cryptography can be engineered with the same rigor demanded in systems programming, toolchain construction, and kernel-level development.

When cryptography is implemented at the instruction level, it ceases to be an abstract concept. It becomes a precise sequence of additions, rotations, XOR operations, and state transitions flowing through the ALU. It becomes a set of aligned buffers placed intentionally in memory. And it becomes a discipline that tolerates no ambiguity in execution. This book aims to make that discipline clear, practical, and accessible to the low-level programmer who seeks complete control over how secure computation is performed on Linux x86-64 systems.

Chapter 1

Data Representation

1.1 Endianness

Endianness defines the byte ordering used to represent multi-byte values in memory. It determines how integers, pointers, and binary fields are encoded at the byte level, directly affecting hash function inputs, cipher state layouts, message parsing, and protocol-defined bit streams. In cryptographic software, endianness is not a peripheral detail; it is a fundamental part of the algorithm definition. Mishandling it produces incorrect digests, invalid MACs, broken key derivations, and incompatible ciphertext. Modern x86-64 processors operate natively in little-endian mode, meaning the least significant byte occupies the lowest memory address. Linux preserves this ordering across all ABI layers, including system calls, ELF relocation formats, stack layout, and pointer representation. However, many cryptographic standards—such as SHA-1, SHA-256, and big-number encodings used in RSA and ECC—define their data structures in big-endian order. This mismatch requires explicit conversion when implementing cryptography on x86-64.

Understanding and controlling endianness is therefore foundational for correct low-level

cryptographic engineering.

1.1.1 Conceptual Definition

Given the 32-bit value:

```
0x11223344
```

A little-endian machine stores it as:

```
44 33 22 11
```

A big-endian machine stores it as:

```
11 22 33 44
```

Registers hold logical values independent of byte order. Endianness becomes observable only when data crosses the register–memory boundary: loads, stores, DMA transfers, or I/O. Cryptographic algorithms operate precisely at these boundaries.

1.1.2 Detecting Endianness at Runtime in Pure Assembly

Although x86-64 is always little-endian, explicit detection logic is sometimes used for validation or portability testing.

```
.intel_syntax noprefix
.global _start

.section .bss
    .align 8
buf:
    .space 8
```

```
.text
_start:
    mov rax, 0x0102030405060708
    mov [buf], rax

    mov al, BYTE PTR [buf]   # read lowest-addressed byte
    cmp al, 0x08             # little-endian check
    jne big_endian

little_endian:
    mov eax, 60              # SYS_exit
    xor edi, edi             # exit code 0
    syscall

big_endian:
    mov eax, 60
    mov edi, 1               # exit code 1
    syscall
```

1.1.3 Endianness and Cryptographic Blocks

Cryptographic standards define byte order explicitly:

- AES processes state as byte-indexed matrices.
- SHA-256 defines message words as big-endian 32-bit integers.
- Curve25519 uses little-endian scalar representation.

The CPU does not reinterpret memory; the programmer must reorder bytes explicitly to satisfy the algorithm specification.

1.1.4 Byte-Swap Primitives on x86-64

x86-64 guarantees efficient byte-reversal instructions:

- bswap r32
- bswap r64

```
.intel_syntax noprefix
.global swap64

swap64:
    mov rax, rdi
    bswap rax
    ret
```

1.1.5 Manual Byte Reversal Using Shifts

In performance-critical paths, shifts and masks may be interleaved with rotations.

```
.intel_syntax noprefix
.global swap32_manual

swap32_manual:
    mov eax, edi

    mov ecx, eax
    shl eax, 24

    shr ecx, 8
    and ecx, 0x00FF0000
    or eax, ecx

    shr ecx, 8
```

```
    and ecx, 0x0000FF00
    or eax, ecx

    shr edi, 24
    or eax, edi
    ret
```

1.1.6 Explicit Endianness in Message Blocks

```
.intel_syntax noprefix
.global store_be32

store_be32:
    mov eax, edi
    bswap eax
    mov [rsi], eax
    ret
```

```
.intel_syntax noprefix
.global load_be32

load_be32:
    mov eax, [rdi]
    bswap eax
    ret
```

1.1.7 Endianness in Multi-Precision Arithmetic

Multi-precision integers are stored internally as little-endian limb arrays, while network formats are big-endian.

```
.intel_syntax noprefix
.global load_limbs_be
```

```
load_limbs_be:
    dec rdx
.loop:
    mov rcx, [rdi + rdx*8]
    bswap rcx
    mov [rsi + rdx*8], rcx
    dec rdx
    jns .loop
    ret
```

1.1.8 Conclusion

Endianness is a defining constraint of cryptographic correctness. On x86-64 Linux, disciplined byte-order handling is mandatory for producing interoperable, secure, and performant cryptographic implementations.

1.2 Word Packing and Unpacking

Cryptographic primitives operate on fixed-width words. Correct packing and unpacking between byte arrays and machine words is required for every block processed.

1.2.1 Packing Big-Endian Words

```
.intel_syntax noprefix
.global pack_be32

pack_be32:
    mov eax, [rdi]
    bswap eax
    ret
```

1.2.2 Unpacking Big-Endian Words

```
.intel_syntax noprefix
.global store_be64

store_be64:
    mov rax, rdi
    bswap rax
    mov [rsi], rax
    ret
```

1.2.3 SHA-256 Message Schedule Packing

```
.intel_syntax noprefix
.global sha256_pack16

sha256_pack16:
```

```
    xor ecx, ecx
.loop:
    mov eax, [rdi + rcx*4]
    bswap eax
    mov [rsi + rcx*4], eax
    inc ecx
    cmp ecx, 16
    jl .loop
    ret
```

1.2.4 Conclusion

Packing and unpacking operations define the correctness boundary between cryptographic specifications and machine execution.

1.3 Memory Alignment

Memory alignment determines how cryptographic data structures interact with the load/store pipeline, SIMD execution units, and cache hierarchy.

1.3.1 Alignment Requirements

- 8-byte alignment for 64-bit limbs
- 16-byte alignment for SSE
- 32-byte alignment for AVX
- 64-byte alignment for cache-line efficiency

1.3.2 Alignment with Assembler Directives

```
.intel_syntax noprefix
.section .bss
.align 32
msg_block:
    .space 128
```

1.3.3 Runtime Alignment Verification

```
.intel_syntax noprefix
.global check_align16

check_align16:
    test rdi, 0xF
    jz aligned
    xor eax, eax
```

```
    ret
aligned:
    mov eax, 1
    ret
```

1.3.4 Stack Alignment

System V ABI requires 16-byte alignment at call boundaries.

```
push rbp
mov rbp, rsp
and rsp, -16
```

1.3.5 Conclusion

Alignment is not optional in cryptographic systems. Correct placement of data ensures predictable performance, SIMD safety, and reliable multi-precision arithmetic. It is a foundational requirement of professional low-level cryptographic engineering.

