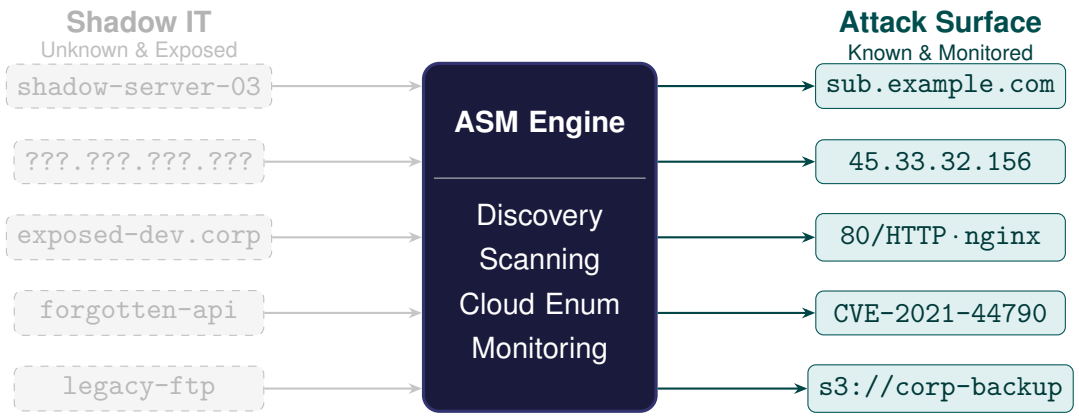
CHALMERS
UNIVERSITY OF TECHNOLOGY




# Automated External Attack Surface Mapping

## Developing a High-Performance Reconnaissance Engine using Concurrent Systems in Go

Degree project report in Computer Engineering (TIDAL)

Arvin Allahbakhsh

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Degree project report 2026

# Automated External Attack Surface Mapping

Developing a High-Performance Reconnaissance Engine using Concurrent Systems in Go

Arvin Allahbakhsh


CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Automated External Attack Surface Mapping
Developing a High-Performance Reconnaissance Engine using Concurrent Systems in Go
Arvin Allahbakhsh



Supervisor: Mahsima Jalooli, Nordic Defender
Supervisor: Sakib Sistek, Chalmers University of Technology, Department of Computer Science and Engineering
Examiner: John J. Camilleri, Chalmers University of Technology, Department of Computer Science and Engineering

Degree project report 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Conceptual before/after diagram of external attack surface management. Unknown and exposed assets (left) are discovered, scanned, and classified by the ASM engine (centre) into a monitored and tracked inventory (right).

Typeset in LaTeX
Gothenburg, Sweden 2026

Automated External Attack Surface Mapping
Developing a High-Performance Reconnaissance Engine using Concurrent Systems in Go

Arvin Allahbakhsh
Department of Computer Science and Engineering
Chalmers University of Technology

# Abstract

Organisations that deploy software rapidly often lose track of what they have exposed to the internet. Forgotten staging servers, misconfigured cloud storage buckets, and outdated services running on undocumented ports accumulate faster than security teams can audit them manually. Attack Surface Management (ASM) addresses this by continuously discovering and monitoring an organisation's externally reachable assets from the perspective of an attacker rather than an administrator.

This project designs and implements a high-performance ASM-engine in Go. The engine operates as a three-phase pipeline. Phase 1, passive Open Source Intelligence (OSINT) discovery retrieves subdomains and IP addresses from public sources without sending any probes to the target. Phase 2, active TCP scanning connects to each discovered asset, extracts service banners, matches versions against an embedded CVE database to surface known weaknesses, and computes a differential analysis against the previous scan. Phase 3, cloud storage probing checks for associated storage buckets across AWS, Azure, and Google Cloud Platform. Within each phase, operations run concurrently, results are persisted to PostgreSQL, and the differential analysis component reports only what has changed since the last scan rather than repeating the full inventory every time. A web-UI and a command-line interface give operators two ways to interact with the engine.

The engine was evaluated against three environments. An isolated Docker Compose sandbox used to verify differential change detection under controlled conditions, a live scan of `scanme.nmap.org`, and a scan of `chalmers.se` to demonstrate the engine at organisational scale. All local scans completed in approximately 130 milliseconds. The entire engine is implemented using Go's standard library and a single external PostgreSQL driver. No API keys or third-party scanning infrastructure are required.

Keywords: attack surface management, OSINT, Go, concurrent scanning, differential analysis, vulnerability mapping, cloud storage probing.

# Acknowledgements

Several people made this work possible, and I am grateful to each of them.
My mother and father have supported me with patience and encouragement throughout my entire education. I could not have done this without knowing they were always there for me.

My wife, Soha, deserves special recognition. She has been my steadiest support throughout this entire degree, patient when progress was slow, motivating when I lost momentum, and always honest. When things were difficult, she reminded me that setbacks are not permanent. I am more grateful than I can express.

At Nordic Defender, I want to thank Vincent Heidari, Chief Executive Officer, for giving me the opportunity to carry out this project in collaboration with the company and for supporting my idea that became the foundation of this thesis. I also want to thank my supervisor, Mahsima Jalooli, for her guidance and support throughout the development and writing process.

Finally, I want to thank the Department of Computer Science and Engineering at Chalmers University of Technology for an outstanding education that gave me the foundation to take on a project of this scope.

Arvin Allahbakhsh, Gothenburg, May 2026

# List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

| | |
|---|---|
| ACID | Atomicity, Consistency, Isolation, Durability |
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| ASM | Attack Surface Management |
| AWS | Amazon Web Services |
| CA | Certificate Authority |
| CLI | Command Line Interface |
| CSP | Content Security Policy |
| CT | Certificate Transparency |
| CVE | Common Vulnerabilities and Exposures |
| DNS | Domain Name System |
| DevOps | Development and Operations |
| FTP | File Transfer Protocol |
| HTTP | HyperText Transfer Protocol |
| IDS | Intrusion Detection System |
| IMAP | Internet Message Access Protocol |
| IT | Information Technology |
| MX | Mail Exchanger |
| OOD | Object-Oriented Design |
| OOP | Object-Oriented Programming |
| OSINT | Open Source Intelligence |
| POP3 | Post Office Protocol version 3 |
| PTR | Pointer Record (a DNS record type that maps an IP address back to a hostname) |
| RDBMS | Relational Database Management System |
| SaaS | Software as a Service |
| SMTP | Simple Mail Transfer Protocol |
| SNMP | Simple Network Management Protocol |
| SOLID | Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion (software design principles) |
| SPF | Sender Policy Framework |
| SSH | Secure Shell |

| | |
|---|---|
| SSE | Server-Sent Events |
| SSRF | Server-Side Request Forgery |
| TCP | Transmission Control Protocol |
| TLS | Transport Layer Security |
| UDP | User Datagram Protocol |
| UI | User Interface |
| UPSERT | Update or Insert (a database operation that inserts a new row or updates an existing one) |
| URL | Uniform Resource Locator |

# Contents

# List of Figures

# List of Tables

# 1
# Introduction

This chapter provides the foundation for the thesis. It introduces the concept of Attack Surface Management (ASM), describes the challenges faced by modern organizations regarding their digital footprint, and outlines the purpose and scope of the developed ASM-engine.

## 1.1 Background

In an era of rapid digital transformation, an organization's security is no longer confined to its internal network boundaries. The proliferation of cloud services, remote work, and decentralized development has led to an explosion of internet-facing assets. The sum total of these assets, including web servers, APIs, cloud storage buckets, and DNS records, constitutes an organization's *external attack surface*.

Attack Surface Management is a proactive security discipline focused on discovering, analyzing, and monitoring these assets from an "outside-in" perspective, exactly how a potential attacker would view the organization. The core objective of ASM is to identify "Shadow IT". The meaning of "Shadow IT" is resources that have been deployed outside the oversight of central IT or security departments and may, therefore, lack proper security controls [33].

## 1.2 Problem Description

The speed of modern software development, often accelerated by AI-assisted tools and the rise of "vibe coding" where functional code is generated rapidly with a focus on immediate results, frequently outpaces security governance. This leads to a growing gap known as *security debt*.

Common symptoms of this problem include:

- Forgotten staging or development servers that remain exposed to the internet.
- Misconfigured cloud storage buckets, for example, AWS S3 or Azure Blobs, containing sensitive data.
- Outdated software services running on unknown ports.

Organizations often lack a comprehensive, automated, and real-time map of their external presence, leaving them vulnerable to attacks on assets they do not even know they possess.

## 1.3 Purpose

The purpose of this degree project is to design and implement a high-performance automated Attack Surface Management engine in Go. The engine aims to transform the organizational posture from "Unknown and Exposed" to "Known and Monitored".

By leveraging concurrent programming in Go, the project seeks to demonstrate how a modern, lightweight engine can efficiently discover subdomains via Open Source Intelligence (OSINT), map active infrastructure through high-speed port scanning, and identify high-risk exposures like public cloud buckets and vulnerable service versions. A central constraint was that the engine must remain fast enough to be used operationally. Scanning an organisation's full external surface should complete in roughly the time it would take a sequential scanner to probe a single host on a single port.

The complete source code is publicly available at:
`https://github.com/arvin-sudo/asm-engine`.

## 1.4 Goals

To achieve the purpose of this project, the following technical and academic goals have been defined. Performance acts as a constraint on all of them. Modularity, correctness, and usability are only valuable if the engine is fast enough to be practical.

1. **Architectural Design:** Develop a modular, interface-based pipeline in Go that adheres to Clean Architecture and SOLID principles, ensuring extensibility for future security modules.
2. **Performance Optimization:** Implement a concurrent port scanning engine using Go's worker pool pattern to significantly reduce execution time compared to sequential methods.
3. **Data Persistence and Analysis:** Design a relational database schema to store discovered assets and implement differential analysis logic to track changes over time, for example, new subdomains or opened ports.
4. **User Accessibility:** Create a built-in web-based user interface that visualizes scan results in real-time, making the tool accessible to developers without extensive command-line experience.
5. **Ethical Verification:** Construct a fully containerized sandbox environment using Docker to safely verify the engine's active scanning capabilities without impacting production networks.

## 1.5 Limitations and Delimitations

To ensure the project remains feasible within the given timeframe, the following limitations and delimitations have been established:

- **External Focus Only:** The engine only maps the *external* attack surface. Internal network discovery, such as scanning local intranets or private subnets, is outside the scope.
- **No Exploitation:** While the engine identifies potential vulnerabilities (Vulnerability Mapping), it does not attempt to exploit them. Automated penetration testing capabilities are excluded.
- **No Remediation:** The engine is designed for discovery and mapping. It does not provide automated fixes, patch management, or code-level remediation for the discovered vulnerabilities.
- **OSINT Source Selection:** Discovery is limited to a selected set of high-impact OSINT sources, for example, Certificate Transparency logs and specific DNS records, rather than an exhaustive search of all available public records.
- **Protocol Support:** The current implementation focuses primarily on the TCP protocol for scanning and fingerprinting, extensive UDP-based discovery is not prioritized.

## 1.6 Target Audience

This report is intended for security engineers, DevOps professionals, and software developers interested in automated reconnaissance, as well as researchers studying the application of Go's concurrency primitives in cybersecurity tooling.

# 2

# Theory and Technical Background

This chapter provides the theoretical framework necessary to understand the implementation and evaluation of the ASM-engine. It covers the core concepts of Attack Surface Management, the passive reconnaissance techniques used for asset discovery, the specific advantages of the Go programming language for network tooling, the methods used to fingerprint exposed services and map known vulnerabilities, and the architectural principles that guided the system's design.

## 2.1 Attack Surface Management (ASM)

Attack Surface Management is a specialized field of cybersecurity that focuses on the continuous discovery, inventory, and vulnerability assessment of an organisation's digital assets. Unlike traditional vulnerability management, which often focuses on known internal assets, ASM adopts an outside-in perspective [28], discovering assets as an attacker would rather than starting from a known internal inventory.

The ASM lifecycle typically consists of four main phases:

1. **Discovery:** Identifying unknown assets (subdomains, IPs, cloud buckets).
2. **Analysis:** Fingerprinting services and identifying owners.
3. **Prioritisation:** Assessing risk based on vulnerability data and asset criticality.
4. **Remediation:** Closing security gaps and monitoring for changes.

The developed engine specifically automates the first two phases while providing the data necessary for prioritisation. Figure 2.1 illustrates this continuous cycle.

