HERO: On the Chaos When PATH Meets Modules

Ying Wang*, Liang Qiao*, Chang Xu†§, Yepang Liu†, Shing-Chi Cheung‡, Na Meng¶,
Hai Yu‡, and Zhiliang Zhu*

* Software College, Northeastern University, China
† State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China
‡ Southern University of Science and Technology, China
¶ The Hong Kong University of Science and Technology, China
§ Software College, Northeastern University, China

Email: wangying@swc.neu.edu.cn, qiaoliangneu@163.com, {yuhai, zzl}@mail.neu.edu.cn
† State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China, Email: changxu@nuj.edu.cn
‡ Southern University of Science and Technology, China, Email: liuyip@sustech.edu.cn
¶ The Hong Kong University of Science and Technology, China, Email: scc@cse.ust.hk

Abstract—Ever since its first release in 2009, the Go programming language (Golang) has been well received by software communities. A major reason for its success is the powerful support of library-based development, where a Golang project can be conveniently built on top of other projects by referencing them as libraries. As Golang evolves, it recommends the use of a new library-referencing mode to overcome the limitations of the original one. While these two library modes are incompatible, both are supported by the Golang ecosystem. The heterogeneous use of library-referencing modes across Golang projects has caused numerous dependency management (DM) issues, incurring reference inconsistencies and even build failures. Motivated by the problem, we conducted an empirical study to characterize the DM issues, understand their root causes, and examine their fixing solutions. Based on our findings, we developed HERO, an automated technique to detect DM issues and suggest proper fixing solutions. We applied HERO to 19,000 popular Golang projects. The results showed that HERO achieved a high detection rate of 98.5% on a DM issue benchmark and found 2,422 new DM issues in 2,356 popular Golang projects. We reported 280 issues, among which 181 (64.6%) have been confirmed, and 160 of them (88.4%) have been fixed or are under fixing. Almost all the fixes have adopted our fixing suggestions.

Index Terms—Golang Ecosystem, Dependency Management

I. INTRODUCTION

The Go programming language (Golang) is quickly adopted by software practitioners since its release in 2009 [1]. Like other modern languages, Golang allows a project to import and reuse functionalities from another Golang project (i.e., library) by simply specifying an import path [2]. There are four popular sites hosting Golang projects, namely, Bitbucket [3], GitHub [4], Launchpad [5], and IBM DevOps Services [6]. Among them, GitHub hosts nearly 90% Golang projects (as of June 2020) [7].

While Golang’s library-based development boosts its adoption, its library-referencing mode has undergone a major change as the language evolves. Prior to Golang 1.11, library-referencing was supported by the GOPATH mode. Libraries referenced by a project are fetched using command go get [8]. This mode does not require developers to provide any configuration file. It works by matching the URLs of the site hosting referenced libraries with the import paths specified by the go get command. However, it fetches only a library’s latest version. To overcome this restriction, developers use third-party tools such as Dep [9] and Glide [10] to manage different library versions under the Vendor directory. To satisfy developers’ need for referencing specific library versions, in August 2018, Golang 1.11 introduced the Go Modules mode, which allows multiple library versions to be referenced by a module using different paths. A module comprises a tree of Golang source files with a go.mod configuration file defined in the tree’s root directory. The configuration file explicitly specifies the module’s dependencies with specific library versions as well as a module path by which the module itself can be uniquely referenced by other projects. The file must be specified according to the semantic import versioning (SIV) rules [11]. For instance, projects whose major versions are v2 or above should include a version suffix like “v/2” at the end of their module paths.

Compared with GOPATH, Go Modules is flexible and allows multiple library versions to coexist in a Golang project [12]. Developers are suggested to migrate their projects’ library-referencing modes from GOPATH to Go Modules. However, the migration took a long time. We sampled 20,000 popular Golang projects on GitHub. As of June 2020, only 35.9% projects had migrated to Go Modules, while the rest 64.1% were still using GOPATH, resulting in the coexistence of two different library-referencing modes. What’s more, many projects suffered from various dependency management (DM) issues caused by such mixed library-referencing modes. Specifically, we made the following three observations:

• Go Modules is not backward compatible with GOPATH.

There are two scenarios. First, a Golang project can be referenced by its downstream projects. After it migrates to Go Modules, its introduced virtual import paths (with version suffixes) cannot be recognized by downstream projects still in GOPATH. This causes build errors to these projects. Second, a downstream project that has migrated to Go Modules may not find its referenced libraries in GOPATH, or may fetch unintended library versions, due to different import path

†The Vendor attribute allows a Golang project to reference a library’s different versions and keep them in different folders under a vendor directory.
interpretations by the two modes.

- **SIV rules can be violated even if a Golang project and its referenced upstream projects both use Go Modules.** For instance, a project of major version v2 may not necessarily include a version suffix at the end of its module path. Such violation can be due to developers’ misunderstanding or weak SIV rule enforcement (discussed later in Sec II-A). They can cause a large number of unresolved library references (“cannot find package” errors) when downstream projects are built.

- **Resolving DM issues for a Golang project requires up-to-date knowledge of its upstream and downstream projects, as well as their possible heterogeneous uses of two library-referencing modes.** However, such information is not provided by the Golang ecosystem to help developers evaluate a solution’s impact on other projects. Resolving a DM issue in a project locally without considering the ecosystem in a holistic way can easily cause new issues to its downstream projects.

Figure 1(a) shows a DM issue example. Project liz4 [13] migrated to Go Modules in version v2.0.7. Following SIV rules, it declared module path `github.com/pierrec/lz4/v2` in its `go.mod` file with version suffix `/v2`. Although the project can be built successfully after migration, it induced DM issues to downstream projects still in GOPATH, since the latter cannot recognize the version suffix in module paths (e.g., issue #530 of `filebrowser` [14]). To fix the problem, liz4 released version v2.2.4, which was still in Go Modules but removed version suffix `/v2` from its module path as a workaround. This resolved the DM issues in its downstream projects in GOPATH, but induced build errors into its downstream projects that had already migrated to Go Modules, since this solution violated SIV rules (e.g., issue #39 of liz4 [15]). As there is no accurate way to estimate the migration impact to its downstream projects, liz4 chose to roll back to GOPATH in v2.2.6 and suspended its migration until its most downstream projects had completed migrations. Such problems are common across Golang projects, imposing unforeseeable risks in mode migration.