Chapter 2

Primitive Operations

2.1 XOR & Bitwise Logic

Bitwise manipulation defines the lowest operational layer of modern cryptography. Hash functions, stream ciphers, block ciphers, MAC constructions, and multi-precision arithmetic rely on deterministic, reversible, constant-time transformations rooted in XOR and Boolean logic. At the machine level, these operations map directly to single-cycle ALU instructions on x86-64 Linux, making them fundamental to secure and high-performance cryptographic implementations.

Cryptographic designs assume precise bit-level semantics. XOR provides linear mixing, inversion supports complement-based constructions, and Boolean combinations form the basis of substitution and diffusion layers. These primitives drive the internal structure of algorithms such as ChaCha20, SHA-256, Keccak (SHA-3), Poly1305, and many bit-sliced block cipher designs.

The x86-64 ALU is optimized for this workload. XOR, AND, OR, and NOT execute with uniform latency, value-independent timing, and high throughput, satisfying the constant-time requirements essential for cryptographic code.

2.1.1 XOR as a Fundamental Cryptographic Operator

Exclusive-OR is the primary mixing operator in symmetric cryptography. Its algebraic properties—linearity, involution, and bit independence— make it suitable for keystream combination, whitening, parity propagation, and reversible state mixing.
XOR satisfies the following identities:

- x XOR k applies a reversible transformation
- (x XOR k) XOR k = x
- each bit position is processed independently

Minimal 64-bit XOR

```
.intel_syntax noprefix
.global xor64

xor64:
    mov rax, rdi
    xor rax, rsi
    ret
```

This instruction pattern appears in virtually every symmetric primitive.

2.1.2 XOR with Memory and Streaming State

Streaming cryptography frequently XORs words or bytes directly from memory, minimizing register pressure.

128-bit Block XOR (Two 64-bit Words)

```
.intel_syntax noprefix
.global xor_block128

xor_block128:
    mov rax, [rsi]
    xor rax, [rdx]
    mov [rdi], rax

    mov rcx, [rsi + 8]
    xor rcx, [rdx + 8]
    mov [rdi + 8], rcx
    ret
```

Used in ChaCha20 state updates, Poly1305 accumulation, and hash mixing loops.

2.1.3 AND, OR, and NOT: Boolean Control

Boolean logic complements XOR by enabling masking, selection, and state partitioning.

- Bit Masking

```
.intel_syntax noprefix
.global mask24

mask24:
    mov eax, edi
    and eax, 0x00FFFFFF
    ret
```

- Bit Injection

```
.intel_syntax noprefix
.global inject_flag
```

```
inject_flag:
    mov eax, edi
    or eax, 0x80000000
    ret
```

- Bitwise Inversion

```
.intel_syntax noprefix
.global invert64

invert64:
    mov rax, rdi
    not rax
    ret
```

2.1.4 Bit-Sliced Boolean Composition

Bit-sliced cryptography expresses substitution logic entirely through Boolean operations.

Boolean Mix: $(x \wedge y) \oplus (\neg x \wedge z)$

```
.intel_syntax noprefix
.global bool_mix

bool_mix:
    mov rax, rdi
    and rax, rsi

    mov rcx, rdi
    not rcx
    and rcx, rdx
```

```
    xor rax, rcx
    ret
```

Appears in Keccak χ, masked S-boxes, and constant-time DES/AES cores.

2.1.5 Boolean Functions in SHA-256

- Ch(x,y,z) = (x AND y) XOR (x AND z)
- Maj(x,y,z) = (x AND y) XOR (x AND z) XOR (y AND z)

```
.intel_syntax noprefix
.global sha256_ch

sha256_ch:
    mov rax, rdi
    and rax, rsi

    mov rcx, rdi
    not rcx
    and rcx, rdx

    xor rax, rcx
    ret
```

```
.intel_syntax noprefix
.global sha256_maj

sha256_maj:
    mov rax, rdi
    and rax, rsi

    mov rcx, rdi
```

```
    and rcx, rdx
    xor rax, rcx

    mov rcx, rsi
    and rcx, rdx
    xor rax, rcx
    ret
```

2.1.6 Constant-Time Behavior of Boolean Logic

XOR, AND, OR, and NOT execute with uniform latency and no data-dependent side effects on all modern x86-64 cores. This makes them ideal for:

- secret-key mixing
- masking and unmasking
- constant-time comparisons
- side-channel resistant arithmetic

2.1.7 Streaming XOR Loop

```
.intel_syntax noprefix
.global xor_stream

xor_stream:
    # rdi = dst, rsi = src, rdx = keystream, rcx = length
.loop:
    mov al, [rsi]
    xor al, [rdx]
    mov [rdi], al
```

```
inc rdi
inc rsi
inc rdx
dec rcx
jnz .loop
ret
```

This pattern defines stream ciphers and CTR-based constructions.

2.1.8 Conclusion

XOR and Boolean logic constitute the irreducible core of cryptographic computation. Their predictability, reversibility, and constant-time behavior make them indispensable for secure implementations on x86-64 Linux.

2.2 Shift vs Rotate

Shifts and rotations reposition bits within words but differ fundamentally in semantics. Shifts discard bits; rotations preserve all state. Cryptographic primitives exploit both behaviors deliberately.

2.2.1 Logical and Arithmetic Shifts

```
.intel_syntax noprefix
.global shl32

shl32:
    mov eax, edi
    shl eax, cl
    ret
```

```
.intel_syntax noprefix
.global shr32

shr32:
    mov eax, edi
    shr eax, cl
    ret
```

```
.intel_syntax noprefix
.global sar32

sar32:
    mov eax, edi
    sar eax, cl
    ret
```

Cryptography overwhelmingly prefers logical shifts.

2.2.2 Rotations

```
.intel_syntax noprefix
.global rotl32

rotl32:
    mov eax, edi
    rol eax, cl
    ret
```

```
.intel_syntax noprefix
.global rotr32

rotr32:
    mov eax, edi
    ror eax, cl
    ret
```

Rotations preserve entropy and enable reversible mixing.

2.2.3 Why Rotations Dominate ARX Designs

Rotations redistribute information without loss. They are fundamental to ChaCha20, Blake2, SipHash, and SHA-2 compression functions.

2.2.4 Portable Rotate Construction (Educational)

```
.intel_syntax noprefix
.global rotr64_manual

rotr64_manual:
    mov rax, rdi
    mov rcx, rax
```

```
    shr rax, cl
    shl rcx, 64
    shl rcx, cl

    or rax, rcx
    ret
```

Native ror must always be preferred in production code.