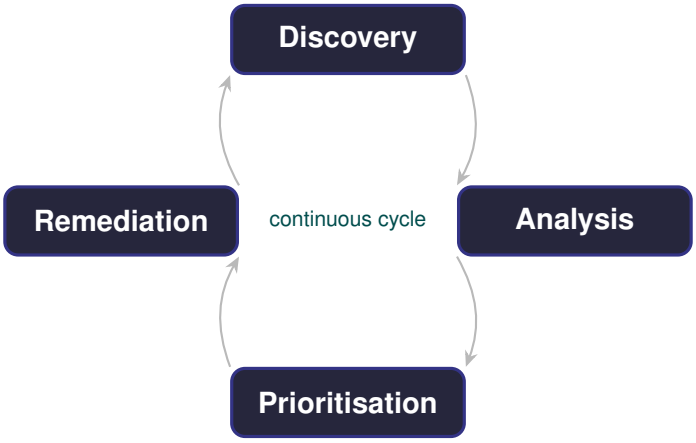



**Figure 2.1:** The four-phase ASM lifecycle. This engine automates the Discovery and Analysis phases; Prioritisation is enabled by the CVE and vulnerability data it surfaces.

## 2.2 Open Source Intelligence (OSINT)

Open Source Intelligence (OSINT) refers to the collection and analysis of data gathered from open, public sources to produce actionable intelligence [1]. In the context of cybersecurity and Attack Surface Management, OSINT is used to perform reconnaissance without directly interacting with the target's infrastructure. This passive approach is crucial as it allows for the discovery of assets while remaining undetected by traditional internal security measures.

The ASM-engine developed in this project draws on four complementary OSINT techniques to build its initial picture of a target organisation's footprint. Certificate transparency logs, passive DNS aggregation, historical URL archives, and DNS record intelligence. Each technique surfaces a different class of assets, and together they provide significantly broader coverage than any single source alone.

## 2.3 Certificate Transparency (CT)

Certificate Transparency is an open framework designed to monitor and audit TLS certificates [2]. Every time a Certificate Authority (CA) issues a certificate for a domain, it must be logged in an append-only, public CT log.

For an ASM-engine, CT logs represent a valuable source of information. By querying these logs, for example, via the crt.sh API [31], the engine can discover subdomains that may not be linked elsewhere on the public web, such as forgotten staging environments or internal-only APIs that were nonetheless issued public certificates.

## 2.4 Passive DNS and Historical Reconnaissance

While Certificate Transparency reveals assets that were formally issued a TLS certificate, many subdomains surface through other channels [3]. Two complementary techniques are used to capture these.

### 2.4.1 Passive DNS Aggregation

Passive DNS is the practice of recording and indexing DNS query–response pairs observed across distributed networks of resolvers [3]. Every time a DNS resolver answers a query, for example, `www.chalmers.se`, that observation can be logged. Over time, passive DNS databases accumulate a historical record of which hostnames have existed for a given domain, even if those names have since been removed from active DNS or were never issued a certificate. The engine queries the HackerTarget passive DNS API [32] to retrieve these historical subdomain observations, surfacing forgotten assets that CT logs alone would not reveal.

### 2.4.2 Historical URL Reconnaissance

The Wayback Machine, operated by the Internet Archive, has crawled and archived publicly accessible web pages since the late 1990s [4]. Its CDX API allows querying the full archive for every URL ever captured under a given domain. This reveals hostnames embedded in historical page links, JavaScript files, and stylesheets: resources that may have been removed from the live site years ago but whose underlying infrastructure may still be running and exposed. Because the Wayback Machine indexes the web as it appeared over time, it captures short-lived environments such as release-specific staging servers that other sources would miss entirely.

### 2.4.3 Reverse DNS Enrichment

Once a set of IP addresses has been discovered through forward DNS resolution of the found subdomains, the engine performs reverse DNS lookups (PTR queries) on those addresses. Reverse DNS maps an IP address back to a hostname. This step surfaces additional hostnames that were never published in certificate logs or passive DNS databases but are nonetheless reachable on the same IP space, a common pattern in organisations that manage multiple services on shared infrastructure.

## 2.5 DNS Record Intelligence

Beyond enumerating subdomains, DNS records contain a layer of organisational intelligence that is valuable for understanding third-party dependencies [25]. Two record types are particularly informative.

SPF records, stored as DNS TXT entries, list the external services authorised to send email on behalf of a domain [5]. An entry such as `include:_spf.google.com` indicates that Google Workspace handles the organisation's email, a similar include

for `spf.protection.outlook.com` points to Microsoft 365.

MX records, which direct incoming email to the correct mail servers, reveal the email provider through the hostname of the designated mail exchanger [6].

By parsing these records, the engine builds a map of the organisation's third-party SaaS dependencies without any active probing. This is a form of shadow IT discovery. It's essentially identifying external services that hold or process the organisation's data, each of which represents an indirect component of the attack surface that should be understood and monitored.

## 2.6 The Go Programming Language

The choice of programming language is critical for high-performance network reconnaissance. Go was selected for this project due to its unique approach to concurrency, its efficient networking standard library, how fluid, light and simplified it is, and its first-class support for cancellation propagation [7].

### 2.6.1 Concurrency Model: Goroutines and Channels

One of Go's primary strengths is its concurrency model, based on the *Communicating Sequential Processes* formalism.

- **Goroutines:** These are lightweight threads managed by the Go runtime. Unlike OS threads, which require megabytes of stack space, a goroutine starts with only a few kilobytes, allowing the ASM-engine to spawn hundreds of concurrent probes, for example, for port scanning, without exhausting system memory [7, 18].
- **Channels:** These are typed conduits through which goroutines communicate. By using channels, the ASM-engine avoids common pitfalls of shared-memory concurrency, such as race conditions, adhering to the Go mantra: "Do not communicate by sharing memory; instead, share memory by communicating." [7].

### 2.6.2 Cancellation and Context Propagation

A recurring challenge in concurrent network programs is ensuring that all in-flight work stops cleanly when the user cancels an operation. Go's standard library addresses this through the `context` package, which provides a lightweight mechanism for carrying cancellation signals across API boundaries [8].

In the ASM-engine, a cancellation context is attached to every scan. When the user closes the browser tab or presses Ctrl+C in the terminal, the server detects the disconnect and cancels the context. This signal propagates automatically through the entire pipeline: HTTP requests to OSINT sources are aborted, port scanner workers stop opening new connections, and cloud bucket probes halt after their current request completes. No manual cleanup code is required at each layer. This approach is idiomatic in Go and is considered a prerequisite for building production-quality

concurrent systems. A program that ignores context cancellation will continue consuming network bandwidth and system resources long after the user has moved on.

## 2.7 Network Fingerprinting

Network fingerprinting is the process of interrogating open network ports to determine the software, version, and technology stack running on them. Where passive OSINT discovers that an asset exists, fingerprinting determines what that asset is, and therefore what vulnerabilities it may carry [13].

Active reconnaissance involves interacting with open ports to identify the software and services running behind them. The engine performs this in two complementary stages.

### 2.7.1 Banner Grabbing

The first stage focuses on the server software itself.

- **TCP Three-Way Handshake:** The engine uses a standard TCP dial to confirm whether a port is open [26].
- **Banner Grabbing:** Once a connection is established, the engine reads the initial greeting sent by the server or sends a minimal HTTP request to inspect the response. Different protocols reveal themselves in distinct ways. An SSH server announces its software version in its first message [20], an HTTP server discloses the underlying web server software in the `Server` response header, and mail servers (SMTP, IMAP, POP3) and FTP typically identify their software in a short greeting. This information is sufficient to determine the service type and, in many cases, the exact version number, which is the input needed for vulnerability matching [14].

### 2.7.2 Technology Stack Fingerprinting

While banner grabbing reveals the underlying server software, it says nothing about the application running on top of it. A server might report nginx, but the web application could be built with WordPress, React, or Laravel, each carrying its own security implications. Technology stack fingerprinting addresses this by analysing the HTTP response at a higher level.

The ASM-engine examines three signals. Response headers such as `X-Powered-By` can disclose the runtime environment or framework. Cookie names reveal the session management library in use, for example, `PHPSESSID` indicates PHP and `JSESSIONID` indicates a Java application server, and patterns in the HTML source identify front-end libraries and content management systems. Together, these signals allow the engine to detect a wide range of technologies. From server-side frameworks, like Django and Laravel, to front-end libraries, like React, Vue, and Angular, without requiring any special access or authentication.

This information is particularly valuable for prioritisation. A known-vulnerable version of a widely deployed content management system exposed on a public-facing asset carries a very different risk profile than a custom static site served by the same web server.

## 2.8 Vulnerability Mapping

Vulnerability mapping is the process of comparing identified service versions against a catalogue of publicly disclosed security flaws, known as CVEs, Common Vulnerabilities and Exposures [11], to determine whether a discovered asset is running software with known weaknesses. In an ASM context this transforms raw service information into prioritised risk findings without requiring any exploitation attempt.

Once a service has been identified and its version extracted through banner grabbing, the ASM-engine compares it against a curated database of known vulnerabilities. Each entry in this database defines a CVE identifier, a severity level of Critical, High, Medium, or Low [16], and the range of software versions known to be affected. This allows the ASM-engine to automatically flag, for example, that a discovered SSH service running a specific version is affected by a known remote code execution vulnerability.

Rather than querying an external vulnerability feed at runtime, the database is compiled directly into the engine binary. This design decision eliminates the need for network access to a third-party service during a scan, ensures that results are fully reproducible, and allows the vulnerability data to be version-controlled alongside the code. The trade-off is that the ruleset must be updated manually rather than receiving automatic updates, an acceptable compromise for a research prototype.

Vulnerability findings are not persisted in the database. The persistent record tracks which services and ports were observed and when. The risk assessment is always derived fresh at scan time by applying the current ruleset to the observed service versions. This separation keeps the storage schema simple and ensures that re-running a scan against historical data always reflects the most up-to-date vulnerability knowledge.

## 2.9 Cloud Storage Architecture

Cloud object storage services, such as Amazon S3, Azure Blob Storage, and Google Cloud Platform, provide HTTP-accessible file storage where access is governed solely by bucket-level permissions [17]. Unlike a traditional file server, a storage bucket has a predictable public URL, meaning a misconfigured bucket is reachable by anyone on the internet who knows or guesses its name.

Modern web applications often offload file storage to cloud providers. The ASM-engine probes for publicly accessible storage resources across three major platforms:

Amazon Web Services, Microsoft Azure, and Google Cloud Platform. Each provider uses a predictable URL structure based on the bucket or container name. For example, `https:/[name].s3.amazonaws.com` for AWS S3, which allows the engine to construct candidate URLs from discovered domain names and test them systematically.

Security risks arise when these resources are misconfigured with public *List* or *Read* permissions, allowing anyone to browse or download the stored contents. By probing each candidate URL and interpreting the HTTP response code, the engine can distinguish genuinely exposed storage from private resources: a 200 response indicates the resource is publicly accessible, while a 403 response confirms it exists but is access-controlled. A 403 still reveals the existence of the bucket, which is itself a useful piece of intelligence.

## 2.10 Data Persistence and Relational Modeling

To perform differential analysis and track changes over time, an Attack Surface Management engine requires a persistent storage layer. A relational database management system (RDBMS) is particularly suited for this task due to its ability to handle complex relationships between entities.

### 2.10.1 Relational Mapping of Infrastructure

Mapping an organisation's digital footprint involves structured data with clear hierarchies. A single *Asset* (a domain) may be reachable at multiple IP addresses, may expose multiple *Ports*, and each port may host a *Service* with a specific name and version. By utilising an RDBMS like PostgreSQL, the engine can enforce referential integrity, ensuring that service records are always correctly linked to their parent port, and that port records are always linked to their parent asset. Discovered cloud storage resources are tracked separately, linked to the domain under which they were found.

### 2.10.2 State Tracking and Differential Analysis

The core value of persistence in ASM is the ability to compare the current state of the attack surface with previous scan results. This requires the database to support efficient atomic operations, such as "UPSERTs" (Update or Insert), which allow the engine to record a finding without checking whether it already exists (the database handles the merge). This allows the engine to maintain a continuous timeline of an asset's exposure, marking properties such as `first_seen` and `last_seen` without duplicating data. When the current scan is compared against the stored history, the engine can surface meaningful changes: a port that was not open in the previous scan (*PORT OPENED*), a service whose version has changed (*VERSION CHANGE*), or an asset that appears for the first time (*NEW ASSET*).