Figure 1(b) shows another example from `go-i18n` [16]. Its version v2.0.1 followed its upstream projects to use Go Modules for finer library-referencing control. However, such change induced at least five DM issues to downstream projects in GOPATH (e.g., issue #184 [17]) due to their inability to interpret version suffix `/v2` in `go-i18n`’s module path. To fix the problem, `go-i18n` v2.0.2’s repository provided an additional subdirectory `go-i18n/v2` with a copy of implementations to support downstream projects in GOPATH. This is a suboptimal solution since it changes the virtual path in Go Modules into a physical one that requires extra maintenance in every project release. In fact, without a holistic view of all dependencies and the interference between their mixed library-referencing modes, it is hard for developers to find a proper solution to fix DM issues without impacting downstream projects.

Such chaos caused by mixed library-referencing modes is unique to Golang ecosystem, unlike existing dependency conflict issues in Java [18]–[20], JavaScript [21] and Python projects [22]. Besides, our study of 20,000 Golang projects on GitHub suggests the severity of DM issues caused by the mode migration, since a majority of these projects have chosen to stay with old GOPATH. To better dig into the problem, we sampled 500 projects from top 1,000 ones, and collected 151 DM issues from the issue trackers for a deeper study of their characteristics and solutions. We identified three DM issue patterns and summarized eight fixing solutions commonly adopted by developers. Leveraging these findings, we further developed an automated tool, HERO (HEalth diagnosis tool for the Golang ecosystem), to detect DM issues. One interesting feature is that HERO can also provide customized fixing suggestions to developers with analyses of potential benefits and consequences incurring to the ecosystem.

To evaluate HERO, we collected 132 real DM issues from top 1,000 Golang projects that were not used in our empirical study and conducted experiments using these issues as a benchmark. HERO achieved a detection rate of 98.5% (only missed two cases). We further applied HERO to the rest 19,000 projects collected from GitHub, and detected 2,422 new DM issues. We submitted 280 of them that were associated with the 1,001st–2,000th popular projects, and suggested fixing solutions. Encouragingly, 181 (64.6%) issues have been confirmed, and 160 (88.4%) of them have been fixed or under fixing using our suggested solutions. Such fixes would cause minimal or acceptable impacts to other projects in the ecosystem. The confirmed issues cover well-known projects, such as `github/hub` [23] and `microsoft/presidio` [24], and have promoted 29 projects’ migration to Go Modules.

To summarize, in this paper, we made three contributions:

- **Originality.** To our best knowledge, we conducted the first empirical study on 20,000 Golang projects to investigate their status of library-referencing mode migration and analyze 151 real DM issues to unveil their characteristics.

- **Technique.** We developed the HERO tool² to diagnose dependency management issues for the Golang ecosystem. It can detect DM issues effectively and provide customized fixing suggestions.

- **Reproduction package.** We provided a reproduction package on HERO website for future research, which includes: (1) detailed information of the 20k subjects and 151 DM issues studied in our empirical study; (2) our benchmark dataset (132 DM issues and subjects used for evaluation);

²http://www.hero-go.com/
II. BACKGROUND

We introduce SIV rules in Go Modules and the concept of module-awareness, to facilitate our later discussions.

A. SIV Rules in Go Modules

Go Modules introduces SIV to support dependency management of multiple project versions. It has three rules:

1) Golang projects should follow a semantic versioning format (Semver). Figure 2(a) gives an example, where projectA tags a release with a semantic version of v2.7.0 on GitHub.

2) When a project’s major version is v2 or above (denoted as v2+), a version suffix like “/2” must be included at the end of its module path declared in the go.mod file. As shown in Figure 2(b), projectA v2.7.0’s module path is “github.com/user/projectA/v2”. To reference it, downstream projects must declare this path and import it in require directive attributes of the go.mod file, as well as in import directive attributes of their .go source files. Figures 2(c) and (d) give two examples.

3) If a project’s major version is v0/v1, its version suffix should not be included in its module or import paths.

Under these SIV rules in Go Modules, multiple major versions of a library can be separately referenced by different paths. In contrast, a project in GOPATH can reference only the latest version of a library.

To be more flexible, the official Golang documentation [11] suggests two strategies to release a v2+ project, namely, major branch and major subdirectory. The former is to update a project’s module and import paths to include a version suffix like “/2”. It is not necessary to physically create a new branch labeled with such a version suffix on the version control system of hosting site. The latter is to physically create a subdirectory (e.g., projectA/v2) with source code and a corresponding go.mod file, and the corresponding module path must end with a version suffix like “/2” accordingly.

As such, module and import paths in the major branch strategy are virtual, but are physical in the major subdirectory strategy. The latter is sometimes used to provide a transition for downstream projects in GOPATH, as shown in Figure 1(b).

B. Module-Awareness in Different Golang Versions

To ease discussion, we refer to the capability of recognizing a virtual path ended with a version suffix like “/2” as module-awareness. This capability is important for referencing libraries in the Golang ecosystem.

As the migration from GOPATH to Go Modules has immense impact on many Golang projects, it was gradually achieved by multiple Golang versions over two years. During migration, “minimal module compatibility” was adopted since Golang 1.9.7 in the series 1.9.* and Golang 1.10.3 in the series 1.10.*, which added module-awareness to projects that had not migrated to Go Modules [25]. As such, we refer to the versions in range [1.0.1, 1.9.7] ∪ [1.10.1, 1.10.3), which manage dependencies in GOPATH without module-awareness, as legacy Go modules. We refer to those in range [1.9.7, 1.10.1] ∪ [1.10.3, 1.11.1], which manage dependencies in GOPATH with module-awareness, as compatible Golang versions. We refer to those of 1.11.1 or above, which allows projects to adopt either GOPATH or Go Modules and support module-awareness, as new Golang versions. We observe that Golang projects in GOPATH often use third-party tools (e.g., Dep [9], Glide [10], etc.) to help manage dependencies. Since none of the tools supports “minimal module compatibility”, their uses actually block module-awareness, messing up library-referencing (e.g., issue #878 of olivere [26] about using Dep and glide, and #103 of migrate [27] about using govendor).

