Archives available at journals.mriindia.com

**Multidisciplinary Journal of Research in Engineering and Technology**

ISSN: 2348-6953

Volume 13 Issue 01, 2026

# Design and Implementation of a Concurrent Port Scanning and Vulnerability Analysis Library in Go

Vedant Ankush Karande
*Indian Institute of Technology Delhi, New Delhi, India*
*Email: vedantk6403@gmail.com*

**Peer Review Information**

*Submission: 12 April 2026*

*Revision: 02 May 2026*

*Acceptance: 23 May 2026*

**Keywords**

*Network Security, GoLang, Concurrency, Port Scanning, Service Enumeration, Vulnerability Analysis, Gorou-tines*

**Abstract**

Network reconnaissance is the most critical first step in understanding and securing any computer networks. Before defending a system, security professionals must know exactly which ports are open and what services are running on them. However, using large, traditional port scanning software can sometimes be too slow or too difficult to integrate into modern, automated security systems and cloud environments. This paper presents the design, architecture, and early testing of a fast, lightweight port scanning and service identification tool built entirely in the Go programming language (GoLang). By utilizing Go's built-in concurrency features—specifically goroutines and channels the proposed tool can check thousands of network ports at the exact same time without overwhelming the host system's memory or central processing unit. This document deeply explores the internal architecture of the tool, the step-by-step logic it uses to check ports and grab service banners, and the impact of operating system limits on concurrent network sockets. Furthermore, we compare its speed against traditional single-threaded methods and the industry standard Nmap tool across multiple experiments. The results prove that our Go library drastically reduces scan times, making it a highly capable, scalable, and easily integrable solution for modern cybersecurity development.

## Introduction

In today's fast-paced digital landscape, computer networks are growing larger, more dynamic, and more complicated every single day. The rise of cloud computing, microservices, and connected Internet of Things (IoT) devices means that a single company might have thousands of active IP addresses and virtual machines running at any given time. Because of this rapid infrastructure growth, finding weak spots and unauthorized open ports quickly has become a major challenge for security professionals. Discovering these open doors before malicious attackers exploit them is a critical part of modern cybersecurity defense strategies.

While there are already excellent tools available on the market, such as the famous Nmap scanner, they were primarily designed decades ago. They are often built as monolithic applications intended to be used by a human operator typing commands into a terminal. If a modern software developer wants to build a custom, automated security application such as a dashboard that automatically scans servers every hour forcing their code to trigger an external tool like Nmap in the background can be clunky, heavy, and slow to process.

To solve this problem of integration and speed, we engi-neered a brand new scanning library

from scratch. We selected the Go programming language (GoLang) because it was built specifically for modern, multi-core processors. It is famous for being incredibly fast, safe with computer memory, and most importantly, it handles "concurrency" (doing many things at the exact same time) better than almost any other popular language. This paper details how we constructed a simple but highly effective library that developers can drop directly into their own codebase to perform rapid, parallel network scanning and vulnerability detection without relying on heavy external software.

**Literature Review and Related Work**

Before designing a new architectural model, it is important to analyze the current landscape of network scanning tools to understand their strengths and weaknesses. To get a baseline for comparison and to identify the Research Gap.

**1. Traditional Scanners (Nmap)**

Nmap (Network Mapper) is widely considered the undis-puted industry standard for network discovery and security auditing. It uses raw IP packets in the standard ways to deter-mine what hosts are available on the network, what services (application name and version) those hosts are offering, and what operating systems they are running. While Nmap is incredibly accurate and feature rich, its vast array of features makes it a very large program. Furthermore, its default timing templates are heavily focused on stealth and avoiding network congestion, which can result in long scan times when raw speed is the primary goal.

**2. Asynchronous Scanners (Masscan and ZMap)**

To address the speed limitations of traditional scanners, asynchronous tools like Masscan and ZMap were created. These tools bypass the host operating system's standard net-work stack entirely and generate raw packets directly. This allows them to scan the entire public Internet in a matter of minutes. However, this extreme speed comes with a major tradeoff: they are "stateless." Because they do not maintain a full connection state, it is very difficult for them to perform

deep banner grabbing or complex service identification. They are excellent for finding open ports, but poor for deep vul-nerability analysis. Which most of the most the professional need but have to rely on other tools or sometimes have to do manual Reconnaissance to achieve the intended information regarding Target.

