Grenze International Journal of Engineering and Technology, June Issue

Visualizing Dependencies in Go

Gajendra Singh Rajput[1], Mubeen Ahmed Khan[2], Kailash Kumar Baraskar[3] and Ajaj Khan[4]
[1-4]Department of Computer Science and Engineering, Medi-Caps University, Indore, Madhya Pradesh, India
Email: gajendrasingh.rajput26@gmail.com, makkhan0786@gmail.com, kailashkumar.baraskar@medicaps.ac.in, ajaj.khan@medicaps.ac.in

***Abstract*—Today in the world of the software industry, the use of libraries and frameworks has become very common to abstract complex implementations in modern software. This gave rise to a new challenge of tracking and monitoring external dependencies due to various reasons like security, company/team design philosophy or both. In this paper, we delve into the tools provided by the Go language to solve this problem and some of its limitations. Based on those learning, we have created a library for visualizing internal and external dependencies for Go based projects. This library generates a tree-like structure to give users a better visual representation about how dependencies are used across projects. We also delve into how popular tools and frameworks by different organizations and individuals make software design choices around the use of external libraries or dependencies.**

***Index Terms*— Go, Go Modules, Dependency, Dependency.**

I. INTRODUCTION

Visualization discovered, the stakeholders will be to pin down the exact package and take precautionary actions.

Apart from simplify, Go also offers a wide variety of packages to build software applications and encourages developers to avoid the use of external dependencies altogether. As software gets more complex, developers are increasingly using pre-written code libraries (third-party libraries) to build applications.[1]. There's a hidden danger in relying on pre-written code (software dependencies). While it's become popular to quickly reuse these building blocks. We haven't figured out the best way to choose them wisely, use them effectively, or even know when they're truly helpful versus when they might cause problems.[2].The way dependencies have been used and managed has changed over time. Until the introduction of Go modules in Go 1.11, the dependency ecosystem in Go revolved around GOPATH and the vendor directory, where GOPATH is an environment variable that defines the project workspace. For stable and reproducible builds, modules record precise dependency requirements[3] along with a checksum of files to verify authenticity. It is crucial for stakeholders (developers and project managers) to understand how their codebase uses dependencies and how their dependencies use other dependencies(transitive). In case some vulnerability is In this paper, we aim to demystify the dependency visualization provided by the language, with commands like go list and go mod. The difficulties one may confront while attempting to comprehend them. Based on this study, we created a tool that helps stakeholders better comprehend the same connections through a tree-like structure.

II. LITERATURE REVIEW

Go list and go mod why are two tools provided by the language that can be used to better understand and manage

Grenze ID: 01.GIJET.10.2.544_1

the dependencies of a project. While go list is used to list the named packages, one per line[4]. It gives a simpler view of which dependencies are used to build the application. On the other hand, go mod why shows the shortest path in the import graph from the main module to each of the listed packages[5]. The go list command is limited to the dependencies our project uses. It doesn't provide the context for transitive dependencies. The go mod why command on the other hand provides the context of where the dependencies are needed. This does help us track down packages which use a specific dependency. In Go, code is organized into packages, which are further grouped into modules. A module defines the dependencies required for your code to function correctly.[6]. These code files are organized in a hierarchical tree structure. Both the tools lack the context of how our dependencies are distributed across the tree.

III. Problem Statement

1. **Visualization with respect to hierarchical file structure**: Tracing of a file which uses a dependency should be as easy as finding that very file in a hierarchical file. For external dependencies, go mod why is quite helpful, but for internally dependencies it lacks the overall context.
2. **Potential supply chain attacks**: If a project imports dependencies, which has its own set of dependencies and one of the dependencies has some malicious code injected into it. This could potentially lead to a supply chain attack, where our code indirectly imports a package with malicious intent.

IV. Proposed Solution

Go uses a directed acyclic graph to map out its dependencies. This data structure can also be represented as a tree with some repeated nodes. This representation of packages which be similar to the file system that is already used by developers.

A. Proposed Algorithm

Leveraging the inherent hierarchical nature of files, we can efficiently identify dependencies using a depth-first search (DFS) traversal. This approach ensures we explore all dependencies within a specific file before moving on to the next. To further optimize the process, we'll incorporate memorization. By storing the resolved dependencies, we eliminate redundant computations and significantly improve performance.