2.2.5 SHA-256 Shift-Based Functions

```
.intel_syntax noprefix
.global sha256_s0

sha256_s0:
    mov eax, edi

    mov ecx, eax
    shr ecx, 7

    mov edx, eax
    shr edx, 18

    mov r8d, eax
    shr r8d, 3

    xor ecx, edx
    xor ecx, r8d
    mov eax, ecx
    ret
```

2.2.6 Rotate-Based Diffusion

```
.intel_syntax noprefix
.global sha512_S1

sha512_S1:
    mov rax, rdi
    mov rcx, rdi
    mov rdx, rdi

    ror rax, 14
    ror rcx, 18
    ror rdx, 41

    xor rax, rcx
    xor rax, rdx
    ret
```

2.2.7 Conclusion

Shifts discard information and serve structural roles; rotations preserve state and enable cryptographic diffusion. Both execute efficiently on x86-64, but rotations dominate reversible cryptographic mixing.

2.3 Constant-Time Operations

Constant-time execution ensures that observable behavior does not depend on secret data. On x86-64 Linux, this requires eliminating secret-driven branches, memory access variation, and speculative leakage.

2.3.1 Constant-Time Selection

```
.intel_syntax noprefix
.global ct_select64

ct_select64:
    mov rax, rdi
    and rax, rdx

    mov rcx, rsi
    not rdx
    and rcx, rdx

    or rax, rcx
    ret
```

2.3.2 Constant-Time Comparison

```
.intel_syntax noprefix
.global ct_cmp

ct_cmp:
    xor eax, eax
.loop:
    mov bl, [rdi]
    xor bl, [rsi]
```

```
    or al, bl

    inc rdi
    inc rsi
    dec rdx
    jnz .loop

    sete al
    ret
```

2.3.3 Constant-Time Mask Generation

```
.intel_syntax noprefix
.global ct_less_u64

ct_less_u64:
    mov rax, rdi
    sub rax, rsi
    shr rax, 63
    neg rax
    ret
```

2.3.4 Constant-Time Swap

```
.intel_syntax noprefix
.global ct_swap64

ct_swap64:
    mov rax, [rdi]
    mov rcx, [rsi]

    mov r8, rax
    xor r8, rcx
```

```
    and r8, rdx

    xor rax, r8
    xor rcx, r8

    mov [rdi], rax
    mov [rsi], rcx
    ret
```

2.3.5 Constant-Time Zeroization

```
.intel_syntax noprefix
.global ct_zero

ct_zero:
.loop:
    mov BYTE PTR [rdi], 0
    inc rdi
    dec rdx
    jnz .loop
    ret
```

2.3.6 Conclusion

Constant-time primitives eliminate timing, branching, and cache-based leakage at the machine level. They are the non-negotiable foundation of secure cryptographic engineering on x86-64 Linux.

Chapter 3

SHA-1 Implementation

3.1 Message Scheduling

Message scheduling expands each 512-bit input block into eighty 32-bit words consumed by the SHA-1 compression function. Although SHA-1 is cryptographically obsolete for collision resistance, its internal structure remains valuable for understanding classical hash designs and their instruction-level behavior. On x86-64 Linux, message scheduling illustrates the interaction between word packing, XOR-based diffusion, and fixed rotations across extended state.

The scheduling phase transforms sixteen big-endian input words into sixty-four additional words using a strict linear recurrence. Any error in this stage results in incorrect digests regardless of the correctness of the compression rounds. Message scheduling therefore forms the foundation of a correct SHA-1 implementation.

3.1.1 Loading the Initial 16 Words

SHA-1 defines its input block as sixteen 32-bit big-endian words. Because x86-64 Linux is little-endian, these values must be explicitly byte- swapped during loading.

Load and Byte-Swap W[0..15]

```
.intel_syntax noprefix
.global sha1_pack16

sha1_pack16:
    # rdi = pointer to 64-byte input block
    # rsi = pointer to W[0..79]
    xor ecx, ecx

.load_loop:
    mov eax, [rdi + rcx*4]
    bswap eax
    mov [rsi + rcx*4], eax

    inc ecx
    cmp ecx, 16
    jl .load_loop
    ret
```

This establishes W[0] through W[15] as big-endian integers suitable for schedule expansion.

3.1.2 Expanding Words W[16] Through W[79]

SHA-1 defines its schedule recurrence as:

```
W[t] = ROTL1( W[t-3] XOR W[t-8] XOR W[t-14] XOR W[t-16] )
```

Each expanded word influences multiple compression rounds, amplifying diffusion across the state.

Schedule Expansion Loop

```
.intel_syntax noprefix
.global sha1_expand

sha1_expand:
    # rsi = pointer to W array
    mov ecx, 16

.expand_loop:
    mov eax, [rsi + (ecx-3)*4]
    xor eax, [rsi + (ecx-8)*4]
    xor eax, [rsi + (ecx-14)*4]
    xor eax, [rsi + (ecx-16)*4]

    rol eax, 1
    mov [rsi + ecx*4], eax

    inc ecx
    cmp ecx, 80
    jl .expand_loop
    ret
```

3.1.3 Combined Scheduler

For performance-sensitive code paths, loading and expansion may be combined.

```
.intel_syntax noprefix
.global sha1_schedule

sha1_schedule:
```

```
    # rdi = input block
    # rsi = W array
    xor ecx, ecx

.load:
    mov eax, [rdi + ecx*4]
    bswap eax
    mov [rsi + ecx*4], eax
    inc ecx
    cmp ecx, 16
    jl .load

.expand:
    mov eax, [rsi + (ecx-3)*4]
    xor eax, [rsi + (ecx-8)*4]
    xor eax, [rsi + (ecx-14)*4]
    xor eax, [rsi + (ecx-16)*4]
    rol eax, 1
    mov [rsi + ecx*4], eax
    inc ecx
    cmp ecx, 80
    jl .expand
    ret
```

3.1.4 Constant-Time Properties

The schedule uses fixed-index loads, XOR, and rotation only:

- no secret-dependent branches
- no data-dependent memory access
- uniform instruction flow

This makes scheduling inherently constant-time.

3.1.5 Conclusion

SHA-1 message scheduling is a linear, rotation-based expansion that demonstrates classical diffusion techniques. On x86-64 Linux, it maps cleanly to predictable, constant-time assembly using only registers and aligned memory access.

3.2 Compression Rounds

The compression function iteratively updates five 32-bit working registers across eighty rounds. Each round combines Boolean logic, rotation, modular addition, a scheduled word, and a round constant.

