Faculty of Engineering – Ain Shams University

Computer and Systems Engineering Department

zGate Gateway: A Zero Trust Database Access Proxy

Graduation Project Thesis

Team Members:

Michael George

Karen ...

Rodina ...

Supervisor:

Dr. ...

Academic Year 2025–2026

Acknowledgments

We would like to thank...

Abstract

This project introduces a Zero Trust–based database access gateway (SecureDB Gateway)...

Contents

List of Figures

List of Tables

1. Team Information

1.1 Team Members

1.2 Roles & Responsibilities

2. Introduction & Problem Definition

2.1 Introduction

2.2 Problem Statement

2.3 Gap in Existing Solutions

2.4 Why Zero Trust for Databases is Different

3. Project Definition

3.1 Project Definition & Scope

3.2 Objectives

3.3 Expected Academic Contribution

4. Requirements Engineering

4.1 Functional Requirements

4.2 Non-Functional Requirements

4.3 Actors & Use Cases

4.4 Use Case Diagrams

4.5 User Stories

5. Proposed Solution

5.1 Overview of the Proposed Solution

5.2 Solution Architecture Layers

5.2.1 Authentication Layer

5.2.2 Proxy Layer

5.2.3 Policy Engine Layer

5.2.4 Data Protection Layer

5.2.5 Observability Layer

5.3 Term 1 vs Term 2 Features

5.4 Feature List

6. Alignment with International Standards

6.1 PCI DSS

6.2 HIPAA

6.3 GDPR

6.4 ISO 27001

7. Competitor & Market Analysis

7.1 Competitor Analysis

7.1.1 Comparison Table

7.1.2 What Competitors Lack

7.2 Market Research

7.2.1 Market Overview

7.2.2 Zero Trust Demand

7.2.3 Market Challenges & Needs

7.2.4 Regulatory Drivers

7.2.5 Trends & Opportunities

7.2.6 Landscape Summary

8. Scientific Research & Literature Review

8.1 Paper 1

8.2 Paper 2

8.3 Paper 3

8.4 Paper 4

8.5 Paper 5

8.6 Paper 6

8.7 Paper 7

9. Technical Background

This chapter provides an in-depth technical overview of all technologies, protocols, and architectural concepts forming the foundation of the zGate Zero Trust Database Access Proxy. The discussion is intentionally extensive to support academic rigor and enable future researchers or developers to extend the project.

9.1 Systems Programming in Go

The zGate gateway is implemented entirely in Go (Golang) version 1.25.4. Go is selected due to its strong concurrency model, built-in memory safety guarantees, and first-class support for networked systems.

9.1.1 Introduction to Go Programming Language

Go, also known as Golang, is a statically typed, compiled programming language designed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. First released publicly in 2009, Go was created to address shortcomings in other languages used for systems programming, particularly in the context of multicore processors, networked systems, and large codebases.

Design Philosophy Go emphasizes simplicity, readability, and pragmatism. Key design principles include:

- **Simplicity**: Minimalist syntax with only 25 keywords
- **Explicit over implicit**: No hidden control flow or magic behaviors
- **Composition over inheritance**: Interfaces and struct embedding instead of class hierarchies
- **Fast compilation**: Designed for rapid build times even in large projects
- **Built-in concurrency**: First-class language support for concurrent programming

Memory Safety Go provides automatic memory management through garbage collection, eliminating entire classes of vulnerabilities:

- **No manual memory management**: Prevents use-after-free and double-free errors
- **Bounds checking**: Array and slice accesses are automatically validated
- **No pointer arithmetic**: Prevents buffer overflows and memory corruption
- **Type safety**: Strong static typing prevents type confusion attacks

Standard Library Go's extensive standard library includes production-ready packages for:

- Network programming (`net`, `net/http`)
- Cryptography (`crypto/*`)
- Encoding/decoding (`encoding/json`, `encoding/xml`)
- Testing and benchmarking (`testing`)
- Concurrent programming (`sync`, `context`)

9.1.2 Go Runtime Model

Go employs a sophisticated user-space thread management architecture based on the G–M–P scheduling model, which enables efficient concurrency without the overhead of traditional OS threads.

The G–M–P Model Explained

- **G (Goroutine)**: A goroutine is a lightweight cooperative thread with a dynamically-sized stack that starts at 2KB and can grow to several megabytes. Unlike OS threads, goroutines are managed entirely in user space by the Go runtime. Goroutines use cooperative scheduling, meaning they yield control at specific points (channel operations, system calls, function calls) rather than being preemptively interrupted.
- **M (Machine)**: An M represents an OS thread managed by the operating system kernel. The Go runtime creates a pool of Ms (typically matching the number of CPU cores) that execute goroutines. When a goroutine performs a blocking system call, the M is detached and a new M may be created to continue executing other goroutines.

- **P (Processor)**: A P is a scheduling context that maintains a local run queue of goroutines. The number of Ps is typically set to the number of available CPU cores (controlled by GOMAXPROCS). Each M must be associated with a P to execute goroutines. When a goroutine blocks, the P can be handed off to another M, allowing other goroutines to continue execution.

Scheduling Mechanism The scheduler implements work-stealing to balance load:

1. Each P maintains a local queue of runnable goroutines
2. When a P's queue is empty, it attempts to steal work from other Ps
3. A global run queue handles goroutines that don't fit in local queues
4. Network poller integration enables efficient I/O multiplexing

Why This Matters for zGate This model enables thousands of goroutines to execute concurrently with negligible overhead. zGate relies on this feature because each client session, backend connection, interceptor callback, and logging pipeline runs as its own goroutine. A typical deployment might handle 10,000+ concurrent database connections, each requiring multiple goroutines, which would be impossible with traditional thread-per-connection models.

9.1.3 Concurrency Primitives

Go provides several built-in primitives for concurrent programming that form the foundation of zGate's concurrent architecture.

Goroutines in Detail Goroutines are created using the `go` keyword followed by a function call. They provide several advantages:

- **Low memory overhead**: Each goroutine starts with only 2KB stack space vs 1-2MB for OS threads
- **Fast creation**: Creating a goroutine takes microseconds vs milliseconds for threads
- **Efficient scheduling**: Context switching between goroutines is faster than kernel thread switches
- **Scalability**: Applications can easily spawn millions of goroutines

In zGate, goroutines are used for:

- **Frontend packet reader**: Continuously reads MySQL packets from client connections
- **Backend packet writer**: Forwards packets to the database server
- **Audit logger**: Asynchronously writes audit entries without blocking query processing
- **TLS handshake worker**: Handles cryptographic handshakes in parallel
- **Interceptor orchestrators**: Executes policy enforcement logic concurrently

Channels in Depth Channels are Go's primary mechanism for communication between goroutines, implementing the CSP (Communicating Sequential Processes) model. Channels are typed, thread-safe queues that can be buffered or unbuffered.

