Target-Specific Code

Sometimes, for performance or compatibility reasons, custom code needs to be written for specific GOARCH and GOOS targets. This page presents some best practices to adopt in that case. It is a required policy for crypto packages as of April 2019.

1. **Minimize code in tagged files.** As much code as possible should build for every target. In particular, the generic Go implementation must build also for targets that have an optimized implementation. This is critical for testing the optimized code against the generic Go, and makes it quicker to notice some build failures. The linker will drop unused code from final binaries.

2. **Name files after their tags**, like `poly1305_amd64.go`. Remember that if a file ends in `_$GOARCH.go`, that counts as a build tag. `_noasm.go` is also a good suffix.

3. **No exported functions in tagged files.** Exported functions define the public API and its documentation, which must be the same across all targets. Having exported functions repeated in each target-specific file makes it likely for them to get out of sync. The mid-stack inliner will probably take care of some of the performance cost.

4. **Test all available implementations.** Running `go test` on a target that has an optimized implementation should test both the generic and the optimized code. You can use sub-tests for this. Ideally, benchmarks too.

5. **Write comparative test.** There should be a test which runs the two implementations for random or edge inputs, and compares the results. As #19109 progresses, these should be fuzz tests.

Tip: you can test that your code and tests compile for a target easily by running e.g. `GOARCH=arm64 go test -c`.

Example

poly1305.go

```
package poly1305

// Sum generates an authenticator for m using a one-time key and puts the
// 16-byte result into out. Authenticating two different messages with the same
// key allows an attacker to forge messages at will.
func Sum(out *[16]byte, m []byte, key *[32]byte) {
    sum(out, m, key)
}

func sumGeneric(out *[16]byte, m []byte, key *[32]byte) {
```

```
    // ...
}
```

poly1305_amd64.go

```
// +build gc,!purego

package poly1305

//go:noescape
func sum(out *[16]byte, m []byte, key *[32]byte)
```

poly1305_amd64.s

```
// +build gc,!purego

// func sum(out *[16]byte, m []byte, key *[32]byte)
TEXT ·sum(SB), $0-128
    // ...
```

poly1305_noasm.go

```
// +build !amd64 !gc purego

package poly1305

func sum(out *[16]byte, m []byte, key *[32]byte) {
    sumGeneric(out, m, key)
}
```

poly1305_test.go

```
package poly1305

import "testing"

func testSum(t *testing.T, sum func(tag *[16]byte, msg []byte, key *[32]byte)) {
    // ...
}

func TestSum(t *testing.T) {
    t.Run("Generic", func(t *testing.T) { testSum(t, sumGeneric) })
    t.Run("Native", func(t *testing.T) { testSum(t, sum) })
}

// TestSumCompare checks the output of sum against sumGeneric.
func TestSumCompare(t *testing.T) {
    // ...
}
```

For more complete examples see the x/crypto/poly1305 and x/crypto/salsa20/salsa packages.

Note that packages in the standard library (as opposed to modules like golang.org/x/crypto) do not use the `gc` and `purego` build tags.