3.2.1 Working Registers

```
a, b, c, d, e
```

Round transformation:

```
temp = ROTL5(a) + F(b,c,d) + e + W[t] + K
e = d
d = c
c = ROTL30(b)
b = a
a = temp
```

3.2.2 Boolean Functions

- Rounds 0–19: Ch(b,c,d)
- Rounds 20–39: Parity
- Rounds 40–59: Maj
- Rounds 60–79: Parity

3.2.3 Round Constants

```
K0 = 0x5A827999
```

```
K1 = 0x6ED9EBA1
K2 = 0x8F1BBCDC
K3 = 0xCA62C1D6
```

3.2.4 Compression Loop Skeleton

```
.intel_syntax noprefix
.global sha1_compress

sha1_compress:
    # rdi = state (h0..h4)
    # rsi = W array

    mov eax, [rdi]
    mov ebx, [rdi+4]
    mov ecx, [rdi+8]
    mov edx, [rdi+12]
    mov r8d, [rdi+16]

    xor r9d, r9d

.round_loop:
    # Boolean function selection
    cmp r9d, 20
    jb .ch
    cmp r9d, 40
    jb .par
    cmp r9d, 60
    jb .maj
    jmp .par

.ch:
    mov r10d, ebx
```

```
    and r10d, ecx
    mov r11d, ebx
    not r11d
    and r11d, edx
    xor r10d, r11d
    jmp .f_done

.par:
    mov r10d, ebx
    xor r10d, ecx
    xor r10d, edx
    jmp .f_done

.maj:
    mov r10d, ebx
    and r10d, ecx
    mov r11d, ebx
    and r11d, edx
    xor r10d, r11d
    mov r11d, ecx
    and r11d, edx
    xor r10d, r11d

.f_done:
    # constant selection
    cmp r9d, 20
    jb .k0
    cmp r9d, 40
    jb .k1
    cmp r9d, 60
    jb .k2
    mov r11d, 0xCA62C1D6
    jmp .k_done
```

```
.k0: mov r11d, 0x5A827999 ; jmp .k_done
.k1: mov r11d, 0x6ED9EBA1 ; jmp .k_done
.k2: mov r11d, 0x8F1BBCDC
.k_done:

    mov r12d, eax
    rol r12d, 5
    add r12d, r10d
    add r12d, r8d
    add r12d, [rsi + r9d*4]
    add r12d, r11d

    mov r8d, edx
    mov edx, ecx
    rol ebx, 30
    mov ecx, ebx
    mov ebx, eax
    mov eax, r12d

    inc r9d
    cmp r9d, 80
    jl .round_loop

    add [rdi], eax
    add [rdi+4], ebx
    add [rdi+8], ecx
    add [rdi+12], edx
    add [rdi+16], r8d
    ret
```

3.2.5 Conclusion

SHA-1 compression demonstrates a disciplined ARX-style structure using rotations, Boolean logic, and modular addition. Its clarity makes it an excellent case study for low-level hash design.

3.3 Digest Finalization

Finalization applies deterministic padding, encodes the message length, processes the final block(s), and serializes the internal state into a 160-bit digest.

3.3.1 Padding

```
.intel_syntax noprefix
.global sha1_pad

sha1_pad:
    # rdi = buffer
    # rsi = used bytes
    # rdx = message length in bytes

    mov BYTE PTR [rdi + rsi], 0x80
    inc rsi

.pad:
    mov rcx, rsi
    and rcx, 63
    cmp rcx, 56
    je .len
    mov BYTE PTR [rdi + rsi], 0
    inc rsi
    jmp .pad

.len:
    mov rax, rdx
    shl rax, 3
    bswap rax
    mov [rdi + rsi], rax
```

```
    ret
```

3.3.2 Digest Serialization

```
.intel_syntax noprefix
.global sha1_write_digest

sha1_write_digest:
    mov eax, [rdi]      ; h0
    bswap eax
    mov [rsi], eax

    mov eax, [rdi+4]
    bswap eax
    mov [rsi+4], eax

    mov eax, [rdi+8]
    bswap eax
    mov [rsi+8], eax

    mov eax, [rdi+12]
    bswap eax
    mov [rsi+12], eax

    mov eax, [rdi+16]
    bswap eax
    mov [rsi+16], eax
    ret
```

3.3.3 Conclusion

Digest finalization completes the Merkle–Damgård chain through explicit padding, length binding, and big-endian serialization. On x86-64 Linux, every step requires

explicit byte control and constant-time execution. Understanding these mechanics remains essential for maintaining and analyzing legacy cryptographic systems.

Chapter 4

HMAC-SHA1

4.1 Inner and Outer Keys

HMAC transforms a cryptographic hash function into a message authentication primitive by incorporating a secret key in a domain-separated and collision-resistant manner. In HMAC-SHA1, the key is injected through two distinct layers: the inner key (K ipad) and the outer key (K opad). These two derived keys form the structural core of HMAC and define its security properties independently of the underlying hash function.

On x86-64 Linux, generating the inner and outer key blocks is a simple yet security-critical preprocessing step. It must correctly normalize keys of arbitrary length, produce exactly one 64-byte block for each layer, and execute in constant time with respect to secret data. Because SHA-1 operates on 512-bit blocks, both derived keys are always exactly 64 bytes long, regardless of the original key size.

Correct construction of these blocks ensures that the hash function starts from a domain-separated state, preventing structural weaknesses, cross-protocol collisions, and length-extension attacks.

4.1.1 Key Normalization and Block Preparation

HMAC defines two normalization rules for the input key:

1. If the key length is greater than 64 bytes, hash it using SHA-1, producing a 20-byte digest.
2. If the key length is less than 64 bytes, pad it with zeros until exactly 64 bytes are reached.

This process produces a fixed 64-byte block K_{64}, which becomes the basis for both inner and outer key derivation:

```
K_ipad = K64 XOR 0x36
K_opad = K64 XOR 0x5C
```

Both blocks are used as the first input block of the inner and outer SHA-1 computations respectively.

4.1.2 Generating K_ipad (Inner Key Block)

The ipad constant is the byte value 0x36 repeated 64 times.

Assembly: Generate Inner Key Block

```
.intel_syntax noprefix
.global hmacsha1_inner_key

hmacsha1_inner_key:
    # rdi = pointer to normalized 64-byte key (K64)
    # rsi = pointer to output buffer (K_ipad)
    mov rcx, 64
```

```
.inner_loop:
    mov al, [rdi]
    xor al, 0x36
    mov [rsi], al

    inc rdi
    inc rsi
    dec rcx
    jnz .inner_loop
    ret
```

This block is processed as the first SHA-1 input in the inner HMAC computation.