### 2.10.3 Concurrency and Data Integrity

In a high-performance system using concurrent execution, data integrity is a primary concern. Utilising a database that adheres to ACID (Atomicity, Consistency, Isolation, Durability) properties ensures that multiple concurrent scanning processes (goroutines) can write findings to the storage layer simultaneously without risking data corruption or partial writes [22].

## 2.11 Real-time Web Communication

To provide users with immediate feedback to the user interface during a scan, the engine utilises a real-time data streaming approach.

### 2.11.1 Server-Sent Events (SSE)

Server-Sent Events (SSE) is a standard allowing servers to push real-time updates to web pages over a single, long-lived HTTP connection [9]. Unlike WebSockets, which provide bidirectional communication, SSE is unidirectional (server-to-client). This is ideal for an ASM UI where the engine needs to stream continuous discovery results and status updates to the user interface as they occur, minimising latency and providing immediate feedback to the security analyst as the scan progresses.

## 2.12 Software Architecture Principles

Software architecture principles are design guidelines that govern how a system's components are structured and connected, independent of any specific technology choice. Applied consistently, they keep complexity manageable as a system grows by making each component's responsibilities clear and its dependencies explicit.
To ensure maintainability and testability, the ASM-engine follows two primary design philosophies:

- **Clean Architecture:** The engine is structured in layers, ensuring that the core business logic (discovery and scanning) is independent of external factors like the CLI or the database [12].
- **SOLID Principles:** Specifically the *Interface Segregation Principle* and *Dependency Inversion*. By depending on interfaces rather than concrete implementations, the engine allows for easy mocking in unit tests and modular expansion [12].

### 2.12.1 Coupling and Cohesion

A primary goal of the system architecture is to achieve *Low Coupling* and *High Cohesion*:

- **Low Coupling:** By ensuring that individual modules, for example, the port scanner and the database store, do not depend on each other's internal logic, the system becomes more flexible. One module can be replaced or updated without affecting others.

- **High Cohesion:** Each package in the ASM-engine is designed to perform one specific task, for example, finding subdomains. This reduces redundancy and makes the codebase easier to maintain and test.

### 2.12.2 Composition over Inheritance

While traditional Object-Oriented Programming (OOP) often relies on complex class hierarchies and inheritance, Go promotes a simpler model based on *composition* and *interfaces*. By moving away from rigid Object-Oriented Design (OOD) patterns and instead using small, focused interfaces, the ASM-engine avoids the "fragile base class" problem and redundancy. This allows the developer to compose complex behaviour by combining simple, independent components.

# 3
# Methods

This chapter describes the strategy and methodology used to design, implement, and verify the ASM-engine. It outlines the project management approach, the iterative development process, and how the system was safely tested in a controlled environment to meet both ethical and academic standards.

## 3.1 Project Management and Work Process

The project followed a *Waterfall-inspired model* rather than an agile framework like Scrum. Given the technical nature of the ASM-engine and the clearly defined sequential phases (Reconnaissance, Mapping, Probing, and Persistence), a linear progression allowed for a deep architectural focus in each stage.

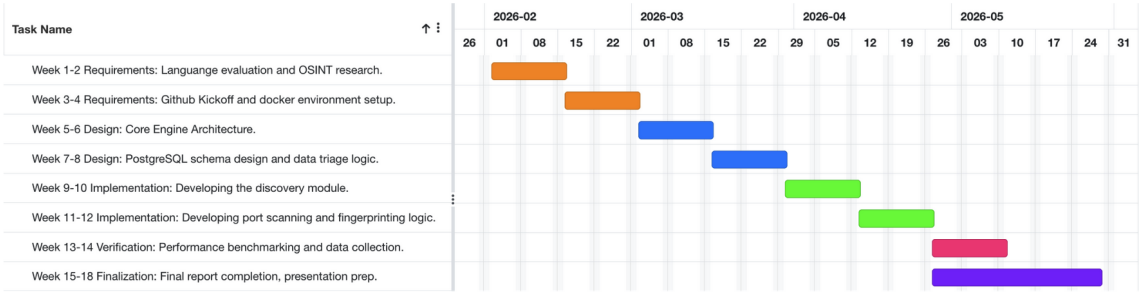



**Figure 3.1:** Gantt-Chart, the project timeline. The phases were carried out sequentially, with each phase building on the confirmed findings of the previous one.

### 3.1.1 Collaboration and Remote Work

The project was carried out in a fully remote environment. Coordination and technical discussions with the supervisor at Nordic Defender were conducted through digital communication platforms. This remote setup required a high degree of self-management and structured documentation to ensure alignment between the academic requirements of Chalmers and the practical expectations of the company.

## 3.2 Development Process

The project followed an iterative development model characterised by a "refactor-first" philosophy. Refactoring is the process of cleaning and improving existing code

without changing its external behaviour [23]. This ensured the system remained well-structured as it grew in complexity. Rather than a purely linear path, each new feature was preceded by a review and improvement pass of the existing codebase.

### 3.2.1 Iterative Refactoring and Hardening

The development was structured as a cycle of feature implementation followed by dedicated hardening passes. A hardening pass is a focused review cycle that targets a specific category of concern, such as correctness, code duplication, security, or resource management. Four formal hardening passes were carried out during the final phase of the project, each addressing a distinct set of issues:

1. **Correctness:** Fixing behavioural bugs such as a mismatched event type name in the web client and unsafe JSON construction in the server's error handler.
2. **Duplication and structure:** Extracting shared logic, such as domain normalisation and the subdomain scope filter, that had been independently duplicated across multiple packages into single, reusable components.
3. **Security:** Adding input validation to the web server, for example, clamping the number of parallel workers to a safe maximum, sanitising error messages before sending them to the client to prevent credential leakage, and adding standard HTTP security headers.
4. **Cancellation propagation:** Ensuring that every in-flight network operation, including OSINT HTTP requests, TCP dial attempts, and cloud bucket probes, respected Go's context cancellation mechanism, so that all work stops cleanly when the user closes the browser or presses Ctrl+C.

This approach ensured that the system maintained a high degree of testability and adhered to SOLID principles throughout the project. It also enabled the incremental identification and resolution of quality issues, such as the absence of deduplication logic across the three parallel OSINT sources, which was added as a shared utility rather than duplicated inline code.

## 3.3 Security-by-Design

Security considerations were treated as a first-class concern throughout development, not deferred to a final review [15]. This reflects the threat model of the tool itself. A network scanner issues requests on behalf of the operator, and if its inputs are not properly validated it can become an attack vector rather than a defence. Without explicit guards, the tool could be used as a proxy to reach internal infrastructure, exhaust system resources, or leak sensitive credentials through its own error output. Four categories of risk were identified early in the design process: scope enforcement, resource exhaustion, credential leakage, and concurrency limits. A mitigation was designed for each before any external interface was exposed. The concrete mechanisms implementing these protections are described in Section 4.5.

## 3.4 System Architecture and Pipeline Design

A core method for achieving the project's performance and maintainability goals was the design of a decoupled, event-driven pipeline. This architecture functions similarly to an assembly line. Data flows through specialised stages, and each stage emits structured events as soon as a finding is produced rather than waiting for the entire scan to complete.

### 3.4.1 Phased Discovery and Analysis

The engine's pipeline was divided into three sequential phases to maintain a clear separation of concerns. Passive reconnaissance produces a confirmed list of live assets, active scanning probes only assets that are known to exist, and cloud probing operates on derived name patterns rather than live hosts. This phased progression keeps each stage's responsibility narrow and allows findings to be persisted incrementally, so differential analysis always has an up-to-date baseline. The specific sub-phases and their implementation are described in detail in Chapter 4 Implementation.

### 3.4.2 Algorithmic Design and Efficiency

Several implementation choices in the engine were directly motivated by their computational properties. This subsection explains the three decisions where the choice of algorithm or data structure had a meaningful impact on correctness or scalability.

The most significant is the TCP scanning worker pool. A serial implementation would attempt one connection at a time, so the total scan time would grow proportionally to the number of hosts multiplied by the number of ports, with each connection blocking the next until it completed or timed out. The worker pool eliminates this bottleneck: with $k$ workers operating concurrently, the effective scan time is reduced to $O(\text{hosts} \times \text{ports}/k)$, since multiple connections are in flight simultaneously. With the default of 100 workers, an organisation's full attack surface can be scanned in roughly the time it would take to probe a single host serially. The pool size is configurable via the `-workers` flag so operators can balance scan speed against the risk of triggering rate-based defences on the target network.

The second decision concerns subdomain deduplication. Because three independent OSINT sources are queried in parallel, the same subdomain may be returned by more than one source. Checking whether a subdomain has already been seen using a Go `map[string]struct{}` costs $O(1)$ per lookup, keeping the total deduplication work linear in the number of results regardless of how many duplicates appear. A naive approach using a slice would require scanning the entire list for each new entry, producing $O(n^2)$ behaviour as the result set grows.

The third decision is the use of a B-tree index on the ports table in PostgreSQL, keyed on the asset domain, port number, and protocol. The UPSERT that saves

each port finding ("insert if new, update timestamp if known") resolves the existing row in $O(\log n)$ time regardless of how many rows the table accumulates over repeated scans. Without the index, each write would require a full table scan, making persistence performance degrade linearly as scan history grows.

### 3.4.3 Real-time Data Communication

To allow the engine to serve both a command-line interface and a web-UI without duplicating logic, a streaming architecture based on a typed event channel was used. The concrete architecture of this channel is described in Chapter 4, Section 4.1.2.

## 3.5 Testing and Quality Assurance

Testing was treated as a first-class concern throughout development [24], ensuring the engine remained correct and stable as its complexity grew.

### 3.5.1 Unit Testing

Automated unit tests were written for every package. Following Go's convention, tests use a *table-driven* structure. Instead of writing one test function per scenario, a single test function iterates over a table of input and expected output pairs. This approach makes it easy to add new edge cases without duplicating test logic, and produces clear failure messages that immediately identify which case failed. Interfaces and dependency injection were used throughout the codebase to allow real components to be replaced with lightweight in-test fakes, keeping tests fast and deterministic without requiring external services.

### 3.5.2 Integration Testing

Integration tests were conducted against a live PostgreSQL instance to verify that the engine communicates correctly with the database. These 24 tests cover the full range of storage operations. Creating and updating assets, saving ports and services with referential integrity, storing cloud findings, and verifying that timestamps are recorded correctly. They also include adversarial cases such as SQL injection payloads, extremely large inputs, and concurrent writes from 20 simultaneous goroutines. These tests are marked with a build tag so they are excluded from ordinary unit test runs and only executed when a database is available, a deliberate separation that keeps the standard test suite fast.

### 3.5.3 Concurrency Verification

Because the engine performs many operations simultaneously, there is a risk that independent goroutines might access shared data in an unsafe way, leading to nondeterministic behaviour that is difficult to reproduce and debug. Go's built-in race detector (`-race`) was enabled on every test run throughout the project. The race

detector instruments the compiled binary at runtime to detect concurrent memory accesses that are not properly synchronised, flagging them immediately rather than allowing them to produce silent data corruption. Alongside the race detector, `go vet` was run regularly to catch a broader category of static issues, suspicious code patterns that the compiler does not reject but that often indicate bugs. Both tools are documented in the project's `Makefile` as standard targets, making them easy to run consistently.

## 3.6 Ethical Considerations and the Sandbox Environment

A central challenge in developing active scanning tools is the risk of accidentally disturbing real-world systems or triggering security alarms on production networks. To act ethically and legally, the engine was never tested against unauthorised targets on the open internet.

Instead, a *Docker Sandbox Environment* was created, a safe isolated environment running entirely on local machine. The sandbox consists of three containerised services managed by Docker Compose [27]: a PostgreSQL database for persistence, the ASM-engine itself, and a deliberately vulnerable "victim" web server running a known-outdated version of nginx. This setup allowed the engine's full capabilities, including vulnerability detection against real CVEs, to be verified end-to-end without any risk to public infrastructure. It also serves as a reproducible demonstration environment. Anyone who clones the repository can start the sandbox with a single command and observe the engine detecting the known vulnerabilities in the victim service.

When the scan target resolves to a private or loopback address, the engine bypasses all OSINT phases and proceeds directly to active scanning, which makes Docker-based integration testing fast and deterministic. Figure 3.2 shows the structure of the Docker Compose sandbox.