Table 1 summarizes module-awareness in different Golang versions. Based on this, we give two definitions below:

**Definition 1 (Module-aware project):** A project is module-aware if and only if it uses a compatible or new Golang version and does not use any DM tool.

**Definition 2 (Module-unaware project):** A project is module-unaware if and only if it uses a legacy Golang version, or it uses a compatible or new Golang version with a DM tool.

Figure 3 shows how module-aware and module-unaware projects differ in parsing an import path with or without a v2+ version suffix. For an import path like github.com/user/projectA, a module-aware project could reference a specific version v0.7.* or v1.2.* of projectA under v2 (latest version under v2, by default), while a module-unaware project would reference the version on projectA’s main branch (typically the latest version). For an import path like github.com/user/projectA/v2, the former could reference a specific version v2.1.* of projectA (latest version under v2, by default), while the latter would refer to the version of projectA's downstream project.
by default), while the latter would fail to recognize it.

According to the above background knowledge, we formally define the DM issues occurred in Golang projects as follows:

**Definition 3 (Dependency management (DM) issue):** If an issue is caused by the different interpretations between module-aware and module-unaware projects or violating SIV rules by Go Modules projects, we refer to it as a DM issue in Golang ecosystem.

A project suffers from a DM issue may fetch the unintended versions of its libraries, or may not find its referenced libraries.

### III. Empirical Study

We empirically study the characteristics of DM issues and the scale of these issues arising from the varying degrees of module-awareness in different Golang versions. We aim to answer the following three research questions:

- **RQ1 (Scale of Module-Awareness):** What is the status quo of library-referring mode migration for projects in the Golang ecosystem? To what extent are they module-aware?
- **RQ2 (Issue Types and Causes):** What are common types of DM issues? What are their root causes?
- **RQ3 (Fixing Solutions):** What are common practices for fixing DM issues? How do they affect the ecosystem?

To answer RQ1, we collected top 20,000 popular and active open-source Golang projects from GitHub to study their migration status. To answer RQ2/3, we randomly selected 500 subjects (denoted as subjectSet1) from top 1,000 of our collected projects. We then collected real DM issues from these projects plus some additional ones. To dig into these issues, we manually analyzed their issue descriptions, developers’ discussions, code commits, and the Golang official documentation. Note that the rest 500 projects (denoted as subjectSet2) in top 1,000 of our collected projects were not used in RQ2/3. They are used to evaluate our DM issue detection technique later (Sec V-A). Below we present our data collection procedure and study results in detail.

#### A. Data Collection

**Step 1: Collecting Golang projects.** We collected top 20,000 popular and active Golang projects from GitHub, which hosts over 90% Golang ones. A project’s popularity is decided by its star counts, and activeness is decided based on whether 50+ code commits exist in its repository since Jan 2020.

Figure 4 shows these projects’ demographics. They are: (1) popular (60.3% having 100+ stars or forks), (2) well-maintained (on average having 339 code commits and 136 issues), and (3) large-sized (on average having 72.3 KLOC). We used these projects for RQ1.

**Step 2: Collecting DM issues.** For the 500 projects in subjectSet1, after filtering the ones that have no issue trackers or code repositories, we considered the remaining 484 projects as subjects. We then added to the seed subjects Golang’s official project go-lang/go [28] and two most popular dependency management tools Dep [9] and Glide [10], for better studying DM issues from the perspective of the ecosystem. In total, we obtained 487 projects for RQ2/3.

As these projects contain many issue reports, we filtered using keywords “go modules” and “go.mod” (case insensitive) to locate potential DM issues for manual analysis (”go.mod” configuration file is a notable new feature in the Go Modules mode). Keyword “go modules” returned 1,342 issue reports, and “go.mod” returned 2,421 ones. We merged overlapping reports and then removed noise. First, we excluded issue reports that did not discuss DM issues (e.g., issue #5559 [29] of project gogs [30] only documented developers’ plan to migrate to Go Modules). Second, we excluded issue reports that discussed nothing about root causes of DM issues.

Three co-authors cross-checked all collected issue reports, and finally obtained a collection of 151 well-documented DM issues, which involves 127 Golang projects. They contain sufficient details for studying RQ2/3.

#### B. RQ1: Scale of Module-Awareness

We analyze the scale of module-awareness as below:

- For all 20,000 projects, we counted the number of projects that have migrated to Go Modules by checking whether go.mod files exist in their latest versions’ repositories.
- For projects that have migrated to Go Modules, we checked whether their major version numbers of latest releases are v2+. If so, we further checked their adopted strategies (i.e., major branch/subdirectory) in the code repositories.
- For projects still in GOPATH, we checked whether they use third-party tools to manage dependencies by the presence of their configuration files. For example, using the Dep [9] or Glide [10] tool requires a Gopkg.toml or glide.yaml configuration file, respectively.
Results. Figure 5 shows analysis results. To see trends, we divided all projects into six (overlapping) groups based on their popularities: top 500, 1k, 5k, 10k, and 20k (1k = 1,000).

From Figure 5(a), we see that the proportion of Go modules migrations increases with the popularity of projects. This suggests that migrating to Go modules is a good practice in the ecosystem. Still, 64.1% projects are in GOPATH despite that two years have passed since Go modules came into being.

Figures 5(b) and 5(c) show that only 4.5% projects that have migrated to Go modules released v2+ versions (i.e., most ones are still in v0/v1), and 91.2% v2+ versions were managed by the major branch strategy. This suggests that the vast majority of v2+ projects should be referenced by virtual module paths ended with version suffixes like "v2+". Then they are likely to induce build failures in module-unaware downstream projects. Besides, for the rest 95.5% projects whose major versions are v0/v1, DM issues can easily occur when they are updated to v2+ versions in future.

Figure 4(d) shows that 79.6% projects in GOPATH use third-party tools to manage dependencies. As aforementioned, this will block module-awareness for projects that adopt compatible Golang versions. Therefore, at least 10,205 of top 20,000 Golang projects (51.0%) are module-unaware.