Channel Types:

- **Unbuffered channels**: Synchronous - sender blocks until receiver is ready
- **Buffered channels**: Asynchronous up to buffer size - sender blocks only when buffer is full
- **Directional channels**: Can be send-only (`chan<-`) or receive-only (`<-chan`)

Channel Operations:

- **Send**: `ch <- value`
- **Receive**: `value := <-ch`
- **Close**: `close(ch)`
- **Select**: Multiplexing over multiple channel operations

In zGate, channels provide synchronization for:

- **Session-level error propagation**: When a critical error occurs in any goroutine handling a session
- **Asynchronous event forwarding**: Audit events, metrics, and alerts
- **Administrative operation coordination**: Graceful shutdown, configuration reloads
- **Work distribution**: Distributing query processing tasks across worker pools

Mutexes and Atomic Operations Mutex (Mutual Exclusion): A mutex is a synchronization primitive that protects shared data from concurrent access:

- **sync.Mutex**: Provides exclusive locking - only one goroutine can hold the lock
- **sync.RWMutex**: Reader-writer mutex - allows multiple readers OR one writer
- **Lock/Unlock pattern**: Must be paired, typically using `defer` to ensure unlock

Atomic Operations: The `sync/atomic` package provides lock-free operations for simple data types:

- **Atomic integers**: Add, Load, Store, Swap, CompareAndSwap operations
- **Performance**: Much faster than mutex-based protection for simple counters
- **Memory ordering**: Provides happens-before guarantees

Critical shared resources in zGate use:

- **sync.Mutex / sync.RWMutex**: For metadata caches (user sessions, prepared statements)
- **sync.Once**: For one-time initialization of secrets, certificates, and database connections
- **sync/atomic**: For high-throughput metrics (query counts, error rates, latency tracking)

9.1.4 Low-Level TCP Socket Programming

Unlike typical database clients that rely on high-level drivers, zGate communicates directly using the MySQL wire protocol over raw TCP sockets. This low-level approach provides complete control over the communication pipeline.

TCP/IP Socket Fundamentals TCP (Transmission Control Protocol):

- **Connection-oriented**: Requires handshake (SYN, SYN-ACK, ACK) before data transfer
- **Reliable**: Guarantees in-order delivery with automatic retransmission
- **Flow control**: Prevents sender from overwhelming receiver
- **Congestion control**: Adapts sending rate based on network conditions

Socket Operations in Go:

- **net.Dial()**: Establishes outbound TCP connection
- **net.Listen()**: Creates listening socket for inbound connections
- **Accept()**: Accepts incoming connection, returns new socket
- **Read()/Write()**: Transfer data over established connection
- **Close()**: Terminates connection gracefully

MySQL Protocol Socket Management Important responsibilities in zGate include:

- **Manual packet header reading**: Every MySQL packet begins with a 4-byte header:
  - Bytes 0-2: Payload length (24-bit little-endian integer, max 16MB)
  - Byte 3: Sequence ID (increments with each packet, wraps at 255)
- **Deadline management**: Network timeouts prevent hung connections:
  - `SetReadDeadline(time.Now().Add(timeout))`: Aborts read if no data arrives
  - `SetWriteDeadline(time.Now().Add(timeout))`: Aborts write if socket buffer is full
  - Deadlines are per-operation, not absolute timeouts
- **Zero-copy packet forwarding**: When packets don't require inspection or modification:
  - Direct buffer passing between client and server sockets
  - Avoids serialization/deserialization overhead
  - Reduces memory allocations and GC pressure
  - Uses `io.Copy()` or `io.CopyN()` for efficient transfer
- **Full connection lifecycle handling**:
  - **Handshake phase**: Initial authentication and capability negotiation
  - **Command phase**: Processing client commands (queries, prepared statements)
  - **Result phase**: Streaming result sets back to client
  - **Cleanup**: Proper resource release on connection termination

Buffer Management Efficient buffer handling is critical for performance:

- **Buffer pools**: Pre-allocated buffers using `sync.Pool` to reduce GC
- **Slice capacity management**: Careful pre-allocation to avoid repeated resizing
- **Memory reuse**: Buffers are reset and returned to pool after use
- **Large packet handling**: Special handling for packets exceeding 16MB (split across multiple packets)

9.1.5 Context Propagation

The `context.Context` package provides a standardized way to carry deadlines, cancellation signals, and request-scoped values across API boundaries and between goroutines.

Context Fundamentals **Context Interface:**

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

Context Creation Functions:

- **context.Background()**: Root context, never cancelled
- **context.TODO()**: Placeholder when context is unclear
- **context.WithCancel()**: Returns context with cancel function
- **context.WithDeadline()**: Cancels at specific time
- **context.WithTimeout()**: Cancels after duration
- **context.WithValue()**: Carries request-scoped data

Context in zGate The proxy uses `context.Context` to ensure bounded execution and clean cancellation. Each incoming query obtains a context with:

- **Query ID**: Unique identifier for request tracing and correlation
- **Deadline and timeout settings**:

  - Default query timeout (e.g., 30 seconds)
  - User-specific timeout overrides
  - Inherited from client connection timeout if shorter
- **User identity and role information**:
  - Authenticated username
  - Active role assignments
  - Permission set
  - Session token information
- **Logging metadata**:
  - Source IP address
  - Client application identifier
  - Connection ID
  - Request timestamp

Cancellation Propagation Context cancellation terminates goroutines cleanly, avoiding resource leakage:

1. Client disconnects → context cancelled → all related goroutines notified
2. Query timeout exceeded → context cancelled → database connection interrupted
3. Admin kills session → context cancelled → graceful cleanup initiated
4. Shutdown signal received → root context cancelled → all sessions terminated

Best Practices in zGate:

- Always pass context as first parameter to functions
- Check `ctx.Done()` in long-running loops
- Use `select` to multiplex context cancellation with other operations
- Never store contexts in structs (pass explicitly)
- Defer cancellation function calls to prevent leaks

9.2 Database Wire Protocols (MySQL/MariaDB)

A distinguishing feature of zGate is its ability to "speak" the MySQL protocol directly, acting as a full proxy rather than a driver or middleware within the application layer.

9.2.1 MySQL Protocol Overview

The MySQL Client/Server Protocol is a binary protocol used for communication between MySQL clients and servers. It was originally designed in the 1990s and has evolved through multiple versions while maintaining backward compatibility.

Protocol Characteristics

- **Binary protocol**: Data is transmitted in compact binary format, not ASCII
- **Stateful**: Server and client maintain session state across multiple packets
- **Packet-oriented**: All communication happens in discrete packets
- **Sequential**: Packets within a command are numbered sequentially
- **Bidirectional**: Both client and server can initiate certain communications

Protocol Phases

1. **Connection Phase**:
    - Server sends initial handshake packet with capabilities and auth challenge
    - Client responds with handshake response containing credentials
    - Server sends OK or ERR packet
2. **Command Phase**:
    - Client sends command packets (queries, prepared statements, etc.)
    - Server responds with result sets, OK, or ERR packets
    - Multiple commands can be sent over same connection
3. **Termination Phase**:
    - Client sends COM_QUIT or closes connection
    - Server releases resources and closes socket

9.2.2 MySQL Packet Structure

Each MySQL packet contains a precisely defined structure that must be correctly parsed and reconstructed by zGate.