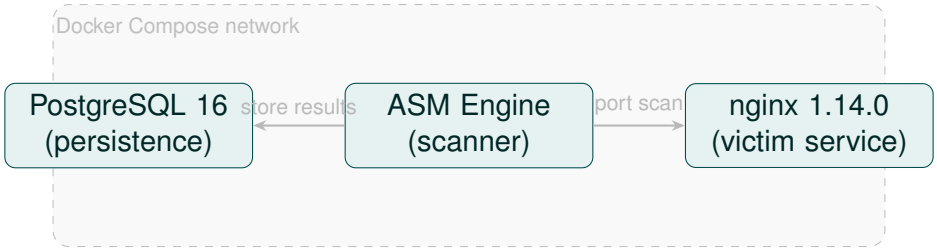



**Figure 3.2:** Docker Compose sandbox used for controlled testing. The ASM engine scans the deliberately outdated nginx victim service and persists findings to PostgreSQL, all within an isolated local network.

# 4

# Implementation

This chapter details the technical construction of the ASM-engine. It describes the software architecture, the internal mechanics of the concurrent scanning pipeline, and the design rationale behind the database, real-time communication, and security layers.

## 4.1 System Architecture and Event Flow

The engine is built using an event-driven, layered architecture inspired by *Clean Architecture* principles. This ensures that the core scanning logic remains decoupled from external concerns such as the database, the web server, or the Command Line Interface (CLI).

### 4.1.1 Interface-Based Design and Decoupling

The system relies heavily on Go interfaces to ensure modularity. By defining a generic `Discoverer` interface for Phase 1, the engine can query multiple OSINT sources, for example, *crt.sh* or *HackerTarget*, without modifying the core pipeline code. To minimize the attack surface and keep the binary compact, the system utilizes the Go standard library, specifically `net/http`, for all network interactions rather than third-party web frameworks.

### 4.1.2 The Typed Event Channel

A central architectural challenge was supporting multiple interface consumers, CLI and Web-UI, simultaneously. This was solved by implementing a `Runner` component that executes the pipeline and emits typed `ScanEvent` objects into a Go channel. This decoupling allows the scanning logic to remain oblivious to the output format. A CLI handler can print events to the terminal, while a Web handler can stream them as real-time updates. Figure 4.1 illustrates this separation.

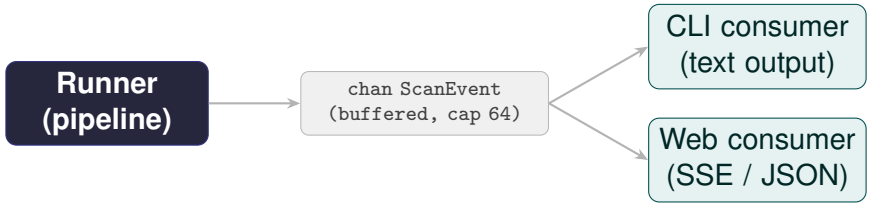



**Figure 4.1:** The typed event channel decouples pipeline execution from output. The `Runner` emits `ScanEvent` values into a buffered channel. Both the CLI and the Web-UI consume the same events independently.

### 4.1.3 Web Dashboard and Real-time Streaming

To provide interactive feedback, the engine features a web-based UI utilizing *Server-Sent Events* (SSE). JSON-encoded events are streamed directly from the Go backend to the browser as they occur, providing real-time updates for port findings and phase completions without the overhead of client-side polling.

The entire frontend is built with vanilla JavaScript and CSS to avoid heavy framework dependencies. Using the `//go:embed` directive, the HTML, CSS, and JS files are embedded directly into the compiled Go binary. This results in the tool being distributed as a single, standalone executable with no external asset requirements. Figures 4.2 and 4.3 show the UI in two states: a scan actively streaming results from the victim-service, and the same scan after it has completed.


**Figure 4.2:** Web UI during an active scan of the victim nginx service. Port findings and phase-completion events stream into the browser in real time via Server-Sent Events.


**Figure 4.3:** Web UI after the scan completes.

## 4.2 The Multi-Phased Scanning Pipeline

The core engine is structured as a high-performance pipeline divided into three distinct phases. Every network operation within the pipeline utilizes `context.Context` for propagation, ensuring that a user-initiated cancellation or timeout gracefully terminates all active goroutines. Figure 4.4 shows the overall structure: Phase 1 performs passive reconnaissance through four sub-phases, Phase 2 performs active scanning and analysis, and Phase 3 probes cloud storage. Results are persisted to PostgreSQL throughout Phases 2 and 3.

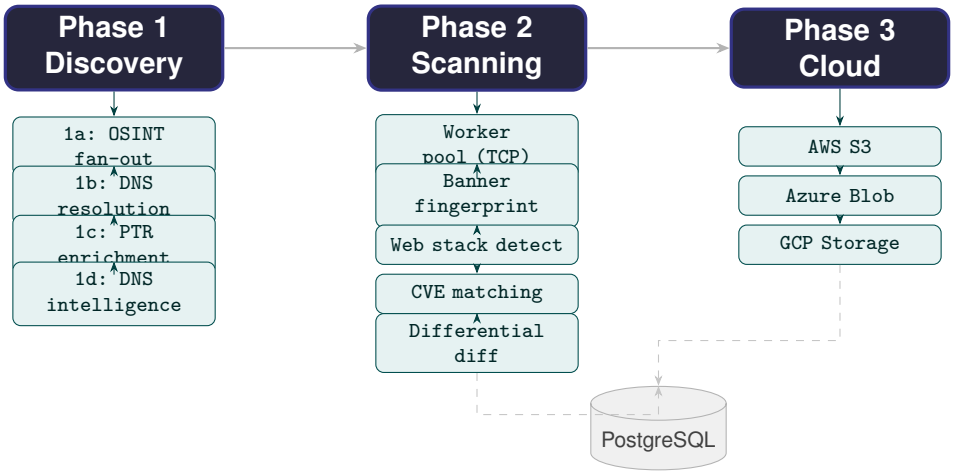



**Figure 4.4:** Three-phase scanning pipeline. Phase 1 performs passive reconnaissance, Phase 2 performs active scanning with fingerprinting and CVE matching, and Phase 3 probes cloud storage. Dashed arrows indicate persistence writes to PostgreSQL.

### 4.2.1 Phase 1: Discovery and DNS Intelligence

Phase 1 maps the target's attack surface through four specialized sub-phases:

- **1a: OSINT Fan-out:** Concurrent queries to CT logs and historical archives. A dedicated `inScope()` filter is applied to CT log results to ensure only subdomains belonging to the target domain are processed, as certificates often cover unrelated third-party domains.
- **1b: DNS Resolution:** Filtering discovered candidates to verify live assets with valid A records.
- **1c: PTR Reverse DNS:** Identifying hidden subdomains by analyzing the reverse pointers of resolved IP addresses.
- **1d: DNS Intelligence:** Analyzing TXT and MX records to reveal third-party service providers, for example, Google Workspace or Microsoft 365.

### 4.2.2 Phase 2: Active Mapping and Fingerprinting

Using a *Worker Pool* pattern, the engine manages concurrent TCP probes to identify open ports. To prevent triggering Intrusion Detection Systems (IDS) or firewall rate-limiting during authorized scans, a shared ticker channel is used to enforce a global rate-limit across all workers. Figure 4.5 illustrates the concurrency model.

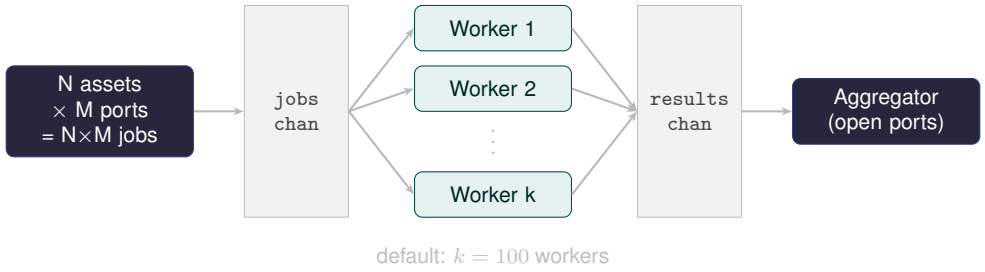



**Figure 4.5:** Worker pool pattern used in Phase 2. $N \times M$ TCP probe jobs are distributed across $k$ concurrent goroutines. Total scan time is bounded by the slowest single connection rather than their sum.

Phase 2 performs three distinct operations:

- **Service Fingerprinting:** Reading the initial TCP banner and HTTP `Server` headers to identify basic service names and versions, for example, *nginx/1.14.0*.
- **Web Technology Fingerprinting:** If an HTTP/HTTPS port is found, a separate request analyzes cookies, headers, and HTML body patterns to detect frameworks or CMS platforms, for example, WordPress, React, or Laravel.
- **Vulnerability Mapping:** Services are matched offline against a built-in CVE dataset for instant risk assessment.

### 4.2.3 Phase 3: Cloud Bucket Hunting

The engine derives potential cloud storage names from the target domain and probes providers including AWS S3, Azure Blob, and GCP Cloud Storage. HTTP status codes are analyzed to identify misconfigured, publicly accessible buckets that may leak sensitive data.

### 4.2.4 Minimal External Dependencies

A deliberate design decision was to keep the engine's dependency on third-party libraries to an absolute minimum. The entire codebase relies on a single external package: the PostgreSQL driver (`github.com/lib/pq`). All other functionality, including HTTP clients, JSON encoding, DNS resolution, TCP dialling, TLS, concurrency primitives, and the embedded web UI, is provided by Go's standard library. This simplifies deployment (the engine compiles to a single self-contained binary), removes exposure to third-party supply chain risk, and ensures the codebase remains fully understandable without knowledge of external frameworks.

## 4.3 Database Design and Rationale

The storage layer, implemented in PostgreSQL, serves as the engine's "memory." Rather than just acting as a passive repository for data, the database is strategically designed to support high-speed comparisons and ensure that the history of an attack surface is preserved accurately over time.

### 4.3.1 Architectural Rationale and Schema

A central theme in the database design was the balance between strict academic normalization and the practical needs of a high-performance scanner. The system utilizes four primary tables, as shown in Table 4.1.

**Table 4.1:** Database Table Rationale

| Table Name | Design Purpose |
|---|---|
| Assets | Uses the domain name as a *Natural Primary Key*. Since domains are globally unique, this removes the need for arbitrary numeric IDs and simplifies data lookups. |
| Ports | Employs a *Composite Key* (Domain, IP, Port, Protocol). This allows the system to track "multi-homed" assets—cases where a single domain is hosted on multiple servers simultaneously. |
| Services | Linked via a *Cascade Delete* relationship. This ensures that if an asset is removed, all its associated port and service data are automatically cleaned up, preventing "orphan" data. |
| Bucket Results | Kept independent of the asset hierarchy. Since a cloud storage bucket (like an AWS S3 bucket) often exists even if a specific subdomain is offline, this table has no hard link to the Assets table. |

Figure 4.6 shows the entity-relationship structure of the four tables.

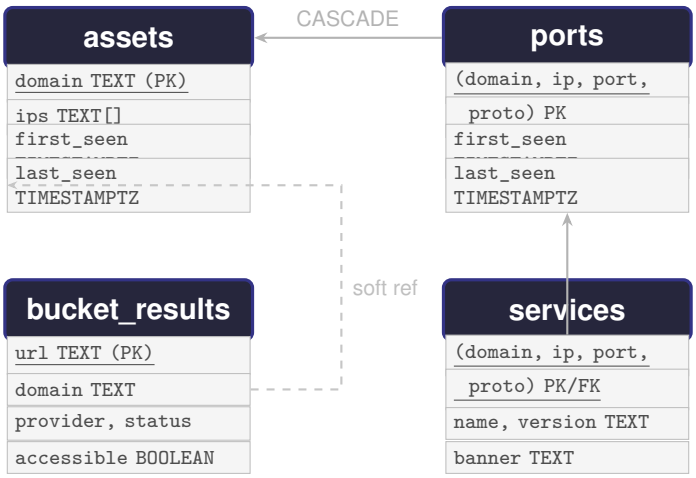



**Figure 4.6:** Entity-relationship diagram. Solid arrows indicate foreign key constraints with CASCADE DELETE. The dashed arrow indicates a soft domain reference from `bucket_results` with no enforced constraint.

An intentional design choice was storing IP addresses as a `TEXT[]` array within the Assets table. While traditional database theory might suggest a separate table for IPs, in the context of Attack Surface Management, a domain and its IPs are almost always updated as a single unit. This choice reduces the number of "joins" the database has to perform, significantly increasing the speed of the engine.

