

Faculty of Engineering – Ain Shams University

Computer and Systems Engineering Department

zGate Gateway: A Zero Trust Database Access Proxy

Graduation Project Thesis

Supervisor:

Dr. Mohammed Sobh

Academic Year 2025–2026

Acknowledgments

We would like to thank...

Abstract

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

Contents

1	Team Information	8
1.1	Team Members	8
1.2	Roles & Responsibilities	8
2	Introduction & Problem Definition	9
2.1	Introduction	9
2.2	Problem Statement	9
2.3	Gap in Existing Solutions	9
2.4	Why Zero Trust for Databases is Different	9
3	Project Definition	10
3.1	Project Definition & Scope	10
3.2	Objectives	10
3.3	Expected Academic Contribution	10
4	Requirements Engineering	11
4.1	Functional Requirements	11
4.2	Non-Functional Requirements	11
4.3	Actors & Use Cases	11
4.4	Use Case Diagrams	11
4.5	User Stories	11
5	Proposed Solution	12
5.1	Overview of the Proposed Solution	12
5.2	Solution Architecture Layers	12
5.2.1	Authentication Layer	12
5.2.2	Proxy Layer	12
5.2.3	Policy Engine Layer	12
5.2.4	Data Protection Layer	12
5.2.5	Observability Layer	12
5.3	Term 1 vs Term 2 Features	12
5.4	Feature List	12

6	Alignment with International Standards	13
6.1	PCI DSS	13
6.2	HIPAA	13
6.3	GDPR	13
6.4	ISO 27001	13
7	Competitor & Market Analysis	14
7.1	Competitor Analysis	14
7.1.1	Comparison Table	14
7.1.2	What Competitors Lack	14
7.2	Market Research	14
7.2.1	Market Overview	14
7.2.2	Zero Trust Demand	14
7.2.3	Market Challenges & Needs	14
7.2.4	Regulatory Drivers	14
7.2.5	Trends & Opportunities	14
7.2.6	Landscape Summary	14
8	Scientific Research & Literature Review	15
8.1	Paper 1	15
8.2	Paper 2	15
8.3	Paper 3	15
8.4	Paper 4	15
8.5	Paper 5	15
8.6	Paper 6	15
8.7	Paper 7	15
9	Technical Background	16
10	System Architecture	17
10.1	High-Level Architecture Diagram	17
10.2	Main System Components	17
10.3	Component Communication Flow	17
10.4	Tech Stack Summary	17
11	Detailed Architecture of the Proxy	18
11.1	Connection Lifecycle	18
11.2	Authentication Flow	18
11.3	Query Filtering Flow	18
11.4	Policy Enforcement Flow	18

11.5	Session Monitoring Flow	18
11.6	User → Gateway → Database Diagram	18
12	High-Level Data Flow Diagrams	19
12.1	Authentication Flow Diagram	19
12.2	Query Filtering Diagram	19
12.3	Logging & Auditing Flow Diagram	19
13	Technology Justification	20
13.1	Why Go	20
13.1.1	Performance & Execution Model	20
13.1.2	Goroutine Concurrency Model	20
13.1.3	Production-Grade Standard Library	21
13.1.4	Security & Cryptography Ecosystem	21
13.1.5	Deployment Simplicity	21
13.1.6	Developer Experience & Code Maintainability	22
13.2	Why Node.js / TS / React	22
13.3	Why SQLite for Internal Storage	22
13.3.1	Zero-Downtime Updates	22
13.3.2	Efficiency & Memory Management	23
13.3.3	API Integration & WebUI Compatibility	23
13.3.4	Enhanced Security Through Data-at-Rest Encryption	24
13.4	Why mTLS (and why TCP is temporary)	24
13.5	Design Decision Summary	24
14	Prototype – Semester 1	25
14.1	Implemented Features	25
14.2	Screenshots (CLI & Dashboard)	25
14.3	What Works vs What Doesn't	25
14.4	Technical Decisions Made	25
14.5	Implementation Challenges	25
15	Development Methodology	26
15.1	Agile Scrum Framework	26
15.2	Meeting Structure	26
15.2.1	Weekly Kick-off Meeting	26
15.2.2	Daily Stand-up Meeting	27
15.2.3	Sprint Review (Weekly Supervisor Meeting)	27
15.3	Collaboration Tools	27
15.4	Documentation & Observability	28