4.1.3 Generating K_opad (Outer Key Block)

The opad constant is the byte value 0x5C repeated 64 times.

Assembly: Generate Outer Key Block

```
.intel_syntax noprefix
.global hmacsha1_outer_key

hmacsha1_outer_key:
    # rdi = pointer to normalized 64-byte key (K64)
    # rsi = pointer to output buffer (K_opad)
    mov rcx, 64

.outer_loop:
    mov al, [rdi]
    xor al, 0x5C
    mov [rsi], al

    inc rdi
    inc rsi
```

```
    dec rcx
    jnz .outer_loop
    ret
```

This block seeds the outer SHA-1 computation after the inner digest is produced.

4.1.4 Constant-Time Properties

Inner and outer key generation is inherently constant-time:

- all 64 bytes are processed unconditionally
- no branches depend on key content
- memory access patterns are fixed and linear
- XOR against immediates has uniform latency

This prevents leakage of key material or key length through timing or cache-based side channels.

4.1.5 Integrated Inner and Outer Key Generation

For improved cache locality, both derived keys may be produced in a single pass:

```
.intel_syntax noprefix
.global hmacsha1_prepare_keys

hmacsha1_prepare_keys:
    # rdi = K64
    # rsi = K_ipad
    # rdx = K_opad
    mov rcx, 64
```

```
.loop:
    mov al, [rdi]

    mov bl, al
    xor bl, 0x36
    mov [rsi], bl

    xor al, 0x5C
    mov [rdx], al

    inc rdi
    inc rsi
    inc rdx
    dec rcx
    jnz .loop
    ret
```

4.1.6 Conclusion

The inner and outer key derivation stage forms the cryptographic foundation of HMAC-SHA1. By normalizing the key into a fixed 64-byte block and applying two domain-separated XOR masks, HMAC establishes independent hash domains for its inner and outer layers. On x86-64 Linux, these operations map cleanly to constant-time byte-wise XOR loops, requiring no library support and providing full control over memory and timing behavior.

4.2 Block Padding

HMAC-SHA1 uses the same padding rules as raw SHA-1, but applies them to different input streams:

```
(K_ipad || message)
(K_opad || inner_digest)
```

Padding must be applied only after the entire concatenated input stream is processed. It is deterministic, content-independent, and essential for both correctness and security.

4.2.1 Padding Rules

Given an input stream S, SHA-1 padding consists of:

1. Append byte 0x80
2. Append zero bytes until length mod 64 = 56
3. Append 64-bit big-endian length of S (in bits)

4.2.2 Constant-Time Padding Routine

```
.intel_syntax noprefix
.global hmacsha1_pad_block

hmacsha1_pad_block:
    # rdi = buffer
    # rsi = current length (bytes)
    # rdx = total length before padding (bytes)

    mov BYTE PTR [rdi + rsi], 0x80
```

```
    inc rsi

.pad:
    mov rcx, rsi
    and rcx, 63
    cmp rcx, 56
    je .len

    mov BYTE PTR [rdi + rsi], 0
    inc rsi
    jmp .pad

.len:
    mov rax, rdx
    shl rax, 3
    bswap rax
    mov [rdi + rsi], rax
    ret
```

4.2.3 Conclusion

Correct padding binds the message length to the authentication code, prevents extension attacks, and ensures compatibility with the SHA-1 compression engine. Padding logic must be unconditional, constant-time, and independent of message content.

4.3 Test Vector Verification

Test vector verification is the definitive correctness check for an HMAC-SHA1 implementation. Because HMAC combines key normalization, domain separation, SHA-1 padding, message scheduling, compression, and digest serialization, any error at the bit level will propagate into an incorrect authentication tag.

4.3.1 Constant-Time Digest Comparison

```
.intel_syntax noprefix
.global hmacsha1_verify_digest

hmacsha1_verify_digest:
    # rdi = computed digest (20 bytes)
    # rsi = expected digest (20 bytes)

    xor eax, eax
    mov rcx, 20

.loop:
    mov bl, [rdi]
    xor bl, [rsi]
    or al, bl

    inc rdi
    inc rsi
    dec rcx
    jnz .loop

    sete al     # al = 1 if equal, 0 otherwise
    ret
```

4.3.2 Conclusion

Test vector verification confirms the correctness of the entire HMAC construction. On x86-64 Linux, this requires constant-time comparison, precise big-endian handling, deterministic padding, and manual buffer management. Verification is not optional—it is a foundational requirement for trustworthy cryptographic software.

Chapter 5

ChaCha20

5.1 State Layout

ChaCha20 is an ARX (Add–Rotate–XOR) stream cipher defined around a 512-bit internal state composed of sixteen 32-bit little-endian words. The exact layout of this state determines every aspect of ChaCha20's security and correctness. Before any quarter-round or block-round operation can occur, the cipher must construct an exact 4×4 state matrix from constants, key material, a counter, and a nonce. Any deviation in endianness, alignment, or word ordering produces an incorrect keystream.
On x86-64 Linux, ChaCha20's little-endian design aligns naturally with the architecture. All words are loaded and stored without byte-swapping, making ChaCha20 particularly efficient in low-level assembly implementations.

5.1.1 The 4×4 ChaCha20 State Matrix

The ChaCha20 state is arranged as follows:

```
[ c0   c1   c2   c3 ]   constants
```

```
[  k0   k1   k2   k3 ]   key (first half)
[  k4   k5   k6   k7 ]   key (second half)
[ ctr  n0   n1   n2 ]   counter + nonce
```

Where:

- Constants: ASCII string ”expand 32-byte k” encoded as four 32-bit little-endian words.
- Key: 256-bit key split into eight 32-bit little-endian words.
- Counter: 32-bit block counter.
- Nonce: 96-bit nonce split into three 32-bit words.

This layout is fixed and must be reproduced exactly.

5.1.2 Encoding the Constants

The ChaCha20 constants are:

```
”expa” ”nd 3” ”2-by” ”te k”
```

Interpreted as little-endian 32-bit integers:

```
0x61707865
0x3320646e
0x79622d32
0x6b206574
```

On x86-64 Linux, these values load directly without conversion.

5.1.3 Memory Layout of the State Buffer

A typical state buffer layout:

```
.section .bss
.align 64
chacha_state:
    .space 64
```

Alignment ensures cache efficiency, predictable access patterns, and compatibility with vectorized implementations.