### 4.3.2 The Persistence Strategy: UPSERT and Timezones

To prevent the database from being flooded with duplicate entries, the engine utilizes "UPSERT" logic (Update or Insert). Every time a scan finds an asset, the database checks if it has seen it before. If it is new, it creates a record. If it exists, it simply updates the "Last Seen" timestamp.

A critical engineering detail is the use of the `TIMESTAMPTZ` data type. Without this, the system would be vulnerable to "silent time shifts." If a developer runs the engine on a laptop in Stockholm (UTC+2) while the database is hosted in a different region, a standard timestamp would record the time incorrectly. By using timezone-aware timestamps, the engine ensures that the timeline of an attack surface remains consistent regardless of where the scan is physically initiated.

```
INSERT INTO assets (domain, ips, first_seen, last_seen)
VALUES ($1, $2, $3, $3)
ON CONFLICT (domain) DO UPDATE
SET ips = EXCLUDED.ips, last_seen = EXCLUDED.last_seen;
```

Notice that `first_seen` is never updated. This creates a permanent "birth certificate" for every asset, allowing the user to see exactly when a specific vulnerability first appeared, no matter how many times it is scanned afterward.

### 4.3.3 Performance Optimization and Reliability

To ensure the system remains responsive as the database grows, specialized indices were created. These act like a "Table of Contents" for the database, allowing the engine to find all ports for a specific domain instantly without searching through thousands of unrelated rows.

Furthermore, the engine follows a "repeatable, self-checking migration" model. Every time the program starts, it checks if the required tables exist and creates them if they are missing. This makes the tool "plug-and-play," as a user can simply start the binary without manually setting up a complex database schema beforehand.

## 4.4 Differential Analysis and the Read-Before-Write Invariant

The primary value of this project is the ability to identify changes, to tell a security team that a new port was opened *today* that was not there *yesterday*. This is achieved through differential analysis.

During development, a significant logical bug was discovered. If the engine saves the latest scan results to the database *before* checking the history, the system effectively compares the new results against themselves. This would result in the engine claiming that "nothing has changed."

To solve this, the engine is now structurally forced to load the historical state from the database *before* the persistence phase begins. By making this order a requirement of the software's architecture, it is no longer possible for the system to accidentally lose track of changes.

## 4.5 Security Hardening and Tool Stability

As established in Section 3.3, four categories of security risk were identified and mitigated by design. This section describes the concrete mechanism used for each.

- **Safe-Scoping (SSRF Protection):** Before initiating any scan, the engine resolves the target and checks whether the resulting address falls within a private or reserved IP range. Targets that match are rejected immediately with an error.
- **Resource Clamping:** A counting semaphore limits the web server to a maximum of 50 simultaneous scans. Requests that arrive when the limit is reached receive an immediate error response rather than being queued indefinitely. Configurable parameters such as worker count and per-connection timeout are also bounded to safe maximum values regardless of what the client submits.
- **Data Redaction:** Raw database error messages, which may contain credentials embedded in the connection string, are replaced with a generic message

before being written to the log or returned to the client.

- **Web Security Headers:** Every response from the web-UI endpoint includes a `Content-Security-Policy` header and related browser security directives (`X-Frame-Options`, `X-Content-Type-Options`) to prevent clickjacking and cross-site script injection [19, 29].

# 5
# Results

This chapter presents the results of evaluating the ASM-engine through three complementary environments. An isolated Docker network was used for controlled differential analysis testing. A live scan of `scanme.nmap.org`, a host maintained by the Nmap Project and explicitly authorised for security scanning, verified the engine against a real single-host internet target. A scan of `chalmers.se` demonstrated the engine's behaviour against a target with a realistic organisational footprint, dozens of subdomains, heterogeneous services, and third-party cloud dependencies.

## 5.1 Experimental Setup

The engine was evaluated in three environments. The first was a self-contained Docker Compose stack consisting of three containers: a PostgreSQL 16 database for persistence, an intentionally outdated nginx 1.14.0 server acting as a victim service, and the ASM-engine container itself. This isolated environment made it possible to test differential change detection in a fully repeatable way without depending on external network conditions.

The second environment was a live scan over the public internet. The target `scanme.nmap.org` (IPv4 `45.33.32.156`) is a Linux server maintained by the Nmap Project (the team behind the widely-used nmap network scanner) and is permanently deployed on the public internet for the express purpose of allowing security tool developers and researchers to test their software against a real host. Unlike a local sandbox, it is reached via normal internet routing, is resolvable through public DNS, and serves genuine service banners on its open ports. Scanning it is explicitly authorised by the Nmap Project, making it the standard choice for verifying a scanner's real-world behaviour without legal or ethical concerns. All four Phase 1 sub-phases ran against this target, making it possible to observe the engine's real-world behaviour from passive reconnaissance through to port scanning and cloud storage probing. All CLI scans used 50 concurrent workers and a five-second per-connection timeout unless otherwise noted.

The third environment was a live scan of `chalmers.se`, the home domain of Chalmers University of Technology. This scan was performed as part of this academic research project. The purpose of the scan was not to assess the security posture of the organisation, but to observe how the engine behaves against a target that more closely resembles a real-world deployment. Multiple subdomain groups, assets

hosted on external cloud providers, and third-party service dependencies that only reveal themselves through DNS records. This environment provided a scale that neither the Docker sandbox nor `scanme.nmap.org` could supply.

## 5.2 OSINT-Phase Discovery

The first thing the engine does when given an internet target is run passive reconnaissance before sending a single packet to the target host. Phase 1 queried three independent OSINT sources in parallel: the Certificate Transparency log aggregator crt.sh, the passive DNS API at HackerTarget, and the Wayback Machine's CDX index. The results were then enriched through DNS resolution, PTR reverse-DNS lookups, and DNS intelligence scanning.

```
Phase 1a: passive recon for scanme.nmap.org (CT logs,
   HackerTarget, WayBack)...
Found 1 subdomain.
  [live]      scanme.nmap.org          45.33.32.156, 2600:3
   c01::f03c:91ff:fe18:bb2f
Phase 1b complete: 1 live, 0 wildcard, 0 dead -- 1 total
   subdomains discovered.
Phase 1c: PTR enrichment on discovered IPs...
Phase 1c complete: 0 new asset(s) discovered via PTR.
Phase 1d: DNS intelligence for scanme.nmap.org...
Phase 1d complete: 0 service indicator(s) found.
```

Phase 1a returned one result: the target domain itself, together with its IPv4 address `45.33.32.156` and IPv6 address `2600:3c01::f03c:91ff:fe18:bb2f`. Finding no additional subdomains is the correct result here: `scanme.nmap.org` is already a leaf-node subdomain, so there are no further sub-subdomains for any source to report. All three OSINT sources were queried and responded correctly; the pipeline ran as designed and simply found no additional hosts.

Phase 1b resolved the discovered host through DNS and classified it as live. The dual-stack result (one IPv4 and one IPv6 address) is relevant because the differential analysis engine tracks ports per IP address. A host with both address families appears as two distinct scan targets when change events are generated.

Phase 1c performed PTR reverse-DNS lookups on the discovered IP addresses to find hostnames that might not appear in certificate logs. No new assets were found, which is consistent with a dedicated server rather than shared infrastructure. Phase 1d queried DNS TXT and MX records to detect third-party service providers such as Google Workspace or Microsoft 365. No indicators were present, meaning `scanme.nmap.org` does not rely on externally-managed email or collaboration services under that subdomain.

To contrast this result, a scan of `chalmers.se` illustrates how the engine behaves against a domain with a much larger footprint. Phase 1a queried the same three OSINT sources and returned 1136 subdomains. Phase 1b resolved each subdomain

through DNS and classified 761 as live, 5 as wildcard, and 370 as dead. The large number of results is explained by internal naming conventions: hostnames such as `dhcp2-160010.ace.chalmers.se` and `f23cntf.ace.chalmers.se` appear in public certificate transparency logs even though the hosts are primarily used inside the university.

Phase 1c performed PTR reverse-DNS lookups on the resolved IP addresses and found 54 additional assets that had not appeared in any certificate log. Two of these are worth noting. The subdomain `lab.ace.chalmers.se` resolved to an address whose PTR record pointed to `lims-web.myfablims.com`, a laboratory information management system operated by an external vendor. The subdomain `ecd.ace.chalmers.se` resolved to `ec2-13-62-16-251.eu-north-1.compute.amazonaws.com`, confirming that at least one university service runs on AWS infrastructure in the eu-north-1 region. Neither finding would have been reachable through certificate logs alone; PTR enrichment was the only pipeline step that exposed them.

```
Phase 1a: passive recon for chalmers.se (CT logs,
   HackerTarget, WayBack)...
Found 1136 subdomains.
Phase 1b complete: 761 live, 5 wildcard, 370 dead -- 1136
   total subdomains discovered.
Phase 1c: PTR enrichment on discovered IPs...
  [ptr-live]  lims-web.myfablims.com
              62.181.227.151
  [ptr-live]  ec2-13-62-16-251.eu-north-1.compute.amazonaws.
   com         13.62.16.251
Phase 1c complete: 54 new asset(s) discovered via PTR.
Phase 1d: DNS intelligence for chalmers.se...
  [MX] Microsoft 365    chalmers-se.mail.protection.outlook.
   com.
Phase 1d complete: 1 service indicator(s) found.
```

Phase 1d queried DNS TXT and MX records and extracted a Mail Exchange record pointing to `chalmers-se.mail.protection.outlook.com`, classifying the institution as relying on Microsoft 365 for its email infrastructure. These external service dependencies do not appear in certificate logs or passive DNS data. Figure 5.1 shows the Phase 1d result as rendered by the web UI.

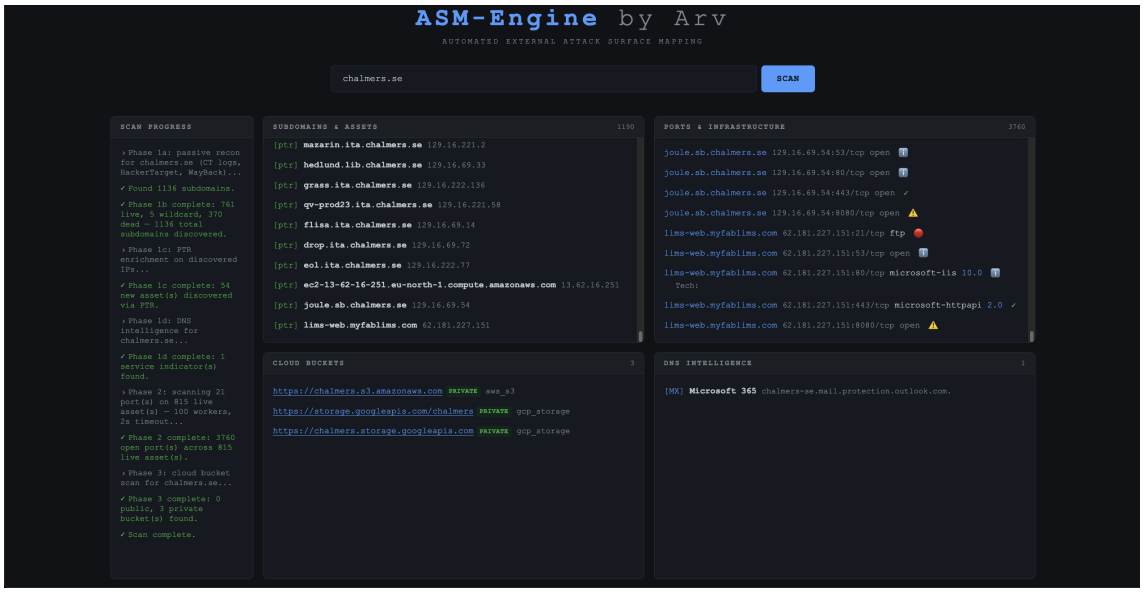



**Figure 5.1:** Web UI output for `chalmers.se`. The dashboard shows the completed scan across all four panels: discovered subdomains and assets, open ports and infrastructure findings, cloud storage buckets, and the DNS Intelligence layer identifying Microsoft 365 as the organisation's email provider.