16 Task Tracking	29
16.1 Team Task Tracking (Actual Examples)	29
16.2 Supervisor Tracking Logs	29
16.3 Blockers, Risks & Resolution Notes	29
17 Milestones	30
17.1 Term 1 Milestone Roadmap	30
17.1.1 Milestone 1	30
17.1.2 Milestone 2	30
17.1.3 Milestone 3	30
17.1.4 Milestone 4	30
17.1.5 Milestone 5	30
17.2 Timeline Chart (Gantt-like)	30
18 Threat Model & Security Considerations	31
18.1 Threat Model	31
18.2 Risks & Attack Vectors	31
18.3 Mitigation Techniques	31
18.4 Why Zero Trust is Needed	31
19 Roadmap for Term 2	32
19.1 Remaining Features	32
19.2 Architecture Improvements	32
19.3 Performance Goals	32
19.4 Testing & Validation Plan	32
20 Team Contribution	33
20.1 Overview of Contribution Approach	33
20.2 Individual Contributions	33
21 Expected Outcomes	34
22 Conclusion	35
22.1 Restated Purpose	35
22.2 Summary of Achievements	35
22.3 Importance & Contribution	35
22.4 Transition to Next Semester	35
A Glossary	37
B Dashboard Mockups	38

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

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 \rightarrow Gateway \rightarrow 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

The selection of Go (Golang) as the primary language for implementing the zGate Gateway proxy was driven by several technical requirements unique to high-performance, security-critical network infrastructure. Unlike interpreted languages, Go provides the performance characteristics and concurrency model essential for building a production-grade database access proxy.

13.1.1 Performance & Execution Model

Go is a compiled language that translates source code directly into machine code, unlike interpreted languages such as Python that execute code line-by-line at runtime. This fundamental architectural difference results in significantly lower latency and higher throughput—critical metrics for a proxy that sits between clients and database servers, where every millisecond of added latency compounds across thousands of queries.

Additionally, Go applications exhibit substantially lower memory overhead compared to equivalent implementations in interpreted languages. This efficiency allows the gateway to handle significantly more concurrent traffic on the same hardware resources, directly impacting the scalability and cost-effectiveness of the system.

13.1.2 Goroutine Concurrency Model

The most significant advantage of Go for proxy server implementation lies in its concurrency model based on goroutines. This feature is crucial for handling the massive number of simultaneous database connections that a production gateway must support.

Traditional threading models impose significant memory overhead, as each thread typically consumes megabytes of stack space. In contrast, goroutines are extremely lightweight, consuming only approximately 2KB of initial stack space. This efficiency enables the gateway to spawn tens of thousands of goroutines to handle concurrent proxy connections without exhausting system resources—a capability that would be impractical with traditional threading approaches.

Furthermore, Go provides *channels* as a first-class language construct for safely communicating between concurrent processes. This built-in mechanism eliminates the complex locking patterns and race conditions that plague concurrent programming in other languages, making the codebase more maintainable and less prone to subtle concurrency bugs.

13.1.3 Production-Grade Standard Library

Go was designed by Google specifically for building networked systems and internet services. The standard library's `net` and `net/http` packages are robust, secure, and production-ready out of the box, eliminating the need for heavy external frameworks.

The `net` library provides fine-grained control over TCP socket behavior, including precise management of timeouts, deadlines, and keep-alive settings. This low-level control is essential for a custom proxy that must intelligently manage traffic flow, implement connection pooling, and enforce session policies at the transport layer.

13.1.4 Security & Cryptography Ecosystem

Security is paramount for a Zero Trust database access gateway, and Go's cryptography ecosystem is exceptionally well-suited to this requirement. The `crypto/tls` package in Go is considered one of the industry's best implementations of SSL/TLS protocols, with built-in support for modern standards including TLS 1.3 by default.

Critically, Go is a memory-safe language despite its performance characteristics comparable to C or C++. The language's garbage collector prevents common security vulnerabilities such as buffer overflows, use-after-free errors, and memory leaks—all of which would be catastrophic in a security gateway positioned between untrusted clients and sensitive database systems.

13.1.5 Deployment Simplicity

Go's compilation model produces a single static binary that bundles all dependencies and libraries. This characteristic eliminates "dependency hell" and dramatically simplifies deployment operations.

To deploy the zGate Gateway to a server or container, operators simply copy a single executable file—no runtime installation, no package manager invocations, and no version conflict resolution required. This simplicity reduces the attack surface, minimizes deployment complexity, and ensures consistent behavior across different target environments.