**3. The Necessity for a Native Library**

The gap in the current ecosystem is the lack of a native, highly concurrent library built specifically for developers. There is a strong need for a tool that sits perfectly in the middle: faster than Nmap by utilizing aggressive concurrency, but more detailed than Masscan by maintaining proper TCP connection states to extract software versions. Building this tool natively in Go fills this exact gap.

I. PROPOSED SYSTEM ARCHITECTURE

To achieve maximum scanning speeds, our architecture had to be designed to eliminate the waiting time to the maximum possible extend. A traditional, un-optimized scanner operates sequentially: it sends a request to port 1, waits for an answer, processes the answer, and only then moves to port

2. In network programming, waiting for a response over a network is the biggest bottleneck. Our proposed system uses a highly optimized "fan-out/fan-in" concurrency model to check hundreds of ports simultaneously.

As depicted in Figure 1, the execution flow begins when the user or parent application inputs the target IP address and the range of ports to be audited. Instead of assigning these ports to a single processing thread, the system serializes the port numbers and funnels them into a central pipeline known as a "Job Queue." In Go, this is implemented using a buffered Channel.

Simultaneously, the system instantiates a large pool of "Workers." In Go, these workers are called goroutines, which are functions capable of running concurrently with other functions. Goroutines are incredibly lightweight compared to traditional operating system threads; a standard computer can easily run tens of thousands of goroutines simultaneously. All of these workers eagerly pull port numbers from the Job Queue. As soon as a worker finishes checking a port, it pushes the result into a "Result Channel" (the fan-in phase) and immediately returns to the queue to request another job. Finally, a dedicated aggregator function collects all the finished results from the Result Channel and organizes them for terminal output or structured JSON export.

II. SCANNING METHODOLOGY AND LOGIC

A fast architectural framework is only effective if the logic executed within the workers is smart, accurate, and resilient to errors. Networks are inherently unreliable; packets drop, firewalls block connections, and servers crash. The worker logic must account for all of these scenarios.

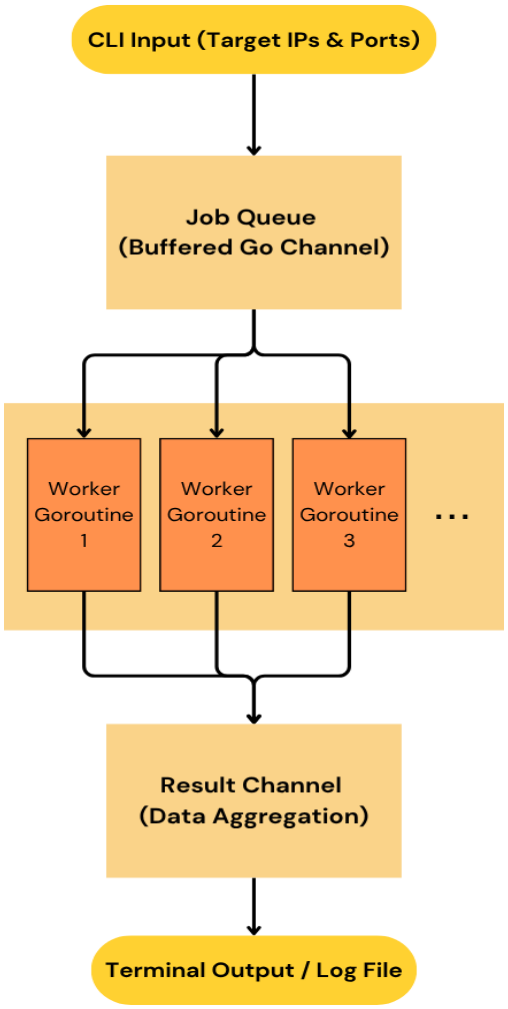


*Fig. 1. System Architecture detailing the fan-out/fan-in concurrency model utilizing Go channels and worker pools.*

## 1. Step-by-Step Validation Flow

Figure 2 breaks down the exact decision-making process that occurs inside a single worker when it receives a port assignment from the queue.

