Our default decision should always be to not backport, but fixes for **security issues**, **serious problems with no workaround**, and **documentation fixes** are backported to the most recent two release branches, if applicable to that branch. (for example, the most current two release branches are `release-branch.go1.16` and `release-branch.go1.17`, from which new `Go 1.16.x` and `Go 1.17.x` releases are cut) Fixes for experimental ports are generally not backported.

A "serious" problem is one that prevents a program from working at all.

As soon as an interested party thinks an issue should be considered for backport, they open one or two "child" issues titled like `package: title [1.17 backport]`. The issue should include a link to the original issue and a short rationale about why the backport might be needed.

GopherBot is capable of opening the backport issues automatically in response to comments like the following on the main issue. (The keywords are `@gopherbot`, `backport`, `please` and optionally the release. The entire message is quoted in the new issue.)

> @gopherbot please consider this for backport to 1.17, it's a regression.
>
> @gopherbot please open the backport tracking issues. This is a severe compiler bug.

The fix is developed for the main issue, which is closed when the fix is merged to the master branch.

The child issue is assigned to the minor release milestone and labeled **CherryPickCandidate**, and its candidacy is discussed there. Once it is approved it transitions to **CherryPickApproved**. Release managers (a subset of the Go team that handles the release process) and/or code owners approve cherry-picks via an informal process.

When the child issue is labeled **CherryPickApproved**, the original author of the change fixing that issue should immediately create and mail a cherry-pick change against the release branch, which will be merged as soon as it is ready, closing the child issue.

At release time, any open backport issue which is not release-blocker is pushed to the next minor release milestone, and a minor release is minted with the already merged changes.

Making cherry-pick CLs

Note that only the authors of the original CL (or maintainers with the "impersonate" permission) have the ability to create the cherry-pick.

Once the main fix has been submitted to master, please make a cherry-pick CL to the applicable release branch.

You can use the Gerrit UI to make a cherry-pick if there are no merge conflicts:

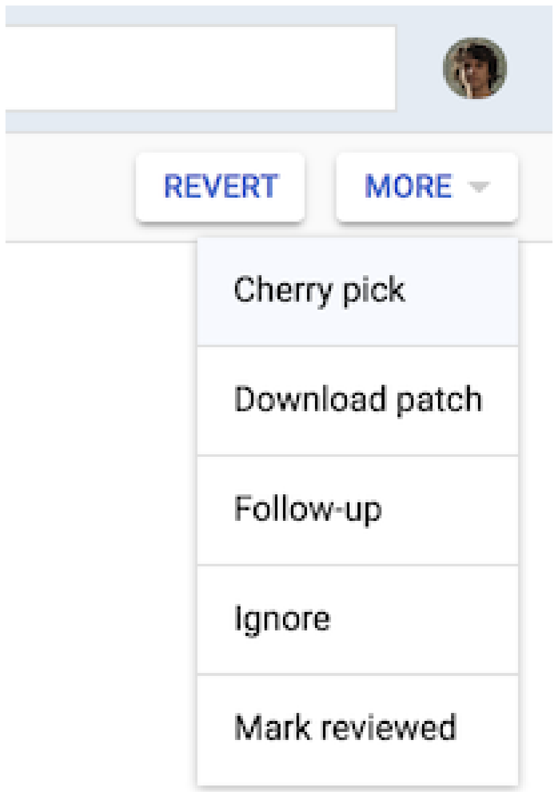

Figure 1: Top right corner > More > Cherry-pick

In the popup enter the branch name (like `release-branch.go1.10`), add the commit message prefix (like `[release-branch.go1.10]`), update the "Fixes" line and do not change any of the other automated lines.

To cherry-pick from the command line or to resolve a merge conflict, take note of the final commit hash, then use `git codereview` and `git cherry-pick` to prepare a cherry-pick CL:

```
git checkout release-branch.go1.17
git codereview change cherry-pick-NNNN
git cherry-pick $COMMIT_HASH
git commit --amend # add message prefix and change Fixes line
git codereview mail
```

The cherry-pick CL must include a message prefix like `[release-branch.go1.10]`, and update the "Fixes" line to the child issue. Do not change or remove the "Change-Id" line nor the other Gerrit lines.

Gerrit is configured to only allow release managers to submit to release branches, but the code review process is otherwise the usual.

At this time, it's not possible to make a cherry-pick CL by sending a [[Pull Request|GerritBot]]. Only Gerrit is supported. See golang.org/issue/30037.

Cherry-pick CLs for vendored golang.org/x packages

The Go standard library includes some generated files whose source of truth is outside the main repository, in golang.org/x repositories. For example, a copy of the `golang.org/x/sys/unix` package is vendored into the Go tree, and a copy of the `golang.org/x/net/http2` package is bundled. That means a fix to a golang.org/x package that needs to be backported to a Go release will need two corresponding CLs:

1. In the golang.org/x repository, cherry-pick the fix from the `master` branch to the `internal-branch.go1.x-vendor` branch.

   The commit message should include "Updates golang/go#nnn" to mention the backport issue.

2. In the main repository on the `release-branch.go1.x` branch, create a CL that pulls in the fix from the golang.org/x internal branch:

   ```
   go get -d golang.org/x/repo@internal-branch.go1.x-vendor
   go mod tidy
   go mod vendor
   go generate -run=bundle std  # If a bundled package needs regeneration.
   ```

   The commit message should include "Fixes #nnn" to close the backport issue.

(As of Go 1.16, the golang.org/x branch name is always `internal-branch.go1.x-vendor`. In Go 1.15, the name of the golang.org/x branch is `release-branch.go1.x` or `release-branch.go1.x-bundle` in special cases.)

Security releases

Note: This section describes process used for security releases before the new security policy was applied. It is not current.

Security releases preempt the next minor release and need to ship only the security fix.

To avoid rolling back the release branch in that exceptional case, a new release branch is created based on the previous release. For example `release-branch.go1.9-security` is branched from tag `go1.9.4`. The release is tagged from that branch, and the branch is then merged into the main release branch. For example `go1.9.5` is tagged from `release-branch.go1.9-security`, which is then merged into `release-branch.go1.9`.