Packet Header Format

```
+-----------------+-----------------+-----------------+
| Byte 0          | Byte 1          | Byte 2          |
| Payload Length (LSB)                                |
+-----------------+-----------------+-----------------+
| Byte 3          |
| Sequence ID     |
+-----------------+
```

- **3-byte payload length** (little-endian):
  - Represents length of packet payload in bytes
  - Maximum value: 0xFFFFFF (16,777,215 bytes = 16MB - 1)
  - Length does NOT include the 4-byte header itself
  - Packets exceeding 16MB-1 are split into multiple packets
- **1-byte sequence ID**:
  - Starts at 0 for each new command
  - Increments by 1 for each packet in the sequence
  - Wraps around after 255 (rare in practice)
  - Used to detect packet loss or out-of-order delivery
  - Server and client must maintain synchronized counters
- **N-byte payload**:
  - Actual packet content (commands, result data, etc.)
  - Format depends on packet type
  - Can contain binary or text data

Large Packet Handling When payload exceeds 16MB-1 bytes:

1. First packet contains maximum payload (0xFFFFFF bytes)
2. Sequence ID increments for continuation packet(s)
3. Last packet contains remaining data (length ¡ 0xFFFFFF)
4. Empty packet (length=0) sent if payload is exact multiple of 16MB-1

Packet Validation in zGate The gateway must validate and reconstruct packets precisely:

- **Length validation**: Ensure payload length matches actual bytes read
- **Sequence synchronization**: Track and validate sequence IDs on both sides
- **Buffer management**: Allocate appropriate buffers based on payload length
- **Error detection**: Identify corrupted or malformed packets
- **Large packet assembly**: Correctly reassemble multi-packet messages

9.2.3 Command Phase Processing

The MySQL protocol defines numerous command types that clients can send to servers. zGate must understand and properly handle each command type.

Command Packet Format

```
+------------------+------------------+
| Command Byte     | Command Payload  |
| (1 byte)         | (variable)       |
+------------------+------------------+
```

Supported Commands in zGate The proxy supports various MySQL commands:

- **COM_QUERY (0x03)**: Execute textual SQL statement
  - Payload: SQL string (NOT null-terminated)
  - Most common command type
  - Response: Result set, OK, or ERR packet
  - Example: `SELECT * FROM users WHERE id=1`
- **COM_INIT_DB (0x02)**: Change default database
  - Payload: Database name string
  - Similar to SQL: `USE database_name`
  - Response: OK or ERR packet
  - Updates session state in both client and server
- **COM_STMT_PREPARE (0x16)**: Prepare SQL statement

  - Payload: SQL statement with ? placeholders
  - Server parses and returns statement ID and parameter metadata
  - Enables binary protocol for faster execution
  - Statement cached on server until explicitly closed
- **COM_STMT_EXECUTE (0x17)**: Execute prepared statement
  - Payload: Statement ID + flags + parameter values (binary encoded)
  - Uses efficient binary protocol instead of text
  - Parameters sent as native types (integers, dates, etc.)
  - Response: Result set in text or binary protocol
- **COM_STMT_CLOSE (0x19)**: Deallocate prepared statement
  - Payload: Statement ID (4 bytes)
  - No response packet (fire-and-forget)
  - Frees server resources
- **COM_QUIT (0x01)**: Close connection
  - No payload
  - No response from server
  - Graceful connection termination
- **COM_PING (0x0E)**: Test connection liveness
  - No payload
  - Response: OK packet
  - Used for keepalive and health checks
- **COM_FIELD_LIST (0x04)**: List table columns (deprecated)
- **COM_STMT_RESET (0x1A)**: Reset prepared statement
- **COM_SET_OPTION (0x1B)**: Set connection options

Command Processing in zGate Each command type has unique processing requirements:

1. **Parse command byte**: Identify command type from first payload byte
2. **Extract command payload**: Read remaining packet data
3. **Policy enforcement**: Check RBAC permissions for this command
4. **SQL rewriting**: Modify query if needed (COM_QUERY only)
5. **Forward to backend**: Send modified or original packet to database
6. **Process response**: Intercept and potentially modify result
7. **Audit logging**: Record command execution details

9.2.4 Result Set Encoding

Result sets in MySQL protocol follow a specific multi-packet format that zGate must parse to enable data masking and filtering.

Result Set Packet Sequence A typical result set consists of:

1. **Column count packet**:
   - Length-encoded integer indicating number of columns
   - Example: 0x03 indicates 3 columns in result
2. **Column definition packets** (one per column):
   - Catalog name (usually "def")
   - Schema (database) name
   - Table alias and original table name
   - Column alias and original column name
   - Character set encoding
   - Column display width
   - Column type (INT, VARCHAR, DATE, etc.)
   - Column flags (NOT NULL, PRIMARY KEY, etc.)
   - Decimal precision (for numeric types)
3. **EOF packet** (if CLIENT_DEPRECATE_EOF not set):

- Marks end of column definitions
- Contains server status flags and warning count

4. **Row data packets**:
   - One packet per row
   - Values encoded as length-encoded strings (text protocol) or native types (binary protocol)
   - NULL represented as 0xFB byte in text protocol
5. **EOF or OK packet**:
   - Marks end of result set
   - Contains affected rows, last insert ID, status flags, warnings

Length-Encoded Integers MySQL uses a variable-length encoding for integers:

- **Value ¡ 251**: Single byte containing the value
- **Value = 251 (0xFB)**: NULL value marker
- **Value = 252 (0xFC)**: Next 2 bytes contain value (little-endian)
- **Value = 253 (0xFD)**: Next 3 bytes contain value (little-endian)
- **Value = 254 (0xFE)**: Next 8 bytes contain value (little-endian)

Text vs Binary Protocol **Text Protocol (COM_QUERY):**

- All values encoded as strings
- Dates, numbers, etc. formatted as text
- Less efficient but human-readable

Binary Protocol (COM_STMT_EXECUTE):

- Values in native binary format
- NULL bitmap at start of row
- Type-specific encoding (4-byte int, 8-byte double, etc.)
- More efficient, less parsing overhead

zGate Row Interception zGate intercepts row-level data for masking and policy enforcement, requiring:

- **Parsing length-encoded integers**: To determine string/value lengths
- **Reconstructing text and binary rows**: After applying masking rules
- **Preserving column metadata**: To understand data types for correct parsing
- **Maintaining packet integrity**: Recalculating payload lengths and sequence IDs
- **Streaming processing**: Handling large result sets without buffering all rows

Example Masking Scenario:

1. Client queries: `SELECT ssn, name FROM employees`
2. zGate parses column definitions, identifies "ssn" column
3. For each row packet:
    - Parse length-encoded string for ssn value
    - Apply masking: "123-45-6789" $\rightarrow$ "XXX-XX-6789"
    - Reconstruct row packet with masked value
    - Recalculate payload length
    - Forward modified packet to client

9.2.5 SQL Parsing and AST Manipulation

To safely inject or modify SQL logic, zGate uses an SQL Abstract Syntax Tree (AST) approach rather than string manipulation.

SQL Abstract Syntax Trees An Abstract Syntax Tree is a tree representation of the syntactic structure of SQL code. Each node in the tree represents a construct in the SQL grammar.