Challenges of migration. Our findings may explain why many Golang projects stay with GOPATH. We also investigate how developers consider this problem from projects still in GOPATH. We focused on the GOPATH part (36.6%) of top 500 out of the 20,000 projects (Fig. 5(a)), and analyzed their issue reports that discuss migration to study reasons for holding the migration. We obtained 52 issue reports specifically discussing unsuccessful migration, and observed three common reasons:

- **Existing versioning scheme incompatible with SIV rules in Go modules (27/52).** Some projects have their own versioning schemes, different from SIV rules in Go modules. To avoid incompatibility (e.g., issue #328 of go-tools [31]), developers chose to stay with GOPATH.

- **Third-party DM tools hindering the migration plan (15/52).** Some projects heavily rely on third-party tools for dependency management. As the tools do not work with Go modules, developers chose to live with the tools instead of migration (e.g., issue #61 [32] of uuid).

- **Causing problems to downstream projects in GOPATH (10/52).** Many projects are still in GOPATH, inconvenient to reference upstream projects in Go modules. For continuous support for downstream projects, developers chose to stay with GOPATH (e.g., issue #103 [27] of migrate).

Due to these challenges, we conjecture that GOPATH and Go modules can co-exist for a long time. This suggests the inevitability of DM issues in the Golang ecosystem and motivates us to study their characteristics and fixing solutions.

Answer to RQ1: Golang projects face challenges in migrating to Go modules. Up till June 2020, only 35.9% of top 20,000 projects on GitHub have migrated to Go modules, and at least 51.0% of top 20,000 projects are module-unaware. The two library-referencing modes may co-exist for a long time in the ecosystem.

C. RQ2: Issue Types and Root Causes

We observed three common types of DM issues in collected issue reports. Below we introduce them and analyze their root causes with examples.

**Type A. DM issues can occur when projects in GOPATH depend on projects in Go modules (41/151=27.2%).** The former are typically module-unaware. Build errors can occur when such projects directly or transitively depend on the latter but cannot recognize their virtual paths with version suffixes, e.g., issue #1017 of glide [33].

Among 41 Type A issues, 35 occurred in module-unaware projects when they upgraded upstream dependencies whose newer versions introduced virtual import paths. This shows that version upgrades of libraries in Go modules can impose threats to their module-unaware downstream projects and developers should estimate such threats before upgrading. The rest 6 issues occurred when introducing new upstream projects that transitively depend on virtual import paths.

**Type B. DM issues can occur when projects in Go modules depend on projects in GOPATH (40/151=26.5%).** There are two cases. The first (Type B.1) is due to the different path interpretations between GOPATH and Go modules, and the second (Type B.2) is due to the interference of Vendor attribute in GOPATH.

**Type B.1 (16/40).** Let project $P_A$ in Go modules depend on project $P_B$ in GOPATH, and $P_B$ further depend on $P_C$ in Go modules with import path `github.com/user/PC`. Suppose that $P_C$ has released a v2+ version with the major branch strategy. From $P_B$’s perspective, it interprets the import path as $P_C$’s latest version (i.e., v2+ version on $P_C$’s main branch). However, in $P_A$’s build environment, the import path is interpreted as a v0/v1 version of $P_C$ (no version suffix in the path). As a result, $P_A$ fails to fetch $P_C$’s correct version and can encounter errors when building with $P_B$.

**Type B.1 issues are difficult to notice, and can easily cause build errors.** For example, issue #47246 of cockroach [34] reported that a client project in Go modules depends on cockroach v19.5.2 in GOPATH, and cockroach further depends on project apd [35] in Go modules (with a v2+ version). Although cockroach itself correctly referenced apd v2.0.0 (latest version) by interpreting import path `github.com/cockroachdb/apd`, the client project instead fetched apd v1.1.0 based on its interpretation of this import path. As a result, the client project’s building failed due to missing an important field (not in apd v1.1.0 but in v2.0.0).

**Type B.2 (24/40).** Let project $P_A$ in Go modules depend on project $P_B$ in GOPATH, and $P_B$ further depend on project $P_C$, which is managed in $P_B$’s Vendor directory. A Vendor directory is a major feature of GOPATH, which localizes the maintenance of remote dependencies’ specific versions. We note that $P_A$ references $P_C$ by import path `github.com/user/PC` declared in $P_B$’s source files rather than from $P_B$’s Vendor directory. Although the build may work for the time being, $P_A$ can fail to fetch $P_C$ if $P_C$ is deleted or moved to another repository (e.g., renaming). Even if the fetching is successful, the version on $P_C$’s hosting site could be different from the
one in $P_y$’s Vendor directory, causing potential build errors due to the inconsistency.

Such situations often occur, since there are essentially two versions of a library at two different sites and their consistency is not guaranteed. We witnessed a Type B.2 issue in project moby [36], which has received 57.6k stars on GitHub and ranked the third in popularity. To support its large number of downstream projects still in GOPATH, moby has not migrated to Go Modules. Its issue #39302 [37] reported that moby referenced project logrus [38] from its Vendor directory, and logrus had been relocated from github.com/Sirupsen/logrus to github.com/sirupsen/logrus (case sensitive) on GitHub. This incurred DM issues to many of moby’s downstream projects in Go Modules (e.g., issues #127 of testcontainers [39] and issue #2 of shnorky [40]), as they could not fetch logrus by the import path in moby’s source files.

**Type C. DM issues can occur when projects in Go Modules depend on projects also in Go Modules but not following SIV rules (70/151=46.4%).** We identified three types of SIV rule violations that caused build failures to downstream projects: (1) lacking version suffixes like “/v2” in module paths or import paths, although the versions of concerned projects are v2+ (37/70) (e.g., issue #1355 [41] of iris); (2) version tags not following the MAJOR.MINOR.PATCH format (18/70) (e.g., issue #1848 [42] of gobgp); (3) module paths in go.mod files are inconsistent with URLs associated with concerned projects on their hosting sites (15/70) (e.g., issue #9 [43] of jupyter).

While downstream projects can encounter build failures, the projects violating SIV rules do not produce warnings or errors themselves when building. Currently, there is no diagnosis technique to detect the three SIV rule violation types, or mechanism to enforce SIV rules, as discussed in issues #1355 of iris [41] and #32695 of golang/go [44] by 1z4’s [45] users. As a result, projects violating SIV rules can “safely” stay in the Golang ecosystem, despite the unexpected consequences to their downstream projects. Regarding such risk, 1z4’s developers commented its severity on issue #32695 that “we need to fix this issue and figure out how big the crater it brings to the ecosystem.”