13.1.6 Developer Experience & Code Maintainability

Go’s minimalist design philosophy emphasizes simplicity and readability, featuring a deliberately small syntax with no complex inheritance hierarchies or implicit behaviors. This characteristic is particularly valuable for security-critical software, where code auditability is essential. If code is easy to read, it is correspondingly easier to audit for security flaws and verify correctness.

As a statically typed language with a powerful type system, Go enables the compiler to catch many classes of bugs—including type mismatches, null pointer dereferences, and interface violations—before code execution. This compile-time validation substantially reduces the likelihood of runtime errors in production environments, contributing to overall system reliability.

13.2 Why Node.js / TS / React

13.3 Why SQLite for Internal Storage

The evolution of the zGate Gateway’s configuration management architecture represents a critical design decision that directly impacts system reliability, security, and operational efficiency. Initially, the system utilized YAML files for storing configuration data, including database connection strings, user permissions, and policy rules. However, this approach proved insufficient for a production-grade security gateway, leading to the adoption of SQLite as the internal storage mechanism.

13.3.1 Zero-Downtime Updates

The most significant limitation of file-based configuration was the requirement for application restarts to apply changes. In the YAML-based implementation, configuration data was loaded into memory only during application startup. Any modification to proxy rules, database connection strings, or user permissions required editing the file and restarting the entire gateway—an operation that necessarily resulted in dropped connections and service interruption.

SQLite fundamentally resolves this issue by enabling real-time configuration queries. When an administrator updates a setting through the WebUI, the change is committed to the database immediately and atomically. Subsequent requests automatically retrieve the updated configuration without requiring any application restart or service disruption. This capability is essential for maintaining high availability in production environments where configuration changes are routine operational tasks.

13.3.2 Efficiency & Memory Management

The YAML approach required loading the entire configuration dataset into memory at startup, creating a cached representation of all configuration data. This architecture introduced several problems, including memory inefficiency and the risk of state drift—a condition where in-memory data diverges from the on-disk representation if files are modified externally or by concurrent processes.

SQLite’s relational model eliminates these issues through structured, on-demand data access. Rather than maintaining complex nested maps and arrays in memory, the gateway queries specific configuration elements exactly when needed. This approach significantly reduces the application’s memory footprint, particularly as the configuration dataset grows to encompass hundreds of database connections, thousands of user accounts, and complex policy rules.

Furthermore, SQLite’s query optimizer ensures that data retrieval operations are efficient even as the dataset scales, something that would require substantial custom indexing logic if implemented with in-memory data structures.

13.3.3 API Integration & WebUI Compatibility

Modern administrative interfaces require comprehensive Create, Read, Update, and Delete (CRUD) operations on configuration data. Implementing these operations safely with YAML files—particularly handling concurrent modifications, maintaining data integrity, and providing transactional semantics—is complex and error-prone.

SQLite provides a standardized SQL interface that dramatically simplifies API development. The Go backend can leverage SQL’s powerful query capabilities to implement sophisticated operations with minimal code. For example, filtering connection strings by environment, paginating audit logs, or searching for users by role becomes trivial with SQL queries:

```
SELECT * FROM connections
WHERE environment='production'
ORDER BY name LIMIT 10
```

This SQL-based approach integrates seamlessly with the React/Node.js WebUI, which can issue standard REST API calls that translate directly to SQL queries. The result is a clean, maintainable codebase with clear separation between the presentation layer (React), business logic (Node.js API), and data persistence (SQLite).

13.3.4 Enhanced Security Through Data-at-Rest Encryption

Security considerations provided the final compelling argument for SQLite adoption. YAML files are inherently plain text, meaning that any attacker who gains read access to the server's filesystem can immediately view all sensitive configuration data, including database credentials, encryption keys, and access tokens.

To address this vulnerability, the gateway implements a data-at-rest encryption strategy using AES (Advanced Encryption Standard). Before persisting sensitive data to the SQLite database, the Go application encrypts it using AES-256 in GCM mode. Database connection strings, API keys, and other sensitive fields are stored only in their encrypted form.

The practical impact of this approach is substantial: even if an attacker obtains a copy of the SQLite database file through a filesystem breach or backup compromise, the sensitive columns contain only cryptographically secure ciphertext. Only the running Go application, which holds the master encryption key in memory (and never persists it to disk), can decrypt and utilize the actual configuration data. This defense-in-depth strategy aligns with Zero Trust principles by assuming that filesystem access controls may be breached and providing an additional layer of protection.