Why AST vs String Manipulation:

- **Semantic understanding**: Parser understands SQL structure, not just text
- **Safe modification**: Changes preserve syntax validity
- **SQL injection prevention**: Prevents introduction of new SQL commands
- **Context awareness**: Distinguishes between identifiers, literals, keywords
- **Complexity handling**: Correctly processes nested queries, subqueries, CTEs

Parsing Pipeline

1. **Lexical Analysis (Tokenization)**:
   - Input: Raw SQL string
   - Process: Break into tokens (keywords, identifiers, operators, literals)
   - Output: Token stream
   - Example: `SELECT name FROM users` → [SELECT][name][FROM][users]
2. **Syntax Analysis (Parsing)**:
   - Input: Token stream
   - Process: Apply grammar rules to build AST
   - Output: AST root node
   - Detects syntax errors
3. **AST Manipulation**:
   - Traverse tree using visitor pattern
   - Inspect and modify nodes
   - Add new nodes (e.g., WHERE clauses)
   - Remove or replace nodes
4. **Code Generation (Serialization)**:
   - Input: Modified AST
   - Process: Traverse tree and generate SQL text
   - Output: Valid SQL string
   - Preserves formatting where possible

AST Node Types Common node types in SQL AST:

- **SelectStmt**: SELECT query root
- **SelectExprList**: Target list (columns to return)
- **FromClause**: Table references
- **WhereClause**: Filter conditions
- **JoinExpr**: JOIN operations

- **BinaryExpr**: Binary operations (AND, OR, =, ¡, etc.)
- **FuncCall**: Function calls (COUNT, MAX, etc.)
- **Identifier**: Table and column names
- **Literal**: String, numeric, date literals
- **Subquery**: Nested SELECT statement

AST Manipulation in zGate zGate performs node-by-node inspection for:

- **Target list modification**:
  - Removing restricted columns from SELECT list
  - Adding computed columns for audit purposes
  - Replacing sensitive columns with masked expressions
  - Example: `SELECT ssn` → `SELECT CONCAT('XXX-XX-', SUBSTR(ssn, 8)) AS ssn`
- **WHERE clause enforcement**:
  - Injecting row-level security predicates
  - Adding tenant isolation filters
  - Enforcing time-based access controls
  - Example: User can only see their own records
  - Original: `SELECT * FROM orders`
  - Modified: `SELECT * FROM orders WHERE user_id = 'alice'`
- **Table-level permission checks**:
  - Identify all referenced tables
  - Verify user has access to each table
  - Check for column-level permissions
  - Reject queries accessing forbidden tables
- **Subquery processing**:
  - Recursively process nested queries
  - Apply same policies to subqueries
  - Handle correlated subqueries correctly

- **JOIN analysis**:
  - Check permissions on all joined tables
  - Inject filters on joined tables if needed
  - Prevent information leakage through joins

Safe Rewriting Examples **Example 1: Column Masking**

```
Original SQL: SELECT ssn, name, salary FROM employees
AST Modification:
  - Locate "ssn" in SelectExprList
  - Replace with FuncCall node: mask_ssn(ssn)
Generated SQL: SELECT mask_ssn(ssn), name, mask_salary(salary)
               FROM employees
```

Example 2: Row Filtering

```
Original SQL: SELECT * FROM medical_records
AST Modification:
  - Navigate to WhereClause (or create if absent)
  - Add BinaryExpr: department = 'cardiology'
Generated SQL: SELECT * FROM medical_records
               WHERE department = 'cardiology'
```

Example 3: Table Access Control

```
Original SQL: SELECT * FROM hr.salaries
AST Modification:
  - Traverse tree, find table reference "hr.salaries"
  - Check user's table permissions
  - If denied: Return error before forwarding to database
  - If allowed: Forward unmodified or with row filters
```

Security Benefits This AST approach avoids string-based manipulation, preventing:

- **SQL injection vulnerabilities**: Cannot introduce new SQL commands
- **Malformed query errors**: Generated SQL is always syntactically valid
- **Escaping issues**: No need to manually escape quotes and special characters
- **Context confusion**: Parser understands string literals vs identifiers
- **Logic errors**: Changes preserve query semantics

9.3 Cryptography and Security Engineering

Security is the cornerstone of the system. The implementation combines modern cryptographic standards with secure design principles to protect data in transit, at rest, and during processing.

9.3.1 TLS / SSL Transport Layer Security

Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) are cryptographic protocols designed to provide secure communication over a computer network.

TLS Protocol Overview **Historical Context:**

- **SSL 1.0**: Never publicly released (Netscape, 1994)
- **SSL 2.0**: Released 1995, deprecated 2011 (security flaws)
- **SSL 3.0**: Released 1996, deprecated 2015 (POODLE attack)
- **TLS 1.0**: Released 1999, deprecated 2020
- **TLS 1.1**: Released 2006, deprecated 2020
- **TLS 1.2**: Released 2008, still widely used
- **TLS 1.3**: Released 2018, current standard

Security Properties:

- **Confidentiality**: Data encrypted using symmetric cipher
- **Integrity**: MAC (Message Authentication Code) prevents tampering
- **Authentication**: X.509 certificates verify server/client identity
- **Forward secrecy**: Session keys not derivable from long-term keys

TLS Handshake Process The TLS handshake establishes a secure session:

1. **Client Hello**:
    - Supported TLS versions
    - Cipher suites (encryption algorithms)
    - Random nonce
    - Supported extensions

2. **Server Hello**:
    - Selected TLS version
    - Selected cipher suite
    - Server random nonce
    - Server certificate (X.509)
3. **Key Exchange**:
    - Client verifies server certificate
    - Ephemeral Diffie-Hellman key exchange (TLS 1.3)
    - Or RSA key exchange (TLS 1.2)
    - Derive master secret
4. **Finished Messages**:
    - Both sides send encrypted "Finished" message
    - Proves they derived same keys
    - Handshake complete, application data can flow

TLS in zGate TLS is used to secure:

- **Client–Gateway communication**:
    - MySQL clients connect via TLS-encrypted connections
    - Optional mutual TLS (mTLS) for client authentication
    - Certificate-based authentication instead of passwords
- **Gateway–Database communication**:
    - Backend connections always use TLS
    - Verifies database server certificate
    - Protects credentials and query data in transit
- **Administrative API (HTTPS)**:
    - REST API served over HTTPS
    - Admin credentials encrypted in transit
    - API tokens protected from network sniffing

Technical Implementation Details Technical aspects in zGate include:

- **Loading X.509 certificates from PEM**:
  - PEM (Privacy Enhanced Mail) format: Base64-encoded DER certificates
  - Private keys protected with passphrase encryption
  - Certificate chain loading (intermediate + root CAs)
  - Key pair validation (public key matches private key)
- **Enforcing TLS 1.2+**:
  - Minimum version set in `tls.Config`
  - Rejects connections using SSL 3.0, TLS 1.0, TLS 1.1
  - Prevents downgrade attacks
- **Certificate chain validation**:
  - Using Go's `crypto/x509` package
  - Verifies certificate signature chain to trusted root CA
  - Checks expiration dates
  - Validates hostname/IP against certificate SAN (Subject Alternative Names)
  - Checks certificate revocation status (OCSP or CRL)
- **Cipher suite selection**:
  - Prefer AEAD ciphers (AES-GCM, ChaCha20-Poly1305)
  - Disable weak ciphers (RC4, 3DES, export ciphers)
  - Enable perfect forward secrecy (ECDHE key exchange)
  - Example strong cipher: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- **Optional client certificate verification (mTLS)**:
  - Server requests client certificate during handshake
  - Client presents certificate signed by trusted CA
  - Server validates certificate and extracts identity (CN or SAN)
  - Used for passwordless authentication
  - Common in service-to-service authentication

9.3.2 Symmetric Encryption (AES-256)

Advanced Encryption Standard (AES) is a symmetric block cipher adopted as a standard by NIST in 2001, replacing the older DES algorithm.