**Answer to RQ2:** DM issues commonly occur due to heterogeneous uses of GOPATH and Go Modules. Their manifestations can be summarized into three types and there are two common root causes: (1) GOPATH and Go Modules interpret import paths in different ways, and (2) SIV rules are not strictly enforced in the Golang ecosystem.

### D. RQ3: Fixing Solutions

Out of the 151 DM issues, 144 issues have fixing patches or fixing plans that developers have agreed on. We studied them and observed eight common fixing solutions, which demonstrate different trade-offs.

**Solution 1: Projects in GOPATH migrate to Go Modules (22/144=15.3%).** Migrating from GOPATH to Go Modules can help fix Type A issues, since these issues are caused by projects still in GOPATH, which are unable to recognize import paths with version suffixes. For example, in issue #454 [46], redis [47] migrated to Go Modules, but its downstream project benthos was still in GOPATH. Then, benthos was suggested to migrate to Go Modules to avoid build errors. This solved benthos’s problem, but caused incompatibility to benthos’s module-unaware downstream projects. As a result, new Type A issues (e.g., issue #232 [48]) arose.

**Solution 2: Projects in Go Modules roll back to GOPATH (13/144=9.0%).** Some projects rolled back to GOPATH after migrating to Go Modules for fixing Types A and C issues. For example, in issue #61 [32] (Type A), project uuid’s [46] migration to Go Modules broke the building of many downstream projects in GOPATH. As a compromise, uuid rolled back to GOPATH, waiting for downstream projects to migrate first. In issue #663 [49] (Type C), gopsutil and its downstream projects were all in Go Modules, but gopsutil violated SIV rules (lacking a version suffix in its module path of v2+ release), causing build errors to downstream projects. As such, gopsutil chose to roll back to GOPATH to make downstream projects work again. This solution solves the problem, but hinders the migration status of the ecosystem.

**Solution 3: Changing the strategy of releasing v2+ projects in Go Modules from major branch to subdirectory (6/144=4.2%).** It helps resolve Type A issues, where module-unaware projects cannot recognize virtual import paths for v2+ libraries in Go Modules. The new strategy creates physical paths by code clone, so that libraries can be referenced by module-unaware projects. However, this just a workaround and needs extra maintenance in subsequent releases (e.g., issue of go-118n [17] as discussed in Sec I).

**Solution 4: Maintaining v2+ libraries in Go Modules in downstream projects’ Vendor directories rather than referencing them by virtual import paths (6/144=4.2%).** Similar to solution 3, this solution also helps resolve Type A issues. By making a copy of libraries in downstream projects’ repositories, it avoids fetching the libraries by virtual import paths. For example, in issue #141 [50], radix’s [51] refused to use the major subdirectory strategy for its v2+ project release in Go Modules. Its downstream projects had to make a copy of radix’s code in their Vendor directories, which requires extra maintenance and potentially cause Type B.2 issues in future.

**Solution 5: Using a replace directive with version information to avoid using import paths in referencing libraries**
(16/144=11.1%). It addresses Types B.1 (problematic import path interpretations) and Type C (import path violating SIV rules) issues. For example, in issue #12 [52], a client project used a directive to replace the original import path: replace github.com/andrewstuart/goq => astuart.co/goq v1.0.0. to reference its expected project goq’s version [53]. However, this would make developers no longer able to use the go get command to automatically fetch upgraded libraries.

Solution 6: Updating import paths for libraries that have changed their repositories (24/144=16.7%). It fixes Type B.2 issues, where libraries in a project’s Vendor directory may be inconsistent with the ones referenced by their import paths. It updates import paths to help a project’s downstream projects in Go Modules fetch consistent library versions. For example, in issue #429 [54], go-cloud managed library etcd in its Vendor directory, etcd later changed its hosting repository from github.com/coreos/etcd to go.etcd.io/etcd. To fix build errors for its downstream projects in Go Modules, go-cloud updated etcd’s import path to the latest one for the consistency. This fixes the issue and benefits all affected downstream projects without impacting others in the ecosystem.

Solution 7: Projects in Go Modules fix configuration items to strictly follow SIV rules (47/144=32.6%). Projects that have migrated to Go Modules are suggested to follow GoLang’s official guidelines on SIV rules to fix their induced Type C issues. For example, in #1149 [55], project redia [47] added a version suffix ”/v7” at the end of its module path to follow SIV rules. However, we noticed that while the issues are fixed, the project’s downstream projects in GOPATH may be impacted (unable to recognize the version suffixes, e.g., issue #1151 [56] reported for redia).

Solution 8: Using a hash commit ID for a specific version to replace a problematic version number in library referencing (10/144=6.9%). It fixes Type C issues, where some projects in Go Modules violate SIV rules in version numbers and cause build errors to downstream projects that are also in Go Modules. It avoids referencing problematic version numbers, by a require directive with a specific hash commit ID. For example, in issue #6048 [57], one of prometheus’s downstream projects in Go Modules chose to use directive require github.com/prometheus/prometheus 43acd0e to reference its expected version v2.12.0. Similar to Solution 5, this solution would also make developers unable to automatically fetch upgraded libraries using command go get.

As summarized in Figure 6, these solutions fix their targeted DM issues, but at the same time they may bring additional benefits (ab1–ab3) or undesired consequences (uc1–uc4). When there are multiple fixing solutions for a specific DM issue, developers are suggested to carefully consider the relevant dependencies and minimize the impact on other projects in the ecosystem, by weighing consequences against benefits.

Answer to RQ3: We observed eight common fixing solutions for DM issues, covering 95.4% of the studied issues. Most solutions could affect other projects in the ecosystem. When fixing a DM issue, developers should find a tradeoff between the benefits and the possible consequences.

IV. HERO: DM ISSUE DIAGNOSIS

Our empirical study reveals the prevalence of DM issues in the Golang ecosystem due to the chaotic use of GOPATH and Go Modules in different projects. This motivates us to develop a tool, named HERO, to help automatically detect DM issues and provide customized fixing solutions. HERO works in two steps. It first extracts dependencies among Golang projects and their library-referencing modes and then diagnoses DM issues in these projects based on our observed issue types and root causes (RQ2). It further provides customized fixing suggestions leveraging the findings in RQ3. HERO can analyze a single Golang project or monitor the heterogeneous use of the two library-referencing modes in the Golang ecosystem. Below we explain how HERO models project dependencies and detects DM issues.