The first action the worker takes is attempting to establish a standard TCP connection (a three-way handshake) with the target port. To ensure accuracy, we implemented a retry mechanism. If the initial connection fails or times out, the worker will automatically retry. If the connection fails twice consecutively, the system officially designates the port as "Closed or Filtered" and terminates the process for that specific port to conserve resources. By doing this we are cutting some time overhead, while improving the throughput and efficiency.

```
func worker(ports <-chan int, results chan<- ScanResult, wg
        *sync.WaitGroup, target string, scannedCount *uint32,
      timeoutMs int) {
    defer wg.Done()
    timeout := time.Duration(timeoutMs) * time.Millisecond

    for p := range ports {
        atomic.AddUint32(scannedCou
        nt, 1)
        address := fmt.Sprintf("%s:%d", target, p)

        // Attempt TCP connection with 2 retries
        if !isPortOpen(address, timeout, 2) {
            continue
        }

        // If open, grab the service banner
        banner := grabBanner(address, p, timeoutMs)
        results <- ScanResult{Port: p, Service: banner}
    }
}
```

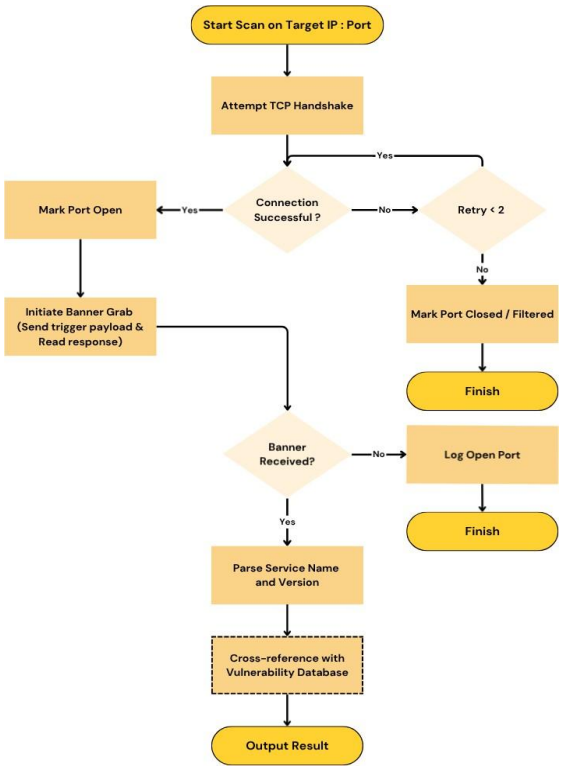


*Fig. 2. Logical flowchart of port validation, connection retries, active banner grabbing, and the vulnerability mapping process.*

Conversely, if the connection is successful, the worker registers the port as "Open" and immediately transitions to the critical phase: Banner Grabbing. Banner Grabbing is the process of interrogating the service running on the open port to reveal its identity. The worker sends a specific payload such as an HTTP HEAD request for web ports and waits for the server's text response. Once the raw buffer is received, the sys-tem parses the text clean by stripping out invisible characters and carriage returns,

leaving only the clean service name and version number. In the future scope of this project, this clean and structured version string will be programmatically cross-referenced with global CVE (Common Vulnerabilities and Exposures) databases to identify known security flaws. The Vulnerabilities which are detected will be given as output with the respective port number, Operating Systems(OS) version and the service it is currently running, In a concise json format.

## 2. Core Code Implementation

The implementation relies heavily on Go's standard 'net' package for socket connections and the 'sync' and 'atomic' packages for safe data handling.

Listing 1. Core logic for parallel worker execution, retry logic, and active service identification.

The function above continuously loops over the 'ports' channel until the channel is closed. We utilize 'atomic.AddUint32' to safely increment a shared progress counter. In highly concurrent applications, if hundreds of workers attempt to update a standard integer variable simultaneously, it creates a "race condition" that crashes the program. The atomic package guarantees that the progress counter is updated safely at the processor level.

## Experimental Setup and Results

To scientifically validate the efficiency of our Go-based architecture, we conducted three distinct benchmark experi-ments. We measured how the tool handles varying workload sizes, how the total number of concurrent workers impacts overall speed, and how the tool compares against the industry standard.