5.1.4 State Initialization in Assembly

The following routine initializes the ChaCha20 state:

- rdi → key pointer (32 bytes)
- rsi → nonce pointer (12 bytes)
- edx → block counter
- rcx → state buffer (64 bytes)

```
.intel_syntax noprefix
.global chacha20_init_state

chacha20_init_state:
    # Constants
    mov DWORD PTR [rcx],     0x61707865
    mov DWORD PTR [rcx+4],   0x3320646e
    mov DWORD PTR [rcx+8],   0x79622d32
    mov DWORD PTR [rcx+12],  0x6b206574
```

```
# Key words
mov eax, [rdi]
mov [rcx+16], eax
mov eax, [rdi+4]
mov [rcx+20], eax
mov eax, [rdi+8]
mov [rcx+24], eax
mov eax, [rdi+12]
mov [rcx+28], eax

mov eax, [rdi+16]
mov [rcx+32], eax
mov eax, [rdi+20]
mov [rcx+36], eax
mov eax, [rdi+24]
mov [rcx+40], eax
mov eax, [rdi+28]
mov [rcx+44], eax

# Counter
mov eax, edx
mov [rcx+48], eax

# Nonce
mov eax, [rsi]
mov [rcx+52], eax
mov eax, [rsi+4]
mov [rcx+56], eax
mov eax, [rsi+8]
mov [rcx+60], eax

ret
```

5.1.5 Endianness

ChaCha20 is strictly little-endian:

- no bswap instructions
- native loads and stores are correct
- output keystream is little-endian

This is ideal for x86-64 Linux.

5.1.6 Conclusion

Correct state layout is the bedrock of ChaCha20. Any deviation in constant placement, key ordering, counter position, or nonce encoding results in an invalid keystream. On x86-64 Linux, ChaCha20's design enables direct native word operations and highly efficient low-level implementations.

5.2 Quarter Round Function

The quarter round is the fundamental nonlinear transformation in ChaCha20. It operates on four 32-bit words using only addition, XOR, and rotation. All diffusion and security properties of ChaCha20 arise from repeated application of this operation.

5.2.1 Quarter Round Definition

```
a += b;   d ^= a;   d = ROTL(d,16)
c += d;   b ^= c;   b = ROTL(b,12)
a += b;   d ^= a;   d = ROTL(d, 8)
c += d;   b ^= c;   b = ROTL(b, 7)
```

5.2.2 Register Mapping

```
a → r8d
b → r9d
c → r10d
d → r11d
```

5.2.3 Quarter Round in Assembly

```
.intel_syntax noprefix
.global chacha20_quarter_round

chacha20_quarter_round:
    # r8d = a, r9d = b, r10d = c, r11d = d

    add r8d, r9d
    xor r11d, r8d
    rol r11d, 16
```

```
add r10d, r11d
xor r9d, r10d
rol r9d, 12

add r8d, r9d
xor r11d, r8d
rol r11d, 8

add r10d, r11d
xor r9d, r10d
rol r9d, 7

ret
```

5.2.4 Why These Operations

- Addition introduces carry-based nonlinearity
- XOR mixes independent bit domains
- Rotation redistributes entropy without loss
- All operations are reversible and constant-time

5.2.5 Conclusion

The quarter round is the atomic ARX unit of ChaCha20. Its efficiency, constant-time behavior, and strong diffusion properties make ChaCha20 exceptionally well-suited for modern CPUs.

5.3 Block Output Generation

After 20 rounds, ChaCha20 produces a working state. The final step adds this state to the original input state (feed-forward) and serializes the result into a 64-byte keystream block.

5.3.1 Feed-Forward Step

```
output[i] = working[i] + original[i]   (mod 2^32)
```

5.3.2 Feed-Forward Assembly

```
.intel_syntax noprefix
.global chacha20_store_block

chacha20_store_block:
    # rcx = original state
    # rdx = output buffer

    add r8d,  [rcx]
    mov [rdx], r8d

    add r9d,  [rcx+4]
    mov [rdx+4], r9d

    add r10d, [rcx+8]
    mov [rdx+8], r10d

    add r11d, [rcx+12]
    mov [rdx+12], r11d

    ret
```

(Additional words follow the same pattern.)

5.3.3 Keystream Semantics

ChaCha20 generates a raw keystream:

```
ciphertext = plaintext XOR keystream
```

The cipher core must never process plaintext directly.

5.3.4 Conclusion

Block output generation completes the ChaCha20 block function. By combining feed-forward addition with little-endian serialization, ChaCha20 produces a high-quality keystream suitable for secure stream encryption. Correct implementation of this step is mandatory for both security and correctness.

Chapter 6

Final Project

6.1 A Complete Cryptography Assembly Library

A fully self-contained cryptographic assembly library for x86-64 Linux represents the culmination of all foundational primitives developed throughout this book. Such a library is not a simple aggregation of independent routines; it is a tightly controlled execution environment where cryptographic correctness, constant-time behavior, register discipline, and memory alignment are enforced without exception.
This final project formalizes the implementation of three fundamental cryptographic families:

1. Hash Functions (SHA-1)
2. Keyed Hashing (HMAC-SHA1)
3. Stream Ciphers (ChaCha20)

In addition, the library provides essential constant-time utilities, secure memory zeroization routines, and strictly aligned internal state buffers. All components are

implemented entirely in Intel-syntax x86-64 assembly, with no dependency on libc or compiler-generated code, making the library suitable for secure system utilities, embedded Linux environments, kernel-adjacent tooling, and controlled educational analysis.

6.1.1 Architectural Philosophy

- Principle 1 — No Undefined Behavior

  Every routine is deterministic, self-contained, and ABI-conformant. There is no heap usage, no dynamic allocation, no reliance on undefined CPU behavior, and no implicit compiler assistance.

- Principle 2 — Constant-Time Execution

  Cryptographic correctness without timing safety is insufficient. Therefore:

  - no branches on secret data
  - no secret-indexed table lookups
  - no data-dependent memory access
  - fully predictable control flow

  This discipline protects against timing, cache, and speculative side-channel leakage on modern x86-64 processors.

- Principle 3 — Register-Centric Execution

  Critical loops execute entirely in registers:

  - r8d–r15d for working state
  - eax–edx for auxiliary arithmetic

  - minimal stack interaction

  Register-only computation minimizes latency and eliminates memory-based timing variance.

- Principle 4 — Zero Dependency on the C Runtime

  There is no reliance on:

  - malloc / free
  - memcmp / memset
  - string functions
  - libc cryptography

  All logic is explicitly implemented in assembly.

6.1.2 Directory-Level Organization

A clear, modular directory structure enables direct inspection and maintenance:

```
/crypto/
   /sha1/
      sha1_init_state.s
      sha1_schedule.s
      sha1_compress.s
      sha1_finalize.s
      sha1_write_digest.s

   /hmac/
      hmac_normalize_key.s
      hmac_inner_outer.s
      hmac_pad.s
```

```
    hmac_inner_hash.s
    hmac_outer_hash.s
    hmac_compare.s

/chacha20/
    chacha20_init_state.s
    chacha20_quarter_round.s
    chacha20_double_round.s
    chacha20_block_core.s
    chacha20_store_block.s
    chacha20_encrypt.s
    chacha20_increment_counter.s

/utils/
    consttime_compare.s
    consttime_select.s
    consttime_mask.s
    secure_zero.s
```

This mirrors industry-grade cryptographic separation of concerns.