A. Constructing Dependency Model

We first build a dependency model for the Golang project under analysis. We formally define the model below.

Definition 3 (Dependency model): The dependency model $D(P_v)$ for version $v$ of a project $P$ is a 3-tuple $(Pr, Ds, Us)$:

- $Pr = (ip, md, t, vd)$ records the information of the current project, where $ip$ and $md$ are $P_v$’s declared module path (for $P_v$ to be referenced by downstream projects) and library-referencing mode (GOPATH or Go Modules), respectively. If $P_v$ is in GOPATH, fields $t$ and $vd$ denote whether $P_v$ depends on any DM tool (yes or no), and a collection of import paths (sets of URLs) referencing those upstream libraries that are maintained in $P_v$’s Vendor directory but cannot be found in the repositories pointed to by URLs (e.g., due to removal or renaming), respectively. Otherwise, the two fields are set to no and null, respectively.
- $Ds = \{dp_1, dp_2, \ldots, dp_n\}$ is a collection of $P_v$’s downstream projects $dp_i$, where $dp_i = (vi, ip_i, md_i, ti_i)$. Field $vi$ denotes $dp_i$’s latest version number. Fields $ip_i, md_i,$ and $ti_i$ denote this version’s import path, library-referencing mode, and whether any DM tool is used, respectively.
- $Us = \{up_1, up_2, \ldots, up_m\}$ is a collection of $P_v$’s upstream projects $up_i$, where $up_i = (vi_i, ip_i, md_i, Si_i, I_i)$. The fields $ip_i$ and $md_i$ denote $vi_i$’s import path and library-referencing mode, respectively. If $P_v$ is in Go Modules, field $vi_i$ denotes $up_i$’s specific version declared in $P_v$’s configuration file. Otherwise (i.e., when $P_v$ is in GOPATH), $vi_i$ denotes $up_i$’s latest version number. If $ip_i$ is a v2+ project in Go Modules, field $Si_i$ denotes whether it is released by the major branch strategy (yes or no), implying whether $ip_i$ is a virtual import path. If both projects $up_i$ and $P_v$ are in Go Modules, field $I_i$ denotes whether $up_i$ is transitively introduced into $P_v$ by any project in GOPATH (yes or no). Otherwise, the two fields are set to null and no, respectively.

We explain how to obtain these field values, taking GitHub (the most popular Golang project hosting site) for example:

Step 1: Collecting Pr information. Leveraging GitHub’s REST API “repository_url” [58], HERO queries with $P_v$’s repository name to obtain its import path $ip$ and library-referencing mode $md$ by checking if a go.mod file exists
in its repository. If \( P_v \) is \( \in \text{GOPATH} \), HERO decides field \( t \) by checking whether any DM tool’s configuration file exists. Field \( vd \) is decided by parsing \( P_v \)’s source files to collect import paths for libraries maintained in the \textit{Vendor} directory, and querying via the \textit{repository_url} API with the collected import paths to check whether the corresponding libraries have been deleted or relocated (e.g., by HTTP 404: Not Found errors [58]).

\textbf{Step 2: Collecting }\textit{Ds} information. Leveraging GitHub’s REST API \textit{“code_search_url”} [58], HERO queries with \( P_v \)’s repository name to check which projects depend on it. This information is from the \textit{require directives} of a project’s \textit{go.mod} file, \textit{import directives} of its source files, or a DM tool’s configuration file. Each found project corresponds to an item \( dp_i \) in the collection \( Ds \). Note that HERO collects the latest version \( v_i \) for \( dp_i \), and decides its associated import path \( ip_i \), library-referencing mode \( md_i \) (by checking whether \( P_v \)’s repository name is declared in its \textit{go.mod} file), and field \( f_i \) (by checking whether its DM tools’ configuration file exists), respectively. These collected downstream projects depend on \( P_v \) and can also reference its earlier versions.

\textbf{Step 3: Collecting }\textit{Us} information. Project \( P_v \)’s upstream projects information is collected in two ways, depending on the library referencing mode of the project:

- \( P_v \) in \textit{Go Modules}: HERO collects \( P_v \)’s upstream projects \( up_i \) with fields \( ip_i \) and \( v_i \) by parsing its \textit{go.mod} file, which configures a project’s direct and transitive dependencies with import paths and specific version numbers. HERO identifies \( up_i \)’s library-referencing mode \( md_i \) by checking whether a \textit{go.mod} file exists in its repository via GitHub’s \textit{“repository_url”} API. If \( up_i \) is a v2+ project in \textit{Go Modules}, HERO identifies its release strategy \( S_i \) by checking whether a subdirectory like \textit{/upx/v2} exists. For projects transitively introduced into \( P_v \) by any project in \textit{Go_PATH}, Golang’s build tool automatically marks them with a \textit{//indirect} comment at the end of their module paths in \( P_v \)’s \textit{go.mod} file [59], with which HERO decides \( t_i \).
- \( P_v \) in \textit{GOPATH}: HERO collects \( P_v \)’s direct dependencies \( up_i \) with import paths \( ip_i \) from its source files. With the import paths, HERO leverages GitHub’s \textit{“repository_url”} API to look into these dependencies’ repositories to collect their latest versions, from which it decides the corresponding version numbers \( v_i \) and library-referencing modes \( md_i \). Then HERO recursively collects the information of \( P_v \)’s transitive dependencies declared in \textit{go.mod} or source files in concerned repositories, and identifies version numbers, import paths, library-referencing modes in a similar way.

\textbf{B. Diagnosing DM Issues}

The dependency model built by HERO contains sufficient information for detecting DM issues and suggesting solutions.

\textbf{Detecting DM issues.} Our study disclosed that most DM issues caused build errors, already observable. Thus, HERO
focuses on detecting DM issues that have not yet manifested, but would probably happen when the concerned projects have their upstream or downstream projects upgraded. Due to page limit, we explain scenarios for which HERO reports issues in this paper with algorithm details on our website.