### 1. Experimental Environment

All benchmarks were conducted within a localized Kali Linux environment, a specialized Debian-based operating sys-tem built for penetration testing and network security auditing. Running the tests on Kali Linux ensures that the network stack is optimized for offensive security tools.

It is vital to note that operating systems have strict limits on how many network connections (file descriptors) a single program can open simultaneously to prevent denial-of-service attacks against the host machine itself. Because our tool is capable of launching thousands of connections at once, we temporarily increased the Linux file descriptor limit using the command 'ulimit -n 10000' prior to running the experiments. This allowed the Go runtime to operate without being artifi-cially throttled by the operating system kernel.

### 2. Experiment 1: Sequential vs. Concurrent Execution

In our first experiment, we sought to quantify the exact time saved by utilizing Go's concurrency model instead of a traditional step-by-step approach. We scanned local ports ranging from 100 up to 5,000, recording the execution time for both models.

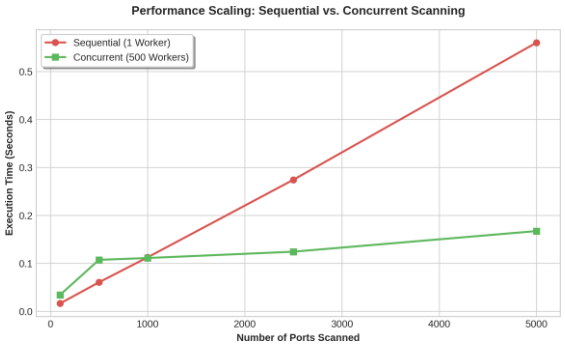


*Fig. 3. Performance Scaling: Execution times of single-threaded vs. con-current (500 workers) scanning.*

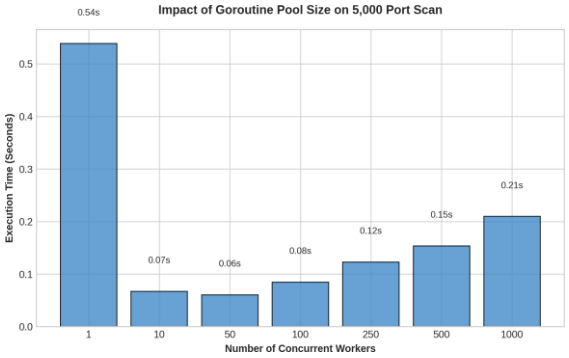


*Fig. 4. Impact of Goroutine Pool Size: Time required to scan 5,000 ports showing diminishing returns.*

Figure 3 clearly illustrates the massive mathematical ad-vantage of parallel scanning. The red line represents a single sequential worker checking ports one by one. As the quantity of target ports increases, the execution time rises sharply in a linear trajectory, eventually requiring nearly 0.6 seconds to scan 5,000 ports locally.

Conversely, the green line represents our proposed tool operating with a pool of 500 workers. While there is a fractional initial overhead required to instantiate the goroutines and channels (making it slightly slower for the very first 100 ports), the execution time line remains almost entirely horizontal as we add thousands of ports to the workload. The concurrent model successfully completed the 5,000-port scan in less than 0.2 seconds. This proves that the architecture can absorb massive workloads with negligible impact on

total execution time.

### 3. Experiment 2: Analyzing Worker Efficiency and Bottlenecks

For the second experiment, we kept the workload statically fixed at 5,000 ports and manipulated the number of workers assigned to the job. The objective was to discover the optimal pool size and identify where system-level bottlenecks occur.

As shown in Figure 4, scaling the pool from 1 worker to

10 workers reduces the scan time drastically, as expected. However, the most critical data point occurs at 50 workers, which achieved the absolute fastest execution time of 0.06 seconds.

Surprisingly, when we aggressively increased the worker pool to 500 or 1,000, the scan time actually deteriorated, rising back up to 0.21 seconds. This phenomenon perfectly demonstrates the concept of "context switching overhead." When there are too many concurrent threads fighting for a limited number of network sockets and CPU cycles, the host operating system spends more computing power managing and scheduling the workers than it does actually sending network packets. This data proves that a carefully balanced worker pool is far superior to simply allocating the maximum possible threads to a problem.