Algorithm:

1. **START**
2. Get all the imports of the current package in []Deps
3. If len(Deps) == 0, then:
    - Print the current package name.
    - Go to **Step 5**

   Else,
   Start a loop for Deps[i] < len(Deps):
    - Add current package to a map.
    - Go to **Step 2**

STOP

Limitation: The algorithm for resolving the version is based on the path where the dependency is stored. In a non-vendor project, the dependency is stored with a version suffix, like github.com/user/repo@x.y.z, in the GOPATH/src directory. So the version is resolved using the suffixed import path. But in case of vendor(ed) projects, the modules are simply downloaded/copied in vendor directory as github.com/user/repo and the version information is written in a vendor/module.txt file. Even if the exact same dependency with the exact same

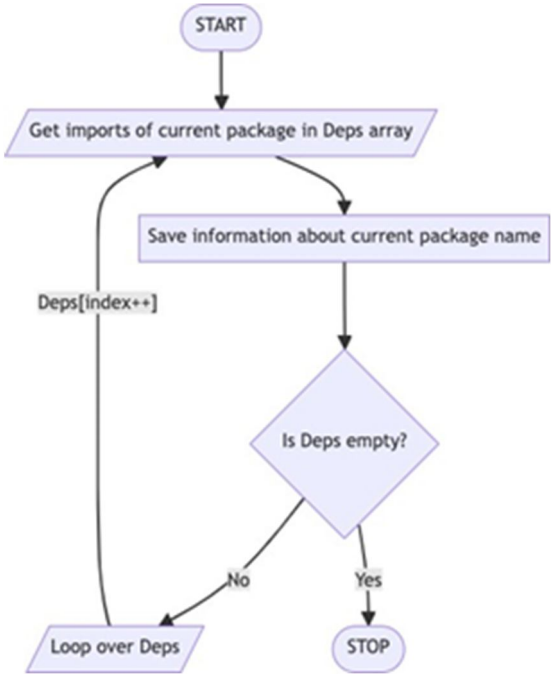

Fig 1: Flowchart of Algorithm

version is in GOPATH/src directory. The first preferred path to get the source code of dependency will be to get it from the vendor directory.

B. Study Dataset

As a test to investigate the usage of external dependencies, we choose a few open-source projects. These projects were primarily sourced based on the researchers' experience with Go programming language and their familiarity with the organization or user and their impact.

V. Results and Conclusion

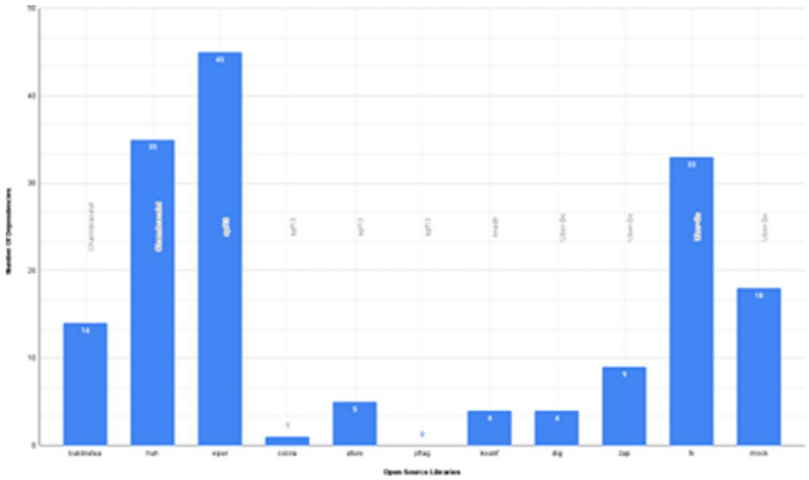

Fig 2: Open-source libraries and their external dependency count

The graphs show in the above pictures shows various open-source libraries (Fig 2.) and developer tools (Fig 3) from different organizations and developers. The Y axis represents the number of external dependencies used, while the X-axis representing the respective projects.