**Type A.** Figure 7(a) shows a scenario, where a module-unaware project \( P_v \) references a specific version of its upstream project \( v_a \) in Go Modules. This version is older than \( v_a \)’s latest version, which newly introduces another upstream project \( v_b \) in Go Modules with a \( v2+ \) version released using the major branch strategy. Build errors do not occur in \( P_v \) when it references \( v_a \)’s old version. However, if \( P_v \) updates \( v_a \) to reference the latest version, it will not be able to recognize \( v_b \)’s virtual import path. When seeing such a possibility, HERO reports a warning of Type A issue for \( P_v \).

**Type B.1.** Figure 7(b) shows a scenario, where project \( P_v \) in Go Modules transitively references a \( v2+ \) upstream project \( v_b \) in Go Modules (released by the major branch strategy) through another module-unaware project \( v_a \) in GOPATH. Since GOPATH and Go Modules interpret import paths differently, \( v_a \) would use \( v_b \)’s latest version (e.g., \( v2.0.0 \)), while \( P_v \) would use \( v_a \)’s old \( v0/v1 \) version, causing inconsistencies. Thus, HERO reports a warning of Type B.1 issue for \( P_v \).

**Type B.2.** Figure 7(c) shows a scenario, where project \( P_v \) in GOPATH references an upstream project \( v_a \) maintained only in its Vendor directory (i.e., \( v_a \) has already been deleted or relocated). No build errors occur when \( P_v \) has no downstream projects in Go Modules. However, if \( P_v \) has such downstream projects, the latter would fetch \( v_a \) via its import path (i.e., hosting repository) rather than from \( P_v \)’s Vendor directory, causing build errors due to failing to fetch \( v_a \). Thus, HERO reports a warning of Type B.2 issue for \( P_v \).

**Type C.** Figure 7(d) shows a scenario, where project \( P_v \) in Go Modules violates SIV rules (as discussed in Sec III-C). The violation may not introduce build errors when \( P_v \) has no downstream projects in Go Modules. However, build errors would occur if such projects exist in future. Thus, HERO reports a warning of Type C issue for \( P_v \).

**Customized fixing suggestions.** Our empirical study has identified applicable fixing solutions for each issue type (Figure 6). We summarize the impacts of these solutions as templates in Figure 8. For each detected DM issue, HERO suggests all applicable solutions to developers by customizing the template with potential impact analysis based on the associated dependency model.

V. EVALUATION

We study two research questions in our evaluation of HERO:

- **RQ4 (Effectiveness):** How effective is HERO in detecting DM issues for Golang projects?
- **RQ5 (Usefulness):** Can HERO detect new DM issues for real-world Golang projects and assist the developers in fixing the detected issues?

For RQ4, we conducted experiments using the 132 DM issues from the 500 Golang projects in subjectSet2. Note that none of them overlap with those issues used in our empirical study. Specifically, we constructed a benchmark dataset containing the 132 DM issues and their project versions for evaluating whether HERO can detect these issues in the buggy versions or predict them in earlier versions. It is worth mentioning that issue-fixing versions are not necessarily issue-free, since new DM issues can be introduced after fixing as we have discussed earlier.

For RQ5, we applied HERO to the rest 19,000 of the top 20,000 Golang projects (i.e., excluding 500 used for RQs 2–3 and 500 used for RQ4). We reported the detected issues together with root cause analyses and fixing suggestions to respective developers. In our issue reports, we also highlighted the preferred solutions based on their impact on other projects.

A. RQ4: Effectiveness

**Experimental setup.** The benchmark dataset contains 38 Type A (28.8%), 15 Type B.1 (11.3%), 28 Type B.2 (21.2%), and 51 Type C (38.6%) DM issues. We collected their corresponding project versions to evaluate HERO’s capability of detecting or predicting DM issues:

- **Type A:** These issues occurred when module-unaware projects in GOPATH referenced \( v2+ \) dependencies in Go Modules by virtual import paths. Since issue occurrences would already cause build errors, we ran HERO on the previous project versions where such issues had not occurred.
- **Type B.1:** These issues occurred when projects in Go Modules referenced dependencies in GOPATH, with different import path interpretations to \( v2+ \) projects released by the major branch strategy. The inconsistency may not lead to immediate build errors or functional failures, but is indeed risky. Thus, we ran HERO on the current project versions to check whether it can detect potential issues.
- **Types B.2 and C:** The former occurred when the dependencies maintained in the current projects’ Vendor directories were deleted or relocated remotely. The latter occurred when the current projects in Go Modules violated SIV rules in both cases, the current projects would not have symptoms like build errors, but their downstream projects in Go Modules would when referencing them in future. Thus, we ran HERO on current project versions to check whether it can detect potential issues.

**Results.** Table 2 shows our experiment results. HERO reported a total of 130 DM issues (all true positives), covering 98.5% issues in the benchmark dataset. HERO achieved such a high detection rate because it constructs a dependency model that captures all necessary information on the characteristics of common DM issues. The only two missing issues are of Type A. HERO failed to detect them due to its conservative nature in identifying module-aware projects in GOPATH without using any DM tools. We note that precisely deciding module-
awareness requires checking a project’s local build environment to know whether it adopts a compatible Golang version. Currently, HERO does not support such checking.

B. RQ5: Usefulness

In total, HERO reported 2,422 new issues after analyzing the 19,000 Golang projects. Although the key information of root causes and fixing suggestions can be automatically generated by HERO, reporting these issues to developers involves substantial manual work, such as communicating with developers, helping them submit PRs, etc. As such, we only managed to report 280 issues for the top 1001–2000 popular projects (top 1–1000 already used for RQs 2–4) in the projects’ issue trackers. Table 3 summarizes the status of our reported issues. Encouragingly, 181 issues (64.6%) were quickly confirmed by the developers, and 160 confirmed issues (88.4%) were later fixed or are under fixing. For all but two fixed issues, developers adopted our suggested fixes. The other issues are still pending (likely due to the inactive maintenance of the projects). We discuss the feedback from the developers below.