## 5.3 Service Detection and Vulnerability Mapping

After reconnaissance, the engine moved to active scanning. Phase 2 connected to `scanme.nmap.org` on each of the 21 default ports and attempted to read a service banner from every port that responded.

```
Phase 2: scanning 21 port(s) on 1 live asset(s) -- 50 workers
   , 5s timeout...
      22/tcp    openssh          6.6.1p1
                [MEDIUM CVE-2018-15473: username enumeration
   via timing
                side-channel in authentication failure path]
      53/tcp    open
      80/tcp    apache           2.4.7
                [HIGH CVE-2021-44790: mod_lua buffer overflow
   via crafted
                request body (pre-2.4.52)]
      443/tcp   open
      8080/tcp  open
Phase 2 complete: 5 open port(s) across 1 live asset(s).
```

Five ports responded. Port 22 identified itself as OpenSSH 6.6.1p1, an SSH server from 2014. The engine's in-memory vulnerability database matched this version against CVE-2018-15473, a medium-severity vulnerability in which an attacker can determine whether a username exists on the server by measuring the time difference between successful and failed authentication attempts. This kind of information

leakage can assist a subsequent credential-stuffing or brute-force attack. Port 80 returned an Apache HTTP Server banner identifying version 2.4.7, released in 2013 and well outside the supported range. The engine matched this against CVE-2021-44790, a high-severity buffer overflow in Apache's `mod_lua` module that affects all versions prior to 2.4.52. Ports 53, 443, and 8080 accepted connections but did not return a recognisable banner within the timeout. The engine reports these as open without fabricating a service identification, which is the correct behaviour. Figure 5.2 shows the same findings rendered in the web-UI.

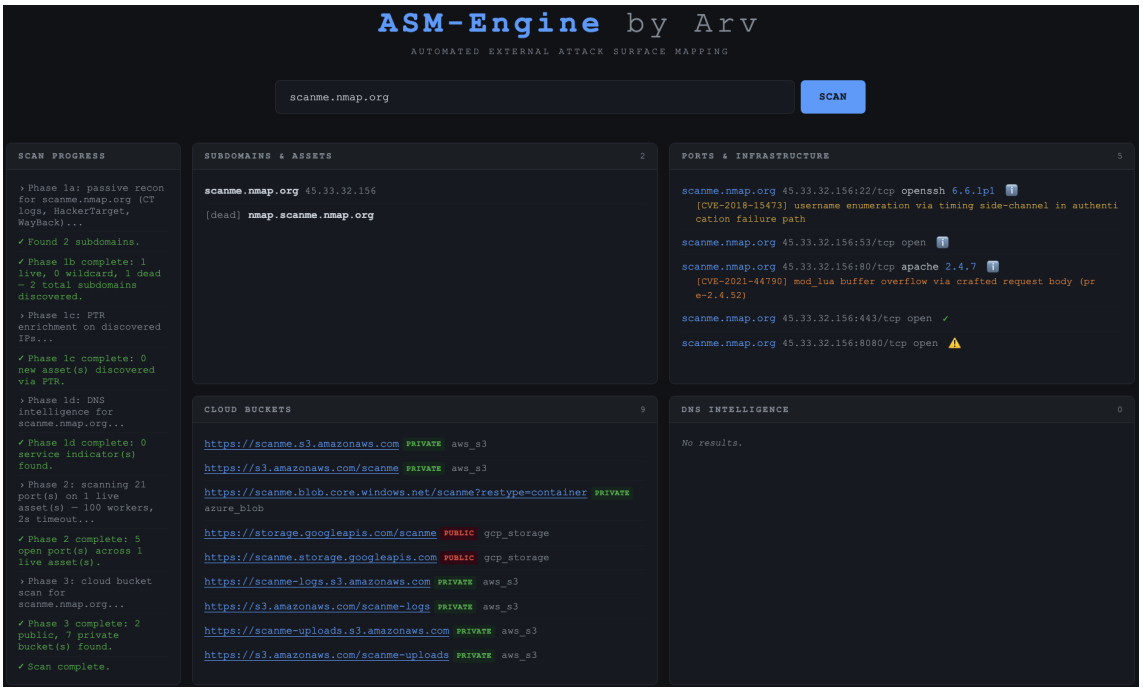



**Figure 5.2:** Web UI output for `scanme.nmap.org`.

The same detection pipeline was also exercised against the victim-service container in the Docker environment, an nginx 1.14.0 server chosen specifically because it is a version known to carry a vulnerability:

```
    80/tcp    nginx           1.14.0
              [HIGH CVE-2021-23017: off-by-one write in DNS
resolver via
              crafted CNAME response (pre-1.20.1)]
```

The match confirmed that version extraction and CVE lookup work correctly end-to-end: the engine read the banner, parsed the version string, and found the corresponding entry in the vulnerability database without any manual intervention.
The same Phase 2 pipeline was also run against `chalmers.se` as part of the scale evaluation. With 815 live assets in scope, the engine sent connection attempts to each of the 21 default ports on every host, for a total of 17 115 port checks. The scan returned 3 760 open ports. A selection of the findings is shown below:

```
Phase 2: scanning 21 port(s) on 815 live asset(s) -- 100
   workers, 2s timeout...
      21/tcp   ftp
      80/tcp   apache          2.4.37
               [HIGH CVE-2021-44790: mod_lua buffer overflow
   via crafted
               request body (pre-2.4.52)]
      443/tcp  apache          2.4.37
               [HIGH CVE-2021-44790: mod_lua buffer overflow
   via crafted
               request body (pre-2.4.52)]
      443/tcp  minio console
               Tech: Next.js (Framework)
      5432/tcp open
      80/tcp   microsoft-iis  10.0
               Tech: ASP.NET (Framework)
Phase 2 complete: 3760 open port(s) across 815 live asset(s).
```

The CVE-2021-44790 match on Apache 2.4.37 is the same rule that triggered on `scanme.nmap.org`'s Apache 2.4.7. The two hosts run different patch levels, but both fall within the vulnerable range, which covers any version before 2.4.52. This confirms that the rule applies correctly across the intended version range rather than matching only one specific version string.

FTP on port 21 was open on two assets. FTP transmits credentials and file content in plain text and is considered an insecure protocol; its presence on an internet-facing host is a finding worth investigating. PostgreSQL on port 5432 was accepting TCP connections on several assets. This does not mean the databases were accessible without authentication, but an exposed database port increases the attack surface compared to a service that is bound only to localhost. Several assets ran Minio, an open-source S3-compatible object storage server, on port 443; two of these exposed a Next.js-based management console alongside it. Two assets ran Microsoft IIS 10.0 with an ASP.NET application stack. Together, these findings represent the kind of heterogeneous service inventory that is typical of a university or medium-sized organisation, where different teams deploy different software independently with no single standard stack.

## 5.4 Differential Change Detection

One of the engine's core goals is to detect changes in an organisation's attack surface over time rather than enumerate it once and stop. To test this, three consecutive scans were run against a local target backed by PostgreSQL persistence.

The first scan targeted `localhost` on port 8081, where the engine's own web interface was listening. Because the database contained no prior record for this host and port, the engine classified the finding as a new discovery and emitted a `PORT OPENED` event:

```
  [PORT OPENED        ] 127.0.0.1:8081/tcp on localhost
```

The second scan used identical parameters. Since nothing had changed since the first scan, the engine produced no differential events at all. An absence of output here is not a failure; it is exactly what correct behaviour looks like when the attack surface is stable. Emitting no events when nothing has changed is as important as emitting the right events when something has, because false positives would erode operator trust in the tool.

The third scan added a second port. A Python HTTP server was started on port 9090 and the scan was re-run with that port included. The engine detected the new open port and reported it on both the IPv4 and IPv6 loopback interfaces:

```
  [PORT OPENED        ] 127.0.0.1:9090/tcp on localhost
  [PORT OPENED        ] ::1:9090/tcp on localhost
```

Port 8081 did not appear in the third scan's output because the engine already had it in the database. Only the genuinely new finding, port 9090, was reported as a change. This confirms that the differential logic correctly separates known findings from new ones, which is the property that makes repeated scanning operationally useful: operators see only what has changed, not the full inventory every time. Table 5.1 summarises the three scans and the events each one produced.

**Table 5.1:** Differential analysis across three consecutive scans. A silent scan (Scan 2) is the correct response when nothing has changed.

| Scan | Condition | Event produced |
|---|---|---|
| 1 | Port 8081 open; no prior history | `[PORT OPENED] 127.0.0.1:8081/tcp` |
| 2 | Port 8081 still open; no changes | *(no events — stable surface)* |
| 3 | Port 9090 newly open | `[PORT OPENED] 127.0.0.1:9090/tcp` `[PORT OPENED] ::1:9090/tcp` |

## 5.5 Cloud Storage Probing

Phase 3 probes for cloud storage resources associated with the target domain. The engine generates candidate bucket names derived from the domain and checks them against AWS S3, Azure Blob Storage, and Google Cloud Storage.

```
Phase 3: cloud bucket scan for scanme.nmap.org...
  [private]  https://scanme.s3.amazonaws.com
      (aws_s3)
```

```
  [private]  https://s3.amazonaws.com/scanme
      (aws_s3)
  [private]  https://scanme.blob.core.windows.net/scanme...
      (azure_blob)
  [public]   https://storage.googleapis.com/scanme
      (gcp_storage)
  [public]   https://scanme.storage.googleapis.com
      (gcp_storage)
  [private]  https://scanme-logs.s3.amazonaws.com
      (aws_s3)
  [private]  https://s3.amazonaws.com/scanme-logs
      (aws_s3)
  [private]  https://scanme-uploads.s3.amazonaws.com
      (aws_s3)
  [private]  https://s3.amazonaws.com/scanme-uploads
      (aws_s3)
Phase 3 complete: 2 public, 7 private bucket(s) found.
```

Nine candidate URLs were checked. The engine classifies each result using the HTTP response code: 200 means the bucket exists and its contents are publicly readable, 403 means the bucket exists but access is denied, and 404 means no such bucket was found. Seven buckets returned 403, indicating that storage resources with these names exist on AWS and Azure but are not publicly accessible. Two Google Cloud Storage buckets returned 200 and were classified as public.

The three-state distinction matters in practice. A security team that only checked for publicly accessible buckets would miss the seven private findings entirely, yet the existence of those buckets, including their names, providers, and access controls, is itself information about the organisation's cloud footprint. Knowing that a bucket named `scanme-logs` exists on AWS, even if currently private, is relevant context for ongoing monitoring.

The same Phase 3 scan was also run against `chalmers.se`:

```
Phase 3: cloud bucket scan for chalmers.se...
  [private]  https://chalmers.s3.amazonaws.com          (
   aws_s3)
  [private]  https://storage.googleapis.com/chalmers    (
   gcp_storage)
  [private]  https://chalmers.storage.googleapis.com    (
   gcp_storage)
Phase 3 complete: 0 public, 3 private bucket(s) found.
```

Three candidate URLs were checked: one on AWS S3 and two on Google Cloud Storage. All three returned 403, meaning the storage resources exist but are not publicly readable. No publicly accessible bucket was found. The lower count compared to `scanme.nmap.org` (3 versus 9) reflects the domain name length. The engine derives candidate names from the root domain, and a shorter name like `chalmers` maps to fewer naming-pattern combinations than `scanme.nmap.org`. The important property is not the count but the classification. Every candidate was checked

and its access state was correctly reported.

## 5.6 Persistence and Storage Verification

The PostgreSQL integration was verified through an automated test suite that exercises the storage layer against a real database instance. Running against a real database rather than a mock is intentional. Mock-based tests verify that code calls the right methods in the right order, but they do not catch schema mismatches, constraint violations, or transaction isolation problems that only appear when the code talks to an actual database engine.

The suite covers 24 distinct scenarios including SQL injection resistance across four different payload types, concurrent writes from 20 simultaneous goroutines, correct enforcement of foreign key constraints, and timestamp ordering guarantees.

```
ok  github.com/arvin-sudo/asm-engine/internal/storage  4.033s
```

All 24 tests passed in 4.033 seconds with the Go race detector enabled. The race detector instruments the binary at compile time and reports any concurrent memory accesses that are not protected by a synchronisation primitive. Passing with it enabled provides strong evidence that the storage layer has no data races under concurrent load.

The persistence design centres on an UPSERT operation: when the engine writes a port result, it either inserts a new row with a `first_seen` timestamp, or updates the existing row's `last_seen` timestamp if that port was already known. Running the same scan twice produces the same database state: the second run does not duplicate rows. Differential events are derived by comparing the current scan's findings against what was already stored before the writes, which is why the engine must read the prior history before performing any inserts.

## 5.7 Performance

All local Docker scans completed in approximately 130 milliseconds. This is fast not because the engine performs any particularly clever computation, but because port scanning is almost entirely network I/O bound. The CPU is idle for most of the scan while the engine waits for TCP connections to complete or time out. The worker pool, configured to 50 workers in these tests and 100 by default, keeps many connections in flight simultaneously instead of waiting for each one to finish before starting the next.