### 4. Experiment 3: Real-World Comparison against Nmap

Finally, we subjected our tool to a rigorous real-world test by running it head-to-head against Nmap. We targeted a remote server over a Wide Area Network (WAN), specifically the public testing server 'scanme.nmap.org'. To ensure a sci-entifically fair, apples-to-apples comparison, we forced Nmap to use a standard full TCP Connect scan ('-sT'), disabled ping sweeps ('-Pn'), and disabled DNS resolution ('-n').

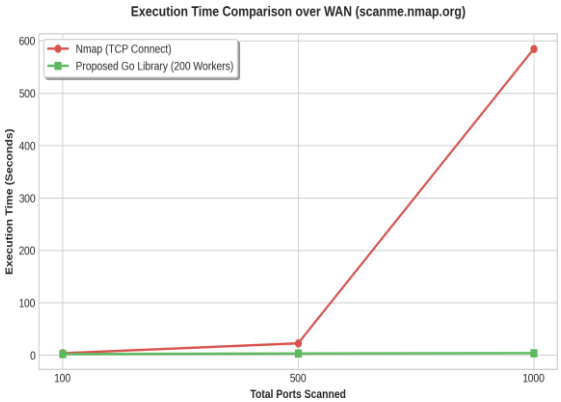


*Fig. 5. Wide Area Network (WAN) Comparison*

Execution time of the proposed Go Library versus Nmap performing a standard TCP Connect scan over the internet.

Figure 5 displays a dramatic disparity in execution speeds. Over a live internet connection, network latency and packet routing become significant factors. Nmap scales upward in-credibly slowly, requiring nearly 600 seconds (10 minutes) to scan just 1,000 ports using full TCP connections.

In stark contrast, our Go library powered through the identical 1,000 ports in just a few seconds. It is important to note that Nmap incorporates highly sophisticated, dynamic timing algorithms designed to slow down its scan rate to avoid crashing remote firewalls or triggering Intrusion Detection Systems (IDS). Our library currently lacks these dynamic rate-limiting features and prioritizes raw, aggressive speed.

This proves that for internal network assessments, cloud-native auditing, or authorized engagements where stealth is not required, our lightweight Go library provides significantly faster raw TCP connection turnaround times than traditional methodologies.

### Future Scope and Advancements

While the core concurrent scanning engine is highly func-tional, there are several advanced features planned for the next iterations of this library.

The primary upcoming feature is the full automation of the vulnerability mapping phase (as denoted previously in Figure 2). By taking the sanitized "Service Name and Version" strings gathered during the banner grabbing phase, the tool will automatically query the National Vulnerability Database (NVD) via REST APIs. This will upgrade the tool from a mere port scanner to an active vulnerability analysis engine, imme-diately highlighting known CVEs associated with discovered software.

Additionally, expanding protocol support to include User Datagram Protocol (UDP) scanning, and implementing raw socket manipulation to allow for stealthy SYN (half-open) scanning, will grant the library parity with enterprise-grade network scanners.

### Conclusion

This paper successfully demonstrates the design, implemen-tation, and rigorous testing of a highly efficient, concurrent network scanning tool developed in the Go programming language. Through careful architectural planning and the strategic utilization of buffered channels and goroutines, we effectively eliminated the

networking bottlenecks commonly found in traditional, sequential scanning programs. Our em-pirical benchmarks, conducted in a Kali Linux environment, proved that the proposed tool drastically reduces scan times on local networks and vastly outperforms standard TCP connect scans over public wide area networks. Ultimately, this scalable library provides software developers and security engineers with a simple, natively embeddable, and exceptionally fast solution for automating network footprinting and identifying perimeter vulnerabilities.

**References**

G. Lyon, “Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning,” Insecure.Com LLC, 2009.

A. Donovan and B. Kernighan, “The Go Programming Language,” Addison-Wesley Professional, 2015.

R. Graham, “Masscan: The entire Internet in 3 minutes,” GitHub Repository. [Online]. Available: https://github.com/robertdavidgraham/masscan.

MITRE Corporation, “Common Vulnerabilities and Exposures (CVE),” [Online]. Available: https://cve.mitre.org/.

Z. Durumeric et al., “ZMap: Fast Internet-wide Scanning and Its Security Applications,” 22nd USENIX Security Symposium, 2013. IEEE, “IEEE Paper Format Guidelines,” 2024.