6.1.3 ABI, Calling Convention, and Register Contract

All routines conform strictly to the System V AMD64 ABI, with additional cryptographic constraints.

1. Register Scrubbing Discipline

   Before returning:

   - temporary registers are cleared
   - sensitive values are explicitly zeroed

2. Stack Alignment

   Every external entry point enforces:

   ```
   push rbp
   mov rbp, rsp
   and rsp, -16
   ```

   This guarantees correct behavior on AVX-capable systems.

3. No Red-Zone Usage

   Although the ABI defines a 128-byte red zone, cryptographic code avoids it entirely because interrupts, kernels, or hypervisors may overwrite it.

6.1.4 Digest Layer (SHA-1)

The SHA-1 implementation includes:

- state initialization
- message scheduling
- 80-round compression
- padding and finalization
- big-endian digest serialization

Scratch Buffer Layout

```
.section .bss
.align 64
sha1_state:       .space 20
sha1_schedule:    .space 320
sha1_block:       .space 64
```

Compression Pipeline

```
Load → Schedule → Expand → 80 rounds → Feed-forward → Store
```

All operations are register-based and constant-time.

6.1.5 HMAC Layer (HMAC-SHA1)

HMAC-SHA1 is implemented as a full two-pass keyed hash construction.

- Key normalization (length-based only)
- Constant-time inner and outer pad generation
- Two independent SHA-1 passes
- Constant-time digest comparison

```
Inner: SHA1(K_ipad || message)
Outer: SHA1(K_opad || inner_digest)
```

6.1.6 ChaCha20 Stream Cipher Layer

ChaCha20 is implemented as a full ARX engine:

- state initialization
- quarter rounds
- double rounds
- 20-round block core
- keystream XOR encryption

Aligned State Buffers

```
.section .bss
.align 64
chacha_state:      .space 64
chacha_working:    .space 64
keystream_block:   .space 64
```

Block Flow

```
State → Working Copy → 20 rounds → Feed-forward → Keystream
```

6.1.7 Constant-Time Utility Layer

- constant-time select
- constant-time mask generation
- explicit register zeroization

```
xor r8d, r8d
xor r9d, r9d
```

6.1.8 Secure Memory Handling

Sensitive buffers are cleared immediately after use.

```
.global secure_zero
secure_zero:
    # rdi = pointer
    # rsi = length
.loop:
    mov BYTE PTR [rdi], 0
    inc rdi
```

```
    dec rsi
    jnz .loop
    lfence
    ret
```

6.1.9 Cross-Primitive Guarantees

- uniform constant-time behavior
- no shared mutable state
- strict endianness separation
- ABI-safe symbol exports

6.1.10 Example: Encryption + Authentication Pipeline

```
# Compute MAC
call hmac_sha1

# Encrypt data
call chacha20_encrypt
```

This design runs safely in kernels, enclaves, bootloaders, and embedded Linux systems.

6.1.11 Unified Memory Layout

```
.section .bss
.align 64

sha1_state:         .space 20
sha1_schedule:      .space 320
sha1_block:         .space 64
```

```
K64_buffer:          .space 64
ipad_buffer:         .space 64
opad_buffer:         .space 64
inner_digest:        .space 20
final_digest:        .space 20

chacha_state:        .space 64
chacha_temp:         .space 64
keystream_block:     .space 64

ct_mask_buffer:      .space 32
zero_buffer:         .space 64
```

6.1.12 Conclusion

This final project demonstrates how to construct a complete, high-assurance cryptographic library entirely in x86-64 assembly. By enforcing constant-time execution, explicit memory control, strict ABI discipline, and register-only arithmetic, the integration of SHA-1, HMAC-SHA1, and ChaCha20 becomes not merely functional but auditable and educational. This library represents the synthesis of all concepts presented in this book and provides a robust foundation for deeper cryptographic engineering and analysis in future volumes.

Conclusion

Low-level cryptography on Linux, when approached from the perspective of x86-64 assembly and constant-time software engineering, differs fundamentally from cryptography implemented in high-level languages. This book has demonstrated that secure cryptographic operations must be understood not merely as abstract algorithms, but as concrete state transformations executed through registers, flags, carefully controlled memory writes, and deterministic instruction sequences. At this level, cryptographic assurance is inseparable from microarchitectural awareness, instruction semantics, and disciplined control flow.

Throughout this volume, the construction of SHA-1, HMAC-SHA1, and ChaCha20 has illustrated the structural principles shared by modern symmetric cryptographic primitives:

1. ARX Transformations
   Addition, rotation, and XOR form an algebraic foundation that scales naturally to contemporary CPUs. These operations execute in constant time on x86-64, require no lookup tables, and provide strong diffusion with a minimal instruction footprint.

2. Bit-Level State Manipulation
   Cryptographic state—whether a 512-bit hash buffer or a 4×4 ChaCha20 matrix—must be constructed explicitly. Layout, alignment, and endianness of multi-word

structures directly determine correctness, interoperability, and performance.

3. Register-Only Critical Paths
   Cryptographic round functions execute entirely within registers, ensuring predictable timing, minimal memory interaction, and resistance to microarchitectural leakage caused by cache behavior or memory stalls.

4. Feed-Forward Constructions
   Both hash functions and stream ciphers rely on combining transformed state with original input via modular addition or XOR. This feed-forward step prevents reversible permutations and enforces full diffusion across rounds.

5. Domain Separation Through Structured Initialization
   HMAC's inner and outer key constructions, and ChaCha20's constant–key–counter–nonce state matrix, demonstrate how cryptographic domains are separated without external checks. All separation occurs through deterministic transformations implemented directly in assembly.

6. Constant-Time Execution Discipline
   Cryptographic correctness is incomplete without timing safety. Every implementation in this book strictly avoids secret-dependent branching, data-dependent memory access, and unpredictable control paths, making constant-time execution a mandatory property rather than an optional optimization.

7. Explicit Padding, Serialization, and Buffer Control
   Hash padding, big-endian digest serialization, little-endian word loading, and block-aligned memory regions are all implemented manually. These operations are integral to cryptographic safety, not auxiliary details.

8. Self-Contained Assembly Library Architecture

> By constructing a complete cryptographic library without libc, this book has shown that robust cryptography can—and in some environments must—be implemented at the lowest software layers to achieve predictability, auditability, and full control.