In Fig. 2, the maximum number of dependencies used is 45, while minimum being 0. On the other hand, in Fig 3, the maximum is 652 and minimum being 15. A general view of both, the graph represents how software

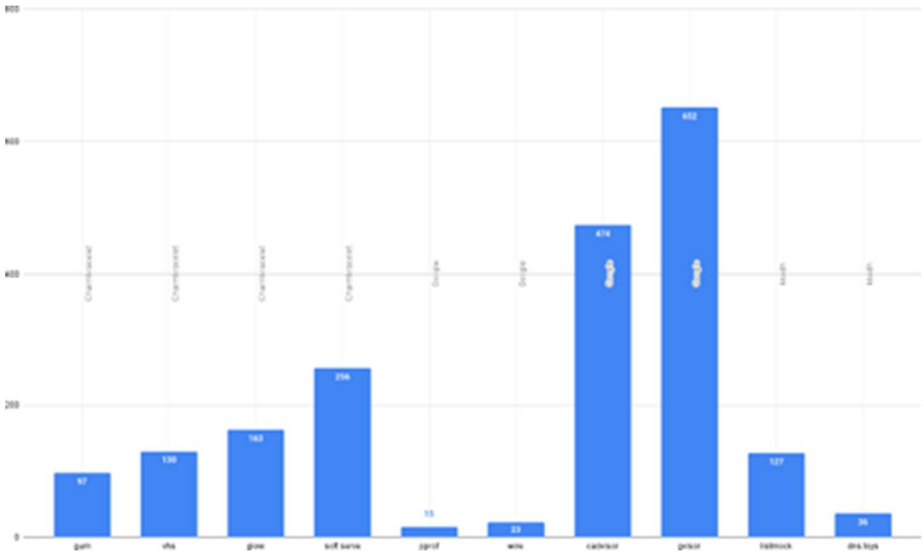

Fig 3: Open-source tools and their external dependency count

programs and libraries are developed. Software is complicated in nature, requires libraries to abstract out complicated implementations, and hence we see a large count of external dependencies. On the other hand, Libraries have to support more than just one project or software program while ensuring that the dependency tree generate is zero to minimum to avoid transitive upgrades and supply chain attacks.

VI. Future Work

Despite having the metadata of vendored dependencies, our core library currently struggles to resolve the specific versions of dependencies included in a project's vendor directory. A hacky workaround would have been to delete the vendor directory and download those vendored directories. But this increases the resolution time while disturbing the project structure. Ideally, in future iterations, a solution could be engineered to parse the vendor/module.txt file. This would allow us to accurately determine version information without modifying the project structure.

While understanding dependencies is valuable, incorporating a security perspective would make a much more meaningful impact on how libraries are used. Google's Open-Source Insights project, maps dependencies of open-source projects alongside their vulnerability scores, this feature would provide crucial security context. But Google's project is limited to open-sourced projects only. Therefore, our tool/library could be used to generate the dependency tree and link to Google's vulnerability score.

Despite the limited information provided by go mod graph command, it is relatively fast compared to our implementation, which is single threaded. A possible implementation of it could be to concurrently resolve the nodes at the same level and use map and mutex to lock the current dependencies being resolved, so that the other concurrent thread doesn't resolve it again.

References

[1] J. Hu, L. Zhang, C. Li, Y. Shao-Horn, S. Huang, and Y. Liu, "Empirical analysis of vulnerabilities Life cycle in Golang Ecosystem," arXiv (Cornell University), Dec. 2023, doi: 10.1145/3597503.3639230.

[2] R. Cox, "research!rsc: Our Software Dependency Problem." https://research.swtch.com/deps

[3] O. Ehrenberger, "Golang Environment – GOPATH vs go.mod," freeCodeCamp.org, May 26, 2023. https://www.freecodecamp.org/news/golang-environment-gopath-vs-go-mod

[4] Golang, "go/src/cmd/go/internal/list/list.go at master·golang/go," GitHub. https://github.com/golang/go/blob/master/src/cmd/go/int ernal/list/list.go

[5] Golang, "go/src/cmd/go/internal/modcmd/why.go at master·golang/go," GitHub. https://github.com/golang/go/blob/master/src/cmd/go/int ernal/modcmd/why.go

[6] "Tutorial: Create a Go module - The Go Programming Language.", https://go.dev/doc/tutorial/create-module.