AES Fundamentals Algorithm Properties:

- **Block cipher**: Encrypts fixed-size blocks (128 bits)
- **Symmetric**: Same key for encryption and decryption
- **Key sizes**: 128, 192, or 256 bits
- **Rounds**: 10 (AES-128), 12 (AES-192), 14 (AES-256)
- **Speed**: Hardware-accelerated on modern CPUs (AES-NI instructions)

AES Operations:

1. **SubBytes**: Substitute each byte using S-box
2. **ShiftRows**: Rotate rows of state array
3. **MixColumns**: Linear transformation of columns
4. **AddRoundKey**: XOR with round key

AES Modes of Operation Block ciphers require a "mode" to encrypt data longer than one block:

- **ECB (Electronic Codebook)**: INSECURE, never use
  - Each block encrypted independently
  - Identical plaintexts produce identical ciphertexts
  - Reveals patterns in data
- **CBC (Cipher Block Chaining)**: Legacy, requires padding
  - Each block XORed with previous ciphertext
  - Requires padding to block boundary
  - Vulnerable to padding oracle attacks if not careful
  - Needs separate MAC for authentication
- **GCM (Galois/Counter Mode)**: Recommended

  - AEAD (Authenticated Encryption with Associated Data)
  - Provides both confidentiality and authenticity
  - No padding required
  - Parallelizable (fast)
  - Includes authentication tag to detect tampering
- **CCM, EAX, OCB**: Alternative AEAD modes

AES-256-GCM in zGate The internal SQLite datastore is encrypted using AES-256-GCM. The system uses:

- **32-byte (256-bit) encryption keys**:
  - Derived from master password using key derivation function
  - Or generated randomly for data-at-rest encryption
  - Stored in secure key management system or HSM
  - Never logged or exposed in API responses
- **CSPRNG-generated nonces**:
  - Nonce (Number Used Once) = Initialization Vector (IV)
  - 12 bytes (96 bits) for GCM mode
  - Must be unique for every encryption operation with same key
  - Generated using cryptographically secure random number generator
  - Stored alongside ciphertext (not secret)
  - Nonce reuse catastrophically breaks GCM security
- **Authentication tag**:
  - 16-byte tag appended to ciphertext
  - Verifies data integrity and authenticity
  - Prevents tampering and bit-flipping attacks
  - Decryption fails if tag doesn't match
- **Associated Data (AD)**:
  - Additional authenticated but unencrypted data
  - Example: database record ID, timestamp, version
  - Binds ciphertext to specific context

  - Prevents ciphertext from being moved/reused elsewhere

- **Key rotation support**:
  - Periodic key changes (e.g., quarterly)
  - Re-encrypt data with new keys
  - Multiple active keys during transition period
  - Track which key version encrypted each record
  - Old keys archived securely for decryption of historical data

Encryption Process

```
Input: Plaintext, Key (32 bytes), Nonce (12 bytes), AD
Process:
  1. Generate random 12-byte nonce
  2. Create GCM cipher instance with key
  3. Seal(nonce, plaintext, AD) → ciphertext || tag
  4. Store: nonce || ciphertext || tag
Output: Encrypted blob
```

Decryption Process

```
Input: Encrypted blob (nonce || ciphertext || tag), Key, AD
Process:
  1. Extract nonce from first 12 bytes
  2. Extract tag from last 16 bytes
  3. Extract ciphertext from middle
  4. Open(nonce, ciphertext || tag, AD) → plaintext
  5. Verify tag matches (automatic in GCM)
Output: Plaintext (or error if tampered)
```

9.3.3 Secure Hashing

Cryptographic hash functions are one-way functions that map arbitrary-size input to fixed-size output. They're essential for password storage and integrity verification.

Hash Function Properties A secure hash function must be:

- **Deterministic**: Same input always produces same output
- **Quick to compute**: Hash calculation should be fast

- **Pre-image resistance**: Computationally infeasible to reverse
- **Collision resistance**: Infeasible to find two inputs with same hash
- **Avalanche effect**: Small input change drastically changes output

bcrypt for Password Hashing Passwords and authentication tokens use:

- **bcrypt algorithm**:
  - Based on Blowfish cipher
  - Designed specifically for password hashing (1999)
  - Adaptive: Adjustable work factor resists brute force
  - Includes automatic salt generation
  - Output format: `$2a$10$somerandomsalt22charactershashvalue31characters`
- **Cost factor explanation**:
  - `$2a$10$...` means cost factor = 10
  - Cost factor N means 2^N iterations
  - Cost 10 = 1,024 iterations (100ms on modern CPU)
  - Cost 12 = 4,096 iterations (400ms)
  - Higher cost = slower hashing = harder to brute force
  - Recommended: 10-12 for user passwords (balance security/UX)
- **Salt generation**:
  - 128-bit (16-byte) random salt
  - Unique salt for every password
  - Prevents rainbow table attacks
  - Salt stored in output string (not secret)
  - Format: Base64-encoded 22-character string
- **Verification process**:
  - Extract salt and cost from stored hash
  - Hash provided password with same salt and cost
  - Compare resulting hash with stored hash
  - Constant-time comparison to prevent timing attacks

SHA-256 for Token Hashing

- **SHA-256 (Secure Hash Algorithm 256-bit)**:
  - Part of SHA-2 family designed by NSA (2001)
  - Produces 256-bit (32-byte) hash
  - Widely used, extensively analyzed
  - Fast computation (important for high-volume operations)
  - Not suitable for passwords (too fast, enables brute force)
- **Use case in zGate**:
  - Hash refresh tokens before storing
  - Client keeps original token
  - Database stores SHA-256(token)
  - If database compromised, attacker cannot use tokens
  - Similar to session ID hashing
- **Token verification**:
  - Client sends token
  - Server computes SHA-256(token)
  - Compare hash with stored value
  - Grant access if match

Security Principle No plaintext-sensitive values are stored at any time:

- Passwords: Stored as bcrypt hashes
- Refresh tokens: Stored as SHA-256 hashes
- API keys: Stored as bcrypt hashes
- Database credentials: Encrypted with AES-256-GCM
- Secrets: Encrypted at rest, decrypted only in memory when needed

9.3.4 JWT Authentication

JSON Web Tokens (JWT) are a compact, URL-safe means of representing claims to be transferred between two parties.

JWT Structure A JWT consists of three Base64URL-encoded parts separated by dots:

```
header.payload.signature
```

```
Example:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Header (Algorithm and Token Type):

```
{
  "alg": "HS256",      // HMAC-SHA256
  "typ": "JWT"          // Token type
}
```

Payload (Claims):

```
{
  "sub": "1234567890",  // Subject (user ID)
  "name": "John Doe",    // Custom claim
  "iat": 1516239022,     // Issued at (Unix timestamp)
  "exp": 1516242622      // Expiration (Unix timestamp)
}
```

Signature:

```
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  secret_key
)
```

JWT in zGate Admin API The admin API uses JSON Web Tokens with:

- **HMAC-SHA256 signing**:
  - Symmetric signing algorithm
  - Secret key shared between issuer and verifier
  - Alternative: RS256 (RSA) for asymmetric signing
  - Signature prevents token tampering
  - Any modification invalidates signature

- **Short-lived access tokens**:
  - Typical lifetime: 15 minutes
  - Contains user identity and permissions
  - Used for API authentication
  - Sent in Authorization header: `Bearer <token>`
  - Short lifetime limits damage if compromised
  - No need to track/revoke (expires quickly)
- **Long-lived refresh tokens**:
  - Typical lifetime: 30-90 days
  - Used to obtain new access tokens
  - Stored hashed in database (SHA-256)
  - Can be revoked (blacklisted)
  - Sent via secure cookie or separate header
  - Refresh endpoint: Exchange refresh token for new access token
- **Embedded role and permission claims**:
  - `sub`: User ID
  - `username`: Human-readable username
  - `roles`: Array of assigned roles
  - `permissions`: Array of granted permissions
  - `iat`: Issued at timestamp
  - `exp`: Expiration timestamp
  - `jti`: JWT ID (unique identifier for revocation)

Token Lifecycle

1. **Login**:
   - User provides username/password
   - Server validates credentials
   - Server generates access + refresh tokens
   - Returns both tokens to client
2. **API Request**:

- Client sends access token in Authorization header
- Middleware validates token signature
- Middleware checks expiration
- Middleware extracts permissions from claims
- Request proceeds if authorized

3. **Token Refresh**:
   - Access token expires
   - Client sends refresh token to /refresh endpoint
   - Server validates refresh token hash
   - Server issues new access token (and optionally new refresh token)
   - Client continues using new access token
4. **Revocation**:
   - Admin revokes user's refresh token
   - Refresh token hash removed from database
   - User cannot obtain new access tokens
   - Existing access tokens expire naturally (15min)
   - For immediate revocation: Maintain token blacklist

Middleware Validation Middleware validates token integrity and permissions before allowing administrative actions:

1. Extract token from Authorization header
2. Verify signature using secret key
3. Check expiration timestamp (`exp` claim)
4. Check issued-at timestamp (`iat` claim) for clock skew
5. Extract user identity and permissions
6. Check if requested action is allowed for user's role
7. Reject request if any validation fails
8. Attach user context to request for downstream handlers

9.4 Software Architecture and Design Patterns

The gateway employs well-defined architectural patterns to maintain extensibility, modularity, and clarity of responsibility.

9.4.1 Proxy Pattern

The Proxy Pattern provides a surrogate or placeholder object that controls access to another object. In zGate, the gateway acts as a full MySQL proxy.

Proxy Pattern Fundamentals **Intent:**

- Control access to an object
- Add additional functionality without modifying original object
- Lazy initialization, logging, access control, caching

Pattern Structure:

- **Subject**: Interface defining operations (MySQL protocol)
- **RealSubject**: Actual object (MySQL database server)
- **Proxy**: Controls access to RealSubject (zGate)
- **Client**: Uses proxy transparently (database client application)

zGate as MySQL Proxy zGate acts as a full MySQL proxy, maintaining:

- **Session state**:
  - **Transaction flags**: IN_TRANSACTION, AUTOCOMMIT status
  - **Character set**: Client and connection character sets
  - **Selected database**: Current default schema
  - **Session variables**: SQL_MODE, time zone, etc.
  - **Connection attributes**: Client version, OS, application name
- **Prepared statement metadata**:
  - Mapping of statement IDs to SQL text
  - Parameter count and types
  - Column metadata for result sets

  - Statement cached on both client and server side
  - Proper cleanup on COM_STMT_CLOSE

- **Sequence ID tracking**:
  - Separate sequence counters for client-side and server-side
  - Reset to 0 at start of each command
  - Increment for each packet sent/received
  - Critical for protocol correctness
  - Mismatch causes protocol errors

- **Connection pooling**:
  - Reuse database connections across client sessions
  - Reduce connection establishment overhead
  - Maintain pool of ready connections
  - Health checking idle connections

Proxy Benefits This isolates the clients from the database while enabling:

- **Transparent enforcement**: Clients unaware of proxy presence
- **Zero Trust controls**: Every query subject to policy
- **Centralized security**: Single enforcement point
- **Audit logging**: Complete visibility into database access
- **Protocol translation**: Can add features database doesn't support
- **Load balancing**: Distribute queries across multiple backends

9.4.2 Interceptor Chain Pattern

The Interceptor (Chain of Responsibility) Pattern allows multiple objects to handle a request without coupling the sender to specific receivers.

Pattern Overview Intent:

- Decouple request senders from receivers
- Allow multiple handlers to process request
- Dynamically configure processing pipeline

Pattern Structure:

- **Handler interface**: Defines processing method
- **Concrete handlers**: Implement specific processing logic
- **Chain**: Ordered sequence of handlers
- **Client**: Initiates request into chain

Interceptor Chain in zGate A configurable chain of interceptors processes every query:

- **BeforeQuery interceptors**:
  - **SQL parsing**: Lex and parse SQL into AST
  - **Permission checks**: Verify table and column access
  - **SQL rewriting**: Inject WHERE clauses, modify SELECT lists
  - **Query validation**: Check for blacklisted patterns
  - **Rate limiting**: Enforce query rate limits per user
  - **Query complexity analysis**: Reject overly expensive queries

  Example interceptors:

  - RBACInterceptor: Role-based access control
  - RowLevelSecurityInterceptor: Inject row filters
  - AuditPreInterceptor: Log original query
- **OnRow interceptors**:
  - **Column masking**: Replace sensitive data with masked values
  - **Row filtering**: Skip rows based on policy
  - **Data transformation**: Encrypt/decrypt column values
  - **Value substitution**: Replace values based on rules

- **Redaction**: Remove columns entirely from results

Example interceptors:

- MaskingInterceptor: Apply data masking rules
- EncryptionInterceptor: Client-side encryption
- SensitiveDataInterceptor: Redact PII

- **AfterQuery interceptors**:
  - **Audit logging**: Record query execution details
  - **Anomaly detection**: Identify suspicious patterns
  - **Performance monitoring**: Track query latency
  - **Alert triggers**: Notify on policy violations
  - **Metrics collection**: Aggregate statistics

  Example interceptors:

  - AuditPostInterceptor: Complete audit records
  - MetricsInterceptor: Update query statistics
  - AlertInterceptor: Trigger security alerts

Chain Execution Flow

1. Client sends query to zGate
2. Query enters interceptor chain
3. Each BeforeQuery interceptor processes in order
4. If any interceptor rejects: Return error, skip remaining chain
5. Forward (possibly rewritten) query to database
6. Database returns result set
7. Each OnRow interceptor processes every row
8. Forward (possibly modified) rows to client
9. Each AfterQuery interceptor processes
10. Complete request handling

Configuration and Extensibility This modularity enables new security features without modifying the core proxy loop:

- **Plugin architecture**: New interceptors implement standard interface
- **Runtime configuration**: Enable/disable interceptors via config file
- **Ordering control**: Specify execution order in configuration
- **Conditional execution**: Interceptors can check context and skip
- **Error handling**: Interceptors can halt chain or allow continuation
- **Performance optimization**: Short-circuit chain when possible

9.5 Embedded Storage (SQLite)

SQLite is a C-language library that implements a small, fast, self-contained, full-featured SQL database engine. It is the most deployed database engine in the world.

9.5.1 SQLite Overview

SQLite Characteristics **Why SQLite for zGate:**

- **Self-contained**: No separate server process required
- **Zero-configuration**: No installation or admin needed
- **Single file**: Entire database in one file on disk
- **Cross-platform**: Works on all major operating systems
- **Reliable**: ACID-compliant, crash-safe transactions
- **Small footprint**: 600KB compiled library
- **Fast**: Optimized for local database operations
- **Public domain**: No licensing concerns

SQLite Architecture:

- **SQL compiler**: Parses SQL into bytecode
- **Virtual Machine**: Executes bytecode
- **B-tree**: Data structure for tables and indexes

- **Pager**: Manages database pages and caching
- **OS Interface**: Platform-specific file I/O

Transaction Support:

- **ACID properties**: Atomicity, Consistency, Isolation, Durability
- **Write-Ahead Log (WAL)**: Improves concurrency and performance
- **Journaling**: Rollback journal for crash recovery
- **Locking**: Database-level locking (readers don't block readers)

Encryption with SQLCipher zGate uses SQLite with SQLCipher for transparent database encryption:

- **SQLCipher**: Open-source extension for encrypted SQLite databases
- **Transparent encryption**: Entire database file encrypted
- **AES-256-CBC**: Encryption algorithm used
- **PBKDF2**: Key derivation from passphrase
- **HMAC-SHA1**: Page authentication
- **No plaintext on disk**: All data encrypted at rest

9.5.2 Schema Design

The RBAC (Role-Based Access Control) system includes carefully designed tables with appropriate constraints and relationships.

Core Tables

- **users table**:

```
CREATE TABLE users (
    user_id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    email TEXT UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_active BOOLEAN DEFAULT 1,
```

```
    last_login TIMESTAMP,
    failed_login_attempts INTEGER DEFAULT 0
);
```

Stores user account information and authentication data.

- **roles table**:

```
CREATE TABLE roles (
    role_id INTEGER PRIMARY KEY AUTOINCREMENT,
    role_name TEXT UNIQUE NOT NULL,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_system_role BOOLEAN DEFAULT 0
);
```

Defines available roles (e.g., admin, analyst, developer).

- **permissions table**:

```
CREATE TABLE permissions (
    permission_id INTEGER PRIMARY KEY AUTOINCREMENT,
    resource TEXT NOT NULL,
    action TEXT NOT NULL,
    description TEXT,
    UNIQUE(resource, action)
);
```

Defines granular permissions (e.g., database:read, table:write).

- **role_permissions table** (junction table):

```
CREATE TABLE role_permissions (
    role_id INTEGER NOT NULL,
    permission_id INTEGER NOT NULL,
    PRIMARY KEY (role_id, permission_id),
    FOREIGN KEY (role_id) REFERENCES roles(role_id)
        ON DELETE CASCADE,
    FOREIGN KEY (permission_id) REFERENCES permissions(permission_id)
```

```
        ON DELETE CASCADE
);
```

Maps which permissions each role has (many-to-many relationship).

- **user_roles table** (junction table):

```
CREATE TABLE user_roles (
    user_id INTEGER NOT NULL,
    role_id INTEGER NOT NULL,
    granted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    granted_by INTEGER,
    PRIMARY KEY (user_id, role_id),
    FOREIGN KEY (user_id) REFERENCES users(user_id)
        ON DELETE CASCADE,
    FOREIGN KEY (role_id) REFERENCES roles(role_id)
        ON DELETE CASCADE
);
```

Assigns roles to users (many-to-many relationship).

- **user_custom_permissions table**:

```
CREATE TABLE user_custom_permissions (
    user_id INTEGER NOT NULL,
    permission_id INTEGER NOT NULL,
    is_grant BOOLEAN NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, permission_id),
    FOREIGN KEY (user_id) REFERENCES users(user_id)
        ON DELETE CASCADE,
    FOREIGN KEY (permission_id) REFERENCES permissions(permission_id)
        ON DELETE CASCADE
);
```

Allows granting/revoking individual permissions to users, overriding role permissions.

- **audit_logs table**:

```
CREATE TABLE audit_logs (
    log_id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    user_id INTEGER,
    username TEXT NOT NULL,
    client_ip TEXT,
    database_name TEXT,
    original_query TEXT NOT NULL,
    rewritten_query TEXT,
    query_result TEXT,
    rows_affected INTEGER,
    execution_time_ms INTEGER,
    policy_action TEXT,
    was_blocked BOOLEAN DEFAULT 0,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
        ON DELETE SET NULL
);
CREATE INDEX idx_audit_timestamp ON audit_logs(timestamp);
CREATE INDEX idx_audit_user ON audit_logs(user_id);
CREATE INDEX idx_audit_blocked ON audit_logs(was_blocked);
```

Comprehensive audit trail of all database access.

Referential Integrity Foreign key constraints ensure referential integrity:

- **ON DELETE CASCADE**: When parent record deleted, child records automatically deleted
  - Example: Delete role → all role_permissions entries deleted
  - Example: Delete user → all user_roles entries deleted
- **ON DELETE SET NULL**: When parent deleted, foreign key in child set to NULL
  - Example: Delete user → audit_logs.user_id set to NULL (preserve audit history)
- **Constraint enforcement**: SQLite validates all constraints on INSERT/UPDATE/DELETE
- **Transaction rollback**: Constraint violations roll back entire transaction

Cascading deletes help maintain consistency by automatically cleaning up related records.

9.5.3 Audit Logging

Every query generates an audit log entry, providing comprehensive visibility and compliance support.

Audit Log Contents Each audit record stores:

- **User identity**:
  - User ID and username
  - Authentication method used
  - User's active roles at time of query
- **Timestamp**:
  - High-precision timestamp (microseconds)
  - Time zone information
  - Server time vs client time
- **Context information**:
  - Client IP address
  - Application name/version
  - Session ID
  - Database name
- **Original query**:
  - Exact SQL as submitted by client
  - Parameter values (for prepared statements)
  - Query type (SELECT, INSERT, UPDATE, DELETE, DDL)
- **Rewritten query**:
  - SQL after policy enforcement modifications
  - Added WHERE clauses
  - Column masking expressions
  - Shows what was actually executed

- **Execution metrics**:
  - Rows affected or returned
  - Execution time in milliseconds
  - Success or error status
  - Error message if failed
- **Policy action taken**:
  - ALLOW: Query permitted and forwarded
  - MODIFY: Query rewritten before execution
  - MASK: Results masked on return
  - BLOCK: Query rejected
  - Reason for action (which policy triggered)

Audit Log Use Cases Logs are essential for:

- **Compliance and regulatory requirements**:
  - GDPR: Demonstrating data access controls
  - HIPAA: Healthcare data access audit trails
  - SOX: Financial data access logging
  - PCI-DSS: Cardholder data access tracking
- **Forensic analysis**:
  - Investigating security incidents
  - Identifying data breach scope
  - Tracking unauthorized access attempts
  - Timeline reconstruction
- **Anomaly detection**:
  - Unusual query patterns
  - Off-hours access
  - Large data extractions
  - Permission escalation attempts
- **Performance analysis**:

  - Identifying slow queries
  - Query frequency analysis
  - User behavior patterns

- **User behavior analytics**:
  - Establish baselines for normal activity
  - Detect insider threats
  - Monitor privileged user actions

Audit Log Management

- **Retention policies**: Configure how long logs are kept
- **Log rotation**: Archive old logs to separate storage
- **Tamper protection**: Logs cryptographically signed
- **Export capabilities**: Export to SIEM or log aggregation systems
- **Query interface**: Search and filter audit logs via API

9.6 REST API Design

The administrative control plane is implemented using a REST (Representational State Transfer) API built with Gorilla Mux, a popular HTTP router for Go.

9.6.1 REST Architectural Style

REST Principles REST is an architectural style for distributed systems defined by Roy Fielding in his 2000 doctoral dissertation. Key constraints include:

- **Client-Server**: Separation of concerns between UI and data storage
- **Stateless**: Each request contains all necessary information
- **Cacheable**: Responses explicitly indicate if they can be cached
- **Uniform Interface**: Standardized communication protocol (HTTP)
- **Layered System**: Intermediaries (proxies, gateways) can be transparent
- **Code on Demand (optional)**: Servers can extend client functionality

HTTP Methods in REST

- **GET**: Retrieve resource (idempotent, safe, cacheable)
- **POST**: Create new resource (not idempotent)
- **PUT**: Update/replace resource (idempotent)
- **PATCH**: Partial update (idempotent)
- **DELETE**: Remove resource (idempotent)
- **HEAD**: Get headers only (like GET without body)
- **OPTIONS**: Describe communication options (CORS)

HTTP Status Codes

- **2xx Success**:
  - 200 OK: Request succeeded
  - 201 Created: Resource created
  - 204 No Content: Success, no response body
- **3xx Redirection**:
  - 301 Moved Permanently
  - 304 Not Modified: Use cached version
- **4xx Client Errors**:
  - 400 Bad Request: Invalid syntax
  - 401 Unauthorized: Authentication required
  - 403 Forbidden: Insufficient permissions
  - 404 Not Found: Resource doesn't exist
  - 409 Conflict: Request conflicts with current state
  - 422 Unprocessable Entity: Validation failed
- **5xx Server Errors**:
  - 500 Internal Server Error
  - 503 Service Unavailable: Temporary overload

9.6.2 API Characteristics

Gorilla Mux Router Gorilla Mux is a powerful HTTP router for Go applications:

- **Path and method-based routing**:
  - Match routes by HTTP method and URL path
  - Path variables: `/users/{id}`
  - Query parameters: `/users?role=admin`
  - Host-based routing: Different handlers for different domains
- **Middleware support**:
  - Chain middleware functions
  - Execute before/after route handlers
  - Common middleware: logging, auth, CORS, rate limiting
- **Subrouters**:
  - Group related routes
  - Apply middleware to route groups
  - Organize API versions (e.g., /api/v1, /api/v2)

Request/Response Format

- **JSON request/response formatting**:
  - All request bodies use JSON
  - All responses return JSON (even errors)
  - Content-Type: application/json header required
  - Consistent error response format
- **Example Request**:

```
POST /api/v1/users
Content-Type: application/json
Authorization: Bearer eyJhbGc...

{
  "username": "alice",
  "email": "alice@example.com",
  "roles": ["analyst"]
}
```

- **Example Success Response**:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "user_id": 42,
  "username": "alice",
  "email": "alice@example.com",
  "roles": ["analyst"],
  "created_at": "2025-12-12T10:30:00Z"
}
```

- **Example Error Response**:

```
HTTP/1.1 403 Forbidden
Content-Type: application/json

{
  "error": "Insufficient permissions",
  "code": "FORBIDDEN",
  "details": "User lacks 'user:create' permission"
}
```

Authentication and Authorization

- **JWT-based authorization middleware**:
  - Extracts token from Authorization header
  - Validates token signature and expiration
  - Checks required permissions for endpoint
  - Attaches user context to request
- **Permission-based access control**:
  - Each endpoint requires specific permissions
  - Example: POST /users requires "user:create"
  - Permissions checked against JWT claims
  - Granular control over API access

Transport Security

- **TLS for transport security**:
  - All API traffic over HTTPS
  - TLS 1.2+ enforced
  - HSTS header: Strict-Transport-Security
  - Certificate pinning option for mobile clients

9.6.3 Administrative Capabilities

The API exposes comprehensive management endpoints.

User and Role Management

- **User operations**:
  - GET /api/v1/users - List all users
  - GET /api/v1/users/{id} - Get user details
  - POST /api/v1/users - Create new user
  - PUT /api/v1/users/{id} - Update user
  - DELETE /api/v1/users/{id} - Delete user
  - POST /api/v1/users/{id}/roles - Assign roles
  - DELETE /api/v1/users/{id}/roles/{roleId} - Revoke role
- **Role operations**:
  - GET /api/v1/roles - List all roles
  - POST /api/v1/roles - Create role
  - PUT /api/v1/roles/{id} - Update role
  - DELETE /api/v1/roles/{id} - Delete role
  - GET /api/v1/roles/{id}/permissions - List role permissions
  - POST /api/v1/roles/{id}/permissions - Grant permissions

Security Operations

- **Key and certificate rotation**:
  - POST /api/v1/crypto/rotate-key - Rotate encryption keys

  - POST /api/v1/crypto/rotate-cert - Update TLS certificates
  - GET /api/v1/crypto/status - View key/cert status

- **Token management**:
  - POST /api/v1/auth/revoke - Revoke refresh token
  - DELETE /api/v1/auth/sessions/{id} - Kill user session
  - GET /api/v1/auth/sessions - List active sessions

Policy Management

- **Masking rules**:
  - GET /api/v1/policies/masking - List masking policies
  - POST /api/v1/policies/masking - Create masking rule
  - PUT /api/v1/policies/masking/{id} - Update rule
  - DELETE /api/v1/policies/masking/{id} - Delete rule
- **RBAC policies**:
  - GET /api/v1/policies/rbac - List access policies
  - POST /api/v1/policies/rbac - Create policy
  - PUT /api/v1/policies/rbac/{id} - Update policy

Monitoring and Administration

- **Active session monitoring**:
  - GET /api/v1/sessions - List active connections
  - GET /api/v1/sessions/{id} - Session details
  - DELETE /api/v1/sessions/{id} - Kill session
- **Audit log access**:
  - GET /api/v1/audit - Query audit logs
  - GET /api/v1/audit/export - Export logs (CSV, JSON)
  - POST /api/v1/audit/search - Advanced search
- **System health**:
  - GET /api/v1/health - Health check endpoint
  - GET /api/v1/metrics - Prometheus metrics
  - GET /api/v1/stats - System statistics

10. System Architecture

10.1 High-Level Architecture Diagram

10.2 Main System Components

10.3 Component Communication Flow

10.4 Tech Stack Summary

11. Detailed Architecture of the Proxy

11.1 Connection Lifecycle

11.2 Authentication Flow

11.3 Query Filtering Flow

11.4 Policy Enforcement Flow

11.5 Session Monitoring Flow

11.6 User → Gateway → Database Diagram

12. High-Level Data Flow Diagrams

12.1 Authentication Flow Diagram

12.2 Query Filtering Diagram

12.3 Logging & Auditing Flow Diagram

13. Technology Justification

13.1 Why Go

13.2 Why Node.js / TS / React

13.3 Why mTLS (and why TCP is temporary)

13.4 Why SQLite / Internal Storage

13.5 Design Decision Summary

14. Prototype – Semester 1

14.1 Implemented Features

14.2 Screenshots (CLI & Dashboard)

14.3 What Works vs What Doesn't

14.4 Technical Decisions Made

14.5 Implementation Challenges

15. Development Methodology

15.1 Agile Method

15.2 Meeting Structure

15.3 Collaboration Tools

15.4 Documentation & Observability

16. Task Tracking

16.1 Team Task Tracking (Actual Examples)

16.2 Supervisor Tracking Logs

16.3 Blockers, Risks & Resolution Notes

17. Milestones

17.1 Term 1 Milestone Roadmap

17.1.1 Milestone 1

17.1.2 Milestone 2

17.1.3 Milestone 3

17.1.4 Milestone 4

17.1.5 Milestone 5

17.2 Timeline Chart (Gantt-like)

18. Threat Model & Security Considerations

18.1 Threat Model

18.2 Risks & Attack Vectors

18.3 Mitigation Techniques

18.4 Why Zero Trust is Needed

19. Roadmap for Term 2

19.1 Remaining Features

19.2 Architecture Improvements

19.3 Performance Goals

19.4 Testing & Validation Plan

20. Team Contribution

20.1 Overview of Contribution Approach

20.2 Individual Contributions

21. Expected Outcomes

22. Conclusion

22.1 Restated Purpose

22.2 Summary of Achievements

22.3 Importance & Contribution

22.4 Transition to Next Semester

References

A. Glossary

B. Dashboard Mockups