Low-level cryptographic engineering is not merely the act of re-implementing algorithms. It is the discipline of eliminating uncertainty—whether introduced by compilers, runtime libraries, speculative execution, or undocumented system behavior. Writing cryptography in assembly forces the programmer to account for every byte stored, every carry propagated, every rotation applied, and every register overwritten. The result is software whose behavior can be reasoned about with precision and validated through controlled test vectors without hidden assumptions.

More importantly, an engineer who implements cryptographic primitives at this level gains a complete understanding of the relationship between algorithmic structure and machine execution. The constraints of x86-64 hardware become guiding principles rather than obstacles: wide registers, predictable ALU behavior, and uniform instruction timing naturally support the construction of secure, high-performance cryptographic systems.

This volume concludes with the recognition that low-level cryptography is ultimately an engineering philosophy. It is the belief that security derives from transparency and control, and that the most reliable way to trust a cryptographic primitive is to understand its behavior at the instruction level. Within the broader context of the Low-Level Programming on Linux (x86-64) series, this book forms a bridge between abstract cryptographic design and practical, high-assurance assembly-level construction. It provides a foundation upon which more advanced primitives—modern hash functions, authenticated encryption schemes, and multi-word ARX ciphers—can be built with confidence, discipline, and clarity.

References

The material presented in this booklet is grounded in established cryptographic specifications, post-2020 analysis of ARX-based constructions, and modern understanding of constant-time software engineering on x86-64 Linux systems. Because this work is a low-level, assembly-centric treatment of cryptography, the references listed here do not take the form of conventional bibliographic citations. Instead, they represent domains of knowledge, formal algorithmic models, and execution guarantees upon which the presented implementations rely.

These reference domains ensure that the cryptographic primitives described throughout this booklet conform to modern standards, predictable CPU behavior, and secure implementation practices expected of system-level cryptography written directly in assembly language.

Modern CPU Architectural Behavior

This booklet relies on contemporary knowledge of x86-64 microarchitectural behavior as understood and validated after 2020. These include, but are not limited to:

- Constant-time execution guarantees for integer addition, XOR, and rotation instructions on modern AMD and Intel processors.

- Predictable instruction latency and throughput for ARX-based primitives across recent CPU generations.
- Microarchitectural side-channel mitigation principles, including the documented behavior of the lfence instruction as a serialization and speculative-execution barrier.
- Cache-line alignment and low-level memory behavior, which are essential for secure execution of hash schedules, ChaCha20 rounds, and streaming XOR operations.

These architectural properties form the execution foundation for all assembly routines presented in this booklet.

Cryptographic Algorithm Specifications (Formal Structures)

The correctness of the implementations in this booklet derives from the formal algorithmic definitions of three cryptographic primitives. Although implemented entirely in assembly, each construction follows its established mathematical structure:

- SHA-1 (Cryptographic Hash Function)
  Defined by its message schedule expansion, 80-round compression function, big-endian serialization rules, and feed-forward state addition. The reference model used here reflects the modern post-2020 understanding of SHA-1's internal ARX-style structure, independent of deprecated security assumptions.
- HMAC-SHA1 (Keyed Hash Construction)
  Defined by its domain-separated structure using (K ipad) || message and (K opad) || inner_digest, along with explicit

bit-length binding, deterministic block padding, and strict constant-time key handling. The structure presented follows the contemporary deterministic interpretation of HMAC used in modern security systems.

- ChaCha20 (Stream Cipher)
  Defined by its 4×4 state matrix, quarter-round permutation, column/diagonal double-round structure, and 20-round block function. Modern cryptographic practice emphasizes deterministic ARX transformations, native little-endian word operations, and counter-based keystream generation.

These formal definitions provide the mathematical foundation on which the assembly-level implementations are constructed.

Cryptographic Engineering Principles

The design and implementation choices throughout this booklet are informed by established principles of low-level cryptographic engineering:

- Constant-time programming discipline, including the avoidance of secret-dependent branching, data-indexed memory access, and variable execution paths.
- Bit-exact state construction, ensuring faithful reproduction of algorithmic endianness, padding structure, and multi-word state layout.
- Secure key handling practices, including normalization, explicit zeroization, and manual buffer cleansing implemented directly in assembly.
- Feed-forward constructions, used in both hash functions and stream ciphers to prevent reversible or weak permutation behavior.

- ARX design philosophy, which leverages addition, rotation, and XOR to produce efficient, pipeline-friendly cryptographic primitives suitable for modern CPUs.

These principles collectively govern every instruction sequence shown in the assembly listings.

Low-Level Linux Execution Model

All code and concepts in this booklet assume a modern Linux execution environment with the following characteristics:

- System V AMD64 ABI compliance, as interpreted and applied in post-2020 toolchains and runtime environments.
- Flat user-space memory model, enabling direct register-based cryptographic computation without segmentation overhead.
- Predictable alignment and cache behavior, critical when allocating 64-byte state buffers for hash and stream cipher operations.
- Complete absence of external runtime libraries within cryptographic routines, requiring explicit management of stack frames, alignment, and pointer arithmetic.

This execution model defines the environment in which all assembly routines in this booklet operate.

Formal Integer and Bitwise Computation Rules

Because cryptographic transformations are executed directly through registers and ALU operations, this booklet relies on the precise and well-defined semantics of:

- 32-bit modular addition (add r32, r32)
- Logical XOR and AND operations
- Left-rotation semantics (rol r32, imm)
- Consistent endianness behavior across registers and memory
- Deterministic overflow and carry propagation in two's complement arithmetic

These formal rules define the mathematical correctness of all ARX-based constructions presented in the text.

Assembly Code Correctness Principles

The cryptographic routines in this booklet adhere to established assembly-level engineering practices:

- Register-only critical paths, ensuring constant-time execution of mixing functions.
- Manually controlled stack frames, including strict 16-byte alignment guarantees.
- Fully static buffer allocation using .bss sections with explicit alignment.
- Instruction-level determinism, allowing reasoning about behavior across CPU generations.
- Complete avoidance of undefined behavior, both architectural and algorithmic.

These principles serve as the practical reference model for implementing cryptographic primitives in pure assembly.

Conclusion

The reference framework for this booklet lies at the intersection of modern CPU behavior (post-2020), formal cryptographic algorithm definitions, constant-time software engineering, and deterministic assembly-level programming. Together, these domains form the conceptual and technical foundation upon which all algorithms, techniques, and code samples in this work are built. By grounding every routine in verified mathematical structures and precise instruction semantics, this booklet maintains the rigor required for trustworthy low-level cryptographic engineering on Linux x86-64 systems.