Feedback on issue detection. While different types of DM issues had different confirmation rates (52.0%–74.4%), most confirmed issues received positive feedback from developers. We give some examples below. In issue #2922 (Type A) of kiali [60], a developer mentioned “I have found the same issue as you describe via the commit c453e89 [61]. I just stuck in an older version of this library”. In issue #256 (Type B.1) of flamingo-commerce [62], developers were previously unaware of the risk and commented “I guess the inconsistency of library version was imported by accident. We will create a PR to remove the occurrence”. In issue #114 (Type B.2) of tomato [63], a developer commented “Nice catch! I think it is nice to clean up our vendor directory, since library bitly/go-nsq repository is not existed anymore.” We also reported issue #16381 (Type C) [64] to project tidb [65] that violated SIV rules and the issue could affect 341 downstream projects! Our report struck a chord with tidb [64]. For category (3), developers of only two reported issues (#3970 of sensu-go [69] and #770 of libvirt [70]) did not take our fixing suggestions. Not wanting to be involved into trouble, they used other similar libraries for substitution.

The above feedback indicates that HERO is useful in discovering and predicting DM issues for Golang projects, as well as suggesting proper fixes with impact analysis. Developers also showed interest in the HERO tool. For example, one developer commented “I found that you sent many contributions on GitHub for this kind of subjects on many repositories. How do you detect the problems with Go Modules? Do you plan to share a tool or something to manage Go Modules issues?” (ovh/cds’s [71] issue #5366 [72]). Another commented “It is a good bot!” (TheThingsNetwork’s issue #780 [73]). Encouraged by such comments, we are planning to release our tool for public use to help build a healthy Golang ecosystem.

VI. DISCUSSIONS

A. Threats to Validity

One possible threat is the representativeness of the studied Golang projects and DM issues. To reduce the threat, we selected top 20,000 projects on GitHub for migration status analysis (RQ1), and randomly chose 500 from the top 1,000 projects to investigate DM issues’ characteristics (RQs 2–3). These projects are popular, large-sized, and well-maintained. We believe that they are proper subjects for our study.

Another possible threat is the generality of the issues that HERO detects since the issue types were observed by studying only 500 Golang projects. To mitigate the threat, we used a different set of DM issues to evaluate HERO (RQ4) and found that HERO can detect 98.5% of these issues, which suggests that our findings on issue characteristics are generalizable. Besides, HERO also detected a large number of real DM issues after analyzing 19,000 Golang projects. This further suggests the generality of the findings in this paper.

In addition, our study involves manual work (e.g., identifying and analyzing issue reports). To reduce the threat of human mistakes, three co-authors have cross-validated all results for consistency.

B. HERO’s Generalizability Beyond the Golang Ecosystem

Two aspects of our methodology are generalizable to the DM issues induced by incompatible library-referencing modes at other ecosystems:

- The scenarios of issue types and their causes: (1) projects in the legacy library-referencing mode depend on projects in the new library-referencing mode, (2) projects in the new mode depend on those in the legacy mode, and (3) projects in the new mode depend on others also in the new mode, can be generalized to analyze similar situations.
The formulation of issue fixing patterns. The methodology to construct the dependency model by collecting information about its upstream and downstream projects can be adapted to other ecosystems. With the aid of such a dependency model, fixing suggestions can be structurally formulated based on applicable solutions and their potential impacts. The generalization of our methodology needs to consider the unique characteristics of the studied programming languages, since our work focuses only on the Golang ecosystem (one of the most influential and fastest growing open-source ecosystems).

VII. RELATED WORK

Software dependency management. Software dependency management has inherent complexities [18]–[22], [74]–[95]. Blincoe et al. [75] studied over 70 million dependencies to find out how developers declared dependencies across 17 package managers. Their results guided research into better practices for dependency management. Abate et al. [96] reviewed state-of-the-art dependency managers and their ability to keep up with evolution at the current growth rate of popular component-based platforms, and conclude that their dependency solving abilities are not up to the task. Some studies [79]–[89], [92] focused on upgrading dependency versions, and some [77], [78], [90], [91], [95] investigated how to migrate client code to adapt to changing dependencies. Researchers [18]–[22] also proposed a series of techniques to detect, test and monitor dependency conflict issues (e.g., missing versions) for JavaScript, Java, and Python projects. Different from such conflict issues, our studied DM issues are due to incompatible library-referencing modes and their broad impacts on related projects in the Golang ecosystem.

Garcia et al.’s work [76] is closely related to our HERO, in which eight inconsistent modular dependencies were formally defined for Java-9 applications on the Java Platform Module System (JPMS). They proposed a technique DARCY to detect and repair such inconsistencies but their targeted issues are architecture-implementation mapping ones, which are different from our focus.

Health of software ecosystems. Literatures on evolving software ecosystems cover Maven [97]–[99], Apache [79], [100], Eclipse [101], Ruby [102]–[104], PyPi [22], GNOME [105], and npm [104], [106]–[112]. Many concerned techniques focus on three aspects: ecosystem modeling and analysis [98], [100], [104], [107], [108], [111]–[113], socio-technical theories within ecosystems [106], [113], and diagnosis and monitoring for ecosystem’s evolution [22], [97], [114]. For example, Blincoe et al. [113] proposed coupling references to model technical dependencies between projects, and explored characteristics of open-source or commercial software ecosystems. Zimmermann et al. [107] modeled dependencies for the npm ecosystem, and analyzed potential risks for packages that could be attacked. To the best of our knowledge, our work is the first attempt to study the health of Golang ecosystem from the perspective of DM issues.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we studied DM issues in Golang projects, which are prevalent and have caused frustrations and troubles to many Golang developers. In particular, we investigated the characteristics of DM issues, analyzed their root causes, and identified common fixing solutions. We refined our findings into detecting algorithms with customizable fixing templates. The evaluation confirmed the effectiveness of our efforts as a tool implementation HERO in detecting and diagnosing DM issues. Leveraging fixing templates and rich diagnostic information, we plan to study DM patch generation in future.

ACKNOWLEDGMENT

The authors express thanks to the anonymous reviewers for their constructive comments. Part of the work was conducted during the first author’s internship at HKUST in 2018. The work is supported by the National Natural Science Foundation of China (Grant Nos. 61932021, 61902056, 61802164, 61977014), Shenyang Young and Middle-aged Talent Support Program (Grant No. ZX2020072), the Fundamental Research Funds for the Central Universities (Grant No. N2017011), the Hong Kong RGC/GRF grant 16207120, MSRA grant, US NSF (Grant No. CCF-1845446) and Guangdong Provincial Key Laboratory (Grant No. 2020B12101001).