The practical effect of concurrency scales with the scope of the scan rather than the speed of any individual connection. On a local Docker network where round-trip times are near zero, all 21 ports complete almost instantly regardless of how many workers are running. On a real internet target such as `scanme.nmap.org`,

each connection incurs tens of milliseconds of network latency. Without parallelism, scanning 21 ports on a single host at 50 ms per connection would take over a second; with 50 workers, those connections overlap and the total time is bounded by the slowest individual connection rather than their sum. For an organisation with hundreds of subdomains spread across many IP addresses, the same principle means that the full scan completes in roughly the time it takes to scan one port on one host sequentially. The 130-millisecond completion time on a 21-port, single-host Docker scan and the elimination of sequential blocking through the worker pool together demonstrate that the "high-performance" goal stated in Chapter 1 is met in practice. Scan time is bounded by the slowest individual connection rather than the sum of all connections.

# 6

# Discussion

The previous chapters described what the engine does and what happened when it was tested. This chapter steps back from those results to ask what they actually mean, where the design worked well, where it fell short, and what the experience of building this project as a solo developer was like.

## 6.1 Results About ASM in Practice

The results in Chapter 5 present numbers and terminal output, but those outputs need interpretation to be useful. The most instructive result from the `scanme.nmap.org` scan is not the two CVEs that were matched, it is the fact that the OSINT phase correctly returned nothing extra. `scanme.nmap.org` is already a leaf-node subdomain, meaning there are no deeper subdomains for any source to report. All three passive sources ran, responded, and found nothing new. A less carefully designed engine might treat an empty result as a failure, however this one treats it as a valid answer.

The silent second scan in the differential analysis test is equally important. When the attack surface has not changed, the engine should say nothing, not repeat what was already known. Emitting no events on a stable surface is as critical as emitting the right event on a change, because false positives would make the engine noisy and operators would stop trusting it.

The 130-millisecond local scan time looks fast but needs context. On a Docker network where all round-trip times are nearly zero, even a sequential scanner would finish quickly. The meaningful performance claim from Chapter 3 is that the worker pool bounds total scan time by the slowest single connection, not by the sum of all connections. On a real internet target, where each TCP connection takes tens of milliseconds, this distinction is what makes the engine practical at scale.

The three-state cloud storage classification (public, private, and not found) also deserves attention beyond the raw numbers. A security team focused only on publicly accessible buckets would miss the seven private AWS findings entirely. Knowing that a storage resource with a given name exists, even when it is access-controlled, is useful intelligence for ongoing monitoring.

## 6.2 Design Decisions and Their Trade-offs

Every major design decision in the engine involved a trade-off between simplicity and flexibility. Three decisions stand out as non-obvious enough to discuss.

The first is the embedded CVE database. Rather than querying an external vulnerability feed at runtime, all CVE rules are compiled directly into the engine binary. This makes scans fully reproducible and removes the dependency on a third-party service during a scan. The cost is that the database does not update automatically, meaning, new vulnerabilities published after the last manual update will not be detected. For a research prototype this trade-off was correct, but for a production ASM product it would need to be rethought.

The second decision was to keep external dependencies to a minimum. The entire codebase relies on a single third-party package: the PostgreSQL driver. Every other capability comes from Go's standard library. This keeps the binary self-contained and makes the code readable to anyone who knows Go, without needing to understand a framework. The downside is that some functionality that would be one import in another project had to be written by hand, such as the HTTP client configuration and retry logic for OSINT sources.

The third decision was to store IP addresses as a `TEXT[]` array inside the Assets table rather than as a separate normalised table. Traditional database design would prefer a separate table with foreign keys, but in the context of ASM a domain and its IP addresses are almost always read and written together as a single unit. Collapsing them into an array removes a join on every query and keeps the write path simpler. The trade-off is a schema that is less strictly normalised, which would complicate any future query that needs to search by IP address alone.

## 6.3 Limitations of the Engine and the Evaluation

The engine has several limitations that are worth stating clearly, both to set accurate expectations and to give a starting point for future improvements.

The CVE database is compiled into the binary and not updated automatically. Any vulnerability disclosed after the last manual update will not be detected, meaning the engine could give a false sense of security for recently published CVEs. This was an acceptable trade-off for a research prototype but would need to be addressed before the engine could be used operationally.

OSINT coverage depends entirely on three external services: crt.sh, HackerTarget, and the Wayback Machine. An organisation that has never registered a TLS certificate, has no passive DNS history, and has never been archived by the Internet Archive would produce no results from Phase 1. The engine would then have nothing to pass to Phase 2, and the scan would complete without finding any assets at all.

Only TCP ports are scanned. Services that communicate exclusively over UDP, such as DNS resolvers, SNMP agents, and NTP servers, are not fingerprinted. This is noted as a deliberate delimitation in Section 1.5, but it is worth restating as a practical constraint on what the engine can see.

The core performance evaluation used a single-host Docker sandbox and one authorised internet target, `scanme.nmap.org`. To address the scale gap this creates, `chalmers.se` was included as a third evaluation environment. That scan covered 815 live assets and produced 3,760 open ports, which is closer to what a real organisation looks like. Even so, the scan was run once from a single external vantage point rather than continuously, which limits what it demonstrates about change detection at real-world scale. The 130-millisecond completion time is accurate for the Docker environment but does not reflect how long a scan over hundreds of internet-facing assets across different networks would take.

Cloud bucket probing derives candidate names from the target domain using a fixed set of naming patterns. Buckets with names that have no obvious relationship to the target domain will not be found. The engine can only discover storage resources it knows to look for.

## 6.4 Positioning the Engine Among Existing Tools

Port scanners and reconnaissance tools already exist and are widely used. Nmap has been the standard for network scanning since 1997 [10]. Shodan (shodan.io) [21] indexes internet-facing services continuously at global scale. Commercial ASM platforms provide continuous monitoring with professional support. It is worth being clear about where this engine fits relative to those tools.

The engine is not trying to replace any of them. Its purpose, stated in Section 1.3, is to demonstrate that the full ASM lifecycle, covering passive discovery, active scanning, vulnerability correlation, differential tracking, and cloud storage probing, can be implemented in a single self-contained binary using only Go's standard library and one external database driver. No API keys are required. No third-party scanning infrastructure is needed. Anyone who can run a Go binary and has access to a PostgreSQL database can use it and inspect every line of what it does.

Because the engine is open source, it can also be extended. The `Discoverer` interface described in Section 4.1.1 makes it straightforward to add new OSINT sources without modifying the core pipeline. The CVE database can be updated by editing a single file. These extension points were designed deliberately so that the engine remains useful beyond the scope of this thesis, and so that other developers who want to build on it have a clean foundation to start from.

## 6.5 Industry Collaboration and Real-world Context

Nordic Defender, a Swedish cybersecurity company, served as the industry partner for this project. Mahsima Jalooli, the company's supervisor, provided guidance throughout the development and thesis writing process. This section describes how that collaboration shaped the project and what it revealed about where a tool like this fits in a real security workflow.

Nordic Defender reviewed drafts of the thesis report at several points during the project. Their feedback highlighted out practical gaps in the initial design, including the lack of a rate-limiting mechanism in the port scanner and the absence of a structured approach to reporting only what had changed between scans rather than repeating the full inventory every time.

The company described the typical scenarios in which a tool like this is used in practice. When a new client is onboarded, a security team runs an initial scan to build a picture of what the client has exposed to the internet. Clients frequently have forgotten staging and development servers that they did not document. Those forgotten servers often run outdated software and are not covered by any patch management process, making them among the highest-risk assets on the surface. A tool that can automatically flag these during the first scan is immediately useful.

For existing clients, the value comes from scheduled re-scans after deployments or on a fixed schedule such as once a week. What security teams need from a re-scan is not the full inventory repeated again, they already have that from the previous run. What they need is a list of what is new or different. A new open port, a version that changed, or a previously unknown subdomain that appeared overnight are the signals worth acting on. This is exactly the problem that the differential analysis component in this engine was designed to solve.

Nordic Defender also noted that cloud storage bucket misconfiguration is one of the most common findings in real engagements. Many organisations created storage resources during development, gave them names derived from the company or product name, and never restricted access. Attackers routinely check predictable bucket names as part of reconnaissance. The cloud probing phase in this engine automates that same check, and the three-state classification (public, private, and not found) mirrors the information a security analyst would record when investigating manually.

A tool that requires minimal setup, produces output that non-specialists can read, and alerts on changes automatically is far more likely to be used consistently than one that requires configuration and interpretation expertise. That feedback directly influenced the decision to build the engine as a single self-contained binary with no API key requirements, and to make the differential output the primary output rather than an optional feature.

## 6.6 Working Solo: Freedom and Its Costs

Working on this project alone had clear advantages. There was no coordination overhead, no need to synchronise design decisions with other contributors, and no dependency on anyone else's schedule or availability. When a decision had to be made, it could be made and acted on immediately. The architecture could be changed freely between phases without negotiating with a team, which made the iterative refactoring cycle described in Section 3.2 straightforward to execute.

The disadvantages were just as real. Without a second person reviewing the code, early design mistakes went unnoticed longer than they would have on a team. There was no natural forcing function for schedule discipline. Progress relied entirely on self-motivation, which was harder to sustain during phases where the work felt less visible. Perhaps most importantly, there was no one to challenge assumptions or suggest a different direction when a design choice seemed obvious but was not. The freedom to decide everything alone also meant there was no external check on whether a decision was actually good.

The four formal hardening passes described in Section 3.2.1 were partly a structural response to this. Each pass served the function that code review from a colleague would have provided on a team project. By scheduling them deliberately, the process introduced the kind of critical distance from the code that normally comes from another person looking at it fresh.

## 6.7 Ethical, Social, and Ecological Considerations

Any tool that performs automated network reconnaissance raises questions about responsible use, and those questions deserve a direct answer rather than a footnote.

From an ethical standpoint, the most important property of the engine is that it is a dual-use instrument. The same capabilities that help a security team map their own infrastructure can be pointed at a target the operator does not own. This risk is not theoretical. The engine was designed with it in mind from the start. The server-side request forgery guard in Section 4.5 prevents the web interface from being used as a proxy to reach internal services on the operator's own network. The scope enforcement logic in Section 3.3 ensures that active probes are only sent to subdomains that were confirmed to belong to the declared target domain. The evaluation throughout this project was conducted exclusively against targets where explicit authorisation existed: the Docker sandbox, `scanme.nmap.org` which the Nmap Project makes permanently available for exactly this purpose, and `chalmers.se` which was scanned as part of this academic research project by a registered student at the institution. Responsible use of any reconnaissance tool rests ultimately with the operator, but the design choices in this engine reduce the surface for accidental or malicious misuse.

From a social standpoint, releasing the engine as open source has a concrete ben-

efit. Commercial ASM platforms typically require ongoing subscription fees and dedicated security staff to operate, which puts continuous external attack surface monitoring out of reach for smaller organisations and independent developers. An open-source engine that requires only a Go binary and a PostgreSQL database lowers that barrier significantly. Any organisation that can run a server can run this engine, inspect every line of its behaviour, and extend it for their own context.

From an ecological standpoint, the engine's resource footprint is small. Passive OSINT, which forms Phase 1, retrieves data from existing public indexes without sending any probes to the target infrastructure at all. Active scanning in Phase 2 completes in roughly 130 milliseconds on a typical target and terminates immediately, leaving no persistent background processes. The entire scan produces a few kilobytes of database writes. This compares favourably to commercial ASM platforms that run continuous background crawlers around the clock. The design choice to bound scan time by the slowest single connection rather than scaling indefinitely with the number of workers also limits the peak network load that any single scan can generate.

## 6.8 Future Improvements and Project Expansions

The engine is complete as a research prototype, but several improvements would be needed before it could be used as a production tool. This section describes the most important ones.

**Automated CVE database updates.** The vulnerability database is compiled into the binary and updated manually. Any CVE published after the last update will not be detected, which means the engine could give a false sense of security for recently disclosed vulnerabilities. A future version could fetch updates from the NIST National Vulnerability Database [30] API on startup or on a regular schedule, keeping the ruleset current without requiring a code change.

**Additional OSINT sources.** Phase 1 currently queries three passive sources: crt.sh, HackerTarget, and the Wayback Machine. Services such as SecurityTrails, VirusTotal, and Shodan maintain their own passive DNS and certificate records and would increase subdomain coverage. Because the engine uses the `Discoverer` interface described in Section 4.1.1, new sources can be added without changing any pipeline code.