13.4 Why mTLS (and why TCP is temporary)

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 Scrum Framework

To manage the complexity of the project and ensure continuous development, the team adopted the Agile Scrum methodology. We structured the development lifecycle into one-week sprints, enabling rapid iteration and frequent feedback cycles. This short sprint duration allowed us to demonstrate tangible progress weekly and incorporate supervisor feedback more frequently, ensuring alignment with project objectives throughout the development process.

15.2 Meeting Structure

Our workflow is organized around three distinct meeting types: the Weekly Kick-off, Daily Stand-ups, and the Sprint Review with stakeholders. This structured rhythm of recurring meetings ensures that team milestones and progress are consistently monitored and synchronized. Collectively, these meetings serve to define, review, and align all weekly tasks in accordance with agile best practices.

15.2.1 Weekly Kick-off Meeting

This meeting is held at the start of every sprint to align the team for the upcoming week. It encompasses three key components:

- **Retrospective:** We briefly analyze the previous sprint, discussing what went well and identifying bottlenecks (e.g., merge conflicts or unclear requirements). This reflection helps the team avoid repeating past mistakes and continuously improve our process.
- **Backlog Refinement:** We review upcoming tasks to ensure they are clearly defined and that all technical requirements are understood before assignment. This step reduces ambiguity and sets clear expectations.
- **Sprint Planning:** We select specific tasks from the refined backlog to be completed in the current week. Tasks are assigned to team members based on priority,

complexity, and estimated effort.

15.2.2 Daily Stand-up Meeting

This brief synchronization meeting is held daily to maintain continuous team alignment and identify blockers early.

- **Duration:** Limited to approximately 15 minutes total, with each member speaking for no more than 2 minutes. This time constraint ensures the meeting remains focused and efficient.
- **Format:** Each team member addresses two specific points: what they accomplished yesterday and what they plan to work on today. Any blockers or dependencies are also raised.
- **Objective:** This practice ensures that no team member works in isolation or is blocked by dependencies without the rest of the team being aware. It promotes transparency and enables rapid problem-solving.

15.2.3 Sprint Review (Weekly Supervisor Meeting)

At the conclusion of each sprint, a formal review is conducted with our project supervisor and mentors to validate progress and gather feedback.

- **Demonstration:** The team presents the functional features completed during the sprint, showcasing working software rather than just documentation or plans.
- **Validation:** Our supervisors provide immediate feedback on the implementation. This feedback is either approved for integration or converted into new change requests to be prioritized in the next sprint's backlog.

15.3 Collaboration Tools

To ensure accessibility and efficient collaboration across all phases of development, we employ an integrated tool stack that supports both synchronous and asynchronous communication:

Central Knowledge Base (Notion): Serves as the single source of truth for the project wiki. It is used for sprint planning, task and goal tracking, documenting new code features, and archiving meeting notes and recordings. All team members have real-time access to project documentation.

Version Control & Technical Documentation (GitHub): The primary repository for source code, version control, and code reviews. GitHub also hosts technical documentation, including the Proxy Installation Guide and API references.

Synchronous Communication: Discord, Microsoft Teams, and Google Meet are used for scheduled meetings and real-time voice/video collaboration. These platforms facilitate immediate discussion and decision-making.

Asynchronous Communication: WhatsApp is used for quick, urgent team notifications and simple coordination when immediate responses are needed outside of scheduled meetings.

Auxiliary Tools: We leverage AI-powered tools for generating meeting summaries and conclusions, creating immediate and searchable records of discussions. Excalidraw is used for creating collaborative diagrams, flowcharts, and architectural drawings during design discussions.

15.4 Documentation & Observability

We prioritize high observability by documenting all progress, regardless of scale. This comprehensive documentation approach ensures transparency and facilitates knowledge transfer within the team.

While day-to-day execution is tracked in Notion, high-level progress reporting is documented in a formal **Supervisor Meeting Log** maintained as a Word document. This log serves as an official record and tracks:

- **Attendance & Date:** A record of meeting participants and timestamps.
- **Retrospective:** Feedback on completed tasks, including what was delivered and any deviations from the plan.
- **Forward Planning:** Clearly defined goals and expected outcomes for the subsequent meeting, establishing accountability and measurable targets.

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