**UDP service discovery.** Only TCP ports are scanned. Services that communicate exclusively over UDP, including DNS resolvers, SNMP agents, and NTP servers, are invisible to the current engine. Adding UDP probing requires a different strategy because UDP is connectionless, there is no three-way handshake to confirm a port is open, and many services only respond to a correctly formatted protocol message rather than a raw connection attempt.

**Alerting and notifications.** At present, differential events appear in the web

dashboard and on the command-line interface. For the tool to be operationally useful without requiring someone to re-run it manually, it would need to send alerts automatically when a new port opens or a new asset appears. A webhook or email notification on each change event would remove the manual monitoring step.

**Performance at extreme large scale, a note on language choice.** For the scope of this project, tens to hundreds of subdomains, Go's concurrency model is well suited and the worker pool keeps scan times short. If the engine were extended into a product that scans thousands of organisations simultaneously, the memory allocator and garbage collector in Go would introduce non-deterministic pauses that become noticeable under sustained high throughput. A rewrite in C or C++ would give complete control over memory layout, raw socket access, and system call overhead, eliminating the runtime entirely. Go was the right choice here because it allowed a correct, concurrent, and maintainable codebase to be built within the timeframe of a bachelor project, and the performance it delivers is more than adequate for the intended use case. A systems-level rewrite would only be warranted if a real performance ceiling were measured against a production workload.

# 7

# Conclusion

Organisations that deploy software rapidly tend to lose track of what they have exposed to the internet. Forgotten staging servers accumulate on unused subdomains, cloud storage buckets are created during development and never secured, and outdated services keep running on undocumented ports long after the teams that deployed them have moved on. Discovering this exposure manually does not scale. By the time a security team has audited the current inventory, the inventory has already changed. This project set out to address that problem by building an automated ASM-engine that discovers, fingerprints, and tracks an organisation's externally reachable assets from the perspective of an attacker.

Five goals were established at the start of the project, and all five were met.

The first goal was to design a modular, interface-based pipeline architecture. The engine is structured as three sequential phases, each communicating through typed channels and depending on interfaces rather than concrete implementations. Every external data source and the database layer can be replaced or extended without modifying any other part of the system.

The second goal was performance sufficient for operational use. A worker pool with a configurable size bounds the total scan time by the slowest single connection rather than the sum of all connection attempts. In practice, scanning all 21 target ports on the local Docker sandbox completed in approximately 130 milliseconds, well within the interactive range.

The third goal was persistent state and differential analysis. Rather than printing the same full inventory on every scan, the engine records each finding with `first_seen` and `last_seen` timestamps, and at the end of each run surfaces only what changed: newly opened ports, version changes, and assets that disappeared. The read-before-write ordering invariant ensures that the diff is always computed against the state that was in the database at the start of the scan, not mid-run.

The fourth goal was accessible output for two different operator workflows. Both a web-UI with real-time streaming via Server-Sent Events and a CLI were implemented, sharing the same pipeline and storage layer.

The fifth goal was ethical evaluation under controlled conditions. All active scanning was performed against targets where explicit authorisation existed. An isolated

Docker Compose sandbox, `scanme.nmap.org` which is permanently authorised for security tool testing by the Nmap Project, and `chalmers.se` which was scanned as part of this academic research project at the institution.

The project broadly followed the waterfall-inspired timeline described in Chapter 3. The four sequential development phases completed in order, and no phase had to be abandoned or significantly redesigned. Two logic bugs surfaced during integration testing and were resolved before the evaluation phase. Neither affected the final results, both were resolved before the evaluation phase. The four formal hardening passes at the end of the implementation phase served the code-review function that would normally come from a team on a larger project.

Two design decisions stand out as particularly successful. The typed event channel that runs through the pipeline decouples the scanning logic from its output entirely. The same events feed the terminal, the web interface, and the differential analysis component without any of them knowing about the others. The read-before-write invariant in the differential analysis is correct by construction rather than by convention. Because the engine reads the previous state once at the start and writes the new state once at the end, there is no window for a race condition to produce a false positive or false negative.

Working with Nordic Defender throughout the project introduced constraints and priorities that a purely academic setting would not have surfaced. The differential analysis component and the single-binary deployment model were both shaped by direct feedback about what makes a tool usable in a real security workflow. Operators do not want to re-read a full inventory on every run.

The most impactful next steps are discussed in detail in Section 6.8. In brief, automated CVE feed updates from the NVD API would remove the one manual maintenance step that currently requires a developer to act, and adding UDP service discovery would surface a whole class of services, DNS resolvers, SNMP agents, and NTP servers, that the current engine cannot see at all.

Go's concurrency model made it possible to build a tool that is simultaneously fast enough for operational use, readable enough for a single developer to maintain, and simple enough to deploy without any infrastructure beyond a database. The standard library provided everything needed for concurrent network programming, context cancellation, HTTP serving, and binary embedding of the frontend. The entire engine depends on exactly one external package. This project demonstrates that high-performance network security tooling does not require a specialist language or a large team. Careful architecture and idiomatic Go are sufficient to deliver a production-quality result within the timeframe of a bachelor project.

# Bibliography


[1] Bazzell, M. (2022). *Open Source Intelligence Techniques: Resources for Searching and Analyzing Online Information*. 9th edn. Independently published.

[2] Laurie, B., Langley, A. and Kasper, E. (2013). Certificate Transparency. *RFC 6962*. Internet Engineering Task Force (IETF).

[3] Weimer, F. (2005). Passive DNS Replication. *Proceedings of the 17th Annual FIRST Conference on Computer Security Incident Handling*. Singapore: FIRST.

[4] Internet Archive (2024). *Wayback Machine* [online]. San Francisco, CA: Internet Archive. Available at: https://web.archive.org [Accessed: May 2026].

[5] Kitterman, S. (2014). Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. *RFC 7208*. Internet Engineering Task Force (IETF).

[6] Klensin, J. (2008). Simple Mail Transfer Protocol. *RFC 5321*. Internet Engineering Task Force (IETF).

[7] Donovan, A.A.A. and Kernighan, B.W. (2015). *The Go Programming Language*. Addison-Wesley.

[8] Ajmani, S. (2014). Go Concurrency Patterns: Context. *The Go Programming Language Blog* [online]. Available at: https://go.dev/blog/context [Accessed: May 2026].

[9] Hickson, I. (2015). *Server-Sent Events*. W3C Recommendation. World Wide Web Consortium.

[10] Lyon, G.F. (2009). *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure.Com LLC.

[11] MITRE Corporation (2024). *Common Vulnerabilities and Exposures (CVE)* [online]. Available at: https://cve.mitre.org [Accessed: May 2026].

[12] Martin, R.C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

[13] NIST (2008). *Technical Guide to Information Security Testing and Assessment*. NIST Special Publication 800-115. National Institute of Standards and Technology.

[14] McNab, C. (2007). *Network Security Assessment*. 2nd edn. O'Reilly Media.

[15] Anderson, R.J. (2020). *Security Engineering: A Guide to Building Dependable Distributed Systems*. 3rd edn. Wiley.

[16] FIRST (2019). *Common Vulnerability Scoring System v3.1: Specification Document* [online]. Forum of Incident Response and Security Teams. Available at: https://www.first.org/cvss/specification-document [Accessed: May 2026].

[17] Mell, P. and Grance, T. (2011). *The NIST Definition of Cloud Computing*. NIST Special Publication 800-145. National Institute of Standards and Technology.
[18] Pike, R. (2012). Go Concurrency Patterns. *Google I/O 2012* [online]. Available at: https://talks.golang.org/2012/concurrency.slide [Accessed: May 2026].
[19] OWASP (2021). *OWASP Top Ten* [online]. Open Web Application Security Project. Available at: https://owasp.org/www-project-top-ten/ [Accessed: May 2026].
[20] Ylonen, T. and Lonvick, C. (2006). The Secure Shell (SSH) Authentication Protocol. *RFC 4252*. Internet Engineering Task Force (IETF).
[21] Matherly, J. (2015). *Complete Guide to Shodan*. Leanpub.
[22] Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.
[23] Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. 2nd edn. Addison-Wesley.
[24] Beck, K. (2002). *Test-Driven Development: By Example*. Addison-Wesley.
[25] Mockapetris, P. (1987). Domain Names – Implementation and Specification. *RFC 1035*. Internet Engineering Task Force (IETF).
[26] Postel, J. (1981). Transmission Control Protocol. *RFC 793*. Internet Engineering Task Force (IETF).
[27] Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239).
[28] Strom, B.E., Applebaum, A., Miller, D.P., Nickels, K.C., Pennington, A.G. and Thomas, C.B. (2020). *MITRE ATT&CK: Design and Philosophy*. McLean, VA: The MITRE Corporation [online]. Available at: `https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf` [Accessed: May 2026].
[29] West, M., Barth, A. and Veditz, D. (2016). *Content Security Policy Level 2*. W3C Recommendation. World Wide Web Consortium [online]. Available at: https://www.w3.org/TR/CSP2/ [Accessed: May 2026].
[30] NIST (2024). *National Vulnerability Database* [online]. National Institute of Standards and Technology. Available at: https://nvd.nist.gov [Accessed: May 2026].
[31] Sectigo (2024). *crt.sh – Certificate Search* [online]. Available at: https://crt.sh [Accessed: May 2026].
[32] HackerTarget (2024). *Passive DNS Lookup* [online]. HackerTarget.com. Available at: https://hackertarget.com/passive-dns-lookup/ [Accessed: May 2026].
[33] Rentrop, C. and Zimmermann, S. (2012). Shadow IT evaluation model. *Proceedings of the 2012 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, pp. 1023–1027.

# A

# PostgreSQL Database Schema

This appendix lists the complete PostgreSQL schema used by the ASM engine. The schema is executed automatically on startup by the `migrate()` function in `internal/storage/postgres_store.go`; every statement uses `IF NOT EXISTS` so the function is safe to run repeatedly against an existing database. The design rationale for each table is discussed in Section 4.3.

```
CREATE TABLE IF NOT EXISTS assets (
    domain     TEXT        PRIMARY KEY,
    ips        TEXT[]      NOT NULL,
    first_seen TIMESTAMPTZ NOT NULL,
    last_seen  TIMESTAMPTZ NOT NULL
);

CREATE TABLE IF NOT EXISTS ports (
    asset_domain TEXT        NOT NULL
                             REFERENCES assets(domain) ON
   DELETE CASCADE,
    ip           TEXT        NOT NULL,
    number       INTEGER     NOT NULL,
    proto        TEXT        NOT NULL,
    first_seen   TIMESTAMPTZ NOT NULL,
    last_seen    TIMESTAMPTZ NOT NULL,
    PRIMARY KEY (asset_domain, ip, number, proto)
);

CREATE TABLE IF NOT EXISTS services (
    asset_domain TEXT        NOT NULL,
    ip           TEXT        NOT NULL,
    port_number  INTEGER     NOT NULL,
    proto        TEXT        NOT NULL,
    name         TEXT        NOT NULL DEFAULT '',
    version      TEXT        NOT NULL DEFAULT '',
    banner       TEXT        NOT NULL DEFAULT '',
    first_seen   TIMESTAMPTZ NOT NULL,
    last_seen    TIMESTAMPTZ NOT NULL,
    PRIMARY KEY (asset_domain, ip, port_number, proto),
    FOREIGN KEY (asset_domain, ip, port_number, proto)
        REFERENCES ports(asset_domain, ip, number, proto)
        ON DELETE CASCADE
);
```

```
CREATE TABLE IF NOT EXISTS bucket_results (
    url         TEXT        PRIMARY KEY,
    domain      TEXT        NOT NULL,
    provider    TEXT        NOT NULL,
    status      INTEGER     NOT NULL,
    accessible  BOOLEAN     NOT NULL,
    first_seen  TIMESTAMPTZ NOT NULL,
    last_seen   TIMESTAMPTZ NOT NULL
);

CREATE INDEX IF NOT EXISTS idx_ports_asset_domain
    ON ports(asset_domain);
CREATE INDEX IF NOT EXISTS idx_services_asset_domain
    ON services(asset_domain);
CREATE INDEX IF NOT EXISTS idx_buckets_domain
    ON bucket_results(domain);
```

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**
**CHALMERS UNIVERSITY OF TECHNOLOGY**
Gothenburg, Sweden
www.chalmers.se


CHALMERS
UNIVERSITY OF TECHNOLOGY