TLS 协议 1.2 学习札记

Cuiting Shi

2016 年 11 月 22 日

摘要

这是一篇关于 TLS 协议 1.2 的学习笔记

目录

1 Introduction

TLS 协议是基于 SSL 3.0 协议的，二者之间的差异并不是很明显. TLS 协议的主要目的是为两个通信进程提供隐私性即数据完整性. 该协议有两层: TLS Record Protocol 和 TLS Handshake Protocol. TLS Record Protocol 是 TLS 的协议的最底层，但是是位于一些传输协议的上层的 (比如 TCP). TLS Record Protocol 为连接提供了安全保障，该连接具有两个主要特点:

- 连接具有隐私性。其中采用了对称加密对数据进行加密 (比如 AES, RC4 等待). 每次连接都会生成新的对称加密的秘钥，该秘钥是基于 TLS Handshake Protocol 生成的一个 secret 生成的 (当然，也可以采用除了 TLS Handshake Protocol 之外的其它协议进行协商获取 secret). TLS Record Protocol 其实也可以不适用加密，而是直接传输明文数据.

- 第二个特点是连接是可靠的. 消息的传输包括使用 a keyed MAC 来对消息的完整性进行校验. 其中，MAC 计算会采用类似于 SHA-1 的 secure hash functions. 当然, TLS Record Protocol 也可以不适用 MAC，但是只会在如下场景中应用 – 即另外一个协议仅仅是把 TLS Record Protocol 当做协商安全参数的传输工具 .

好吧，大家经常是使用 TLS Record Protocol 来封装各种高层协议. 比如，TLS Record Protocol 可以用来封装 TLS Handshake Protocol, 从而使得在 application protocol 传输或者接收数据的首字节之前, 便可以允许 server 和 client 端进行认证和协商加密算法、cryptographic keys. TLS Handshake protocol 提供的安全连接具有下面三个基本特征:

- client 或者 server 的身份可以使用公钥、crpytography (比如, RSA, DSA 等等). 该认证是可选的, 但通常至少有一方是要进行认证的.
- 对于共享的 secret 的协商过程是安全的
- 协商过程是可靠的, 因为只要攻击者试图修改协商的信息，通信的双方必定会觉察到的。

TLS 是独立于应用协议的，高层的协议可以透明地建立在 TLS 协议上。然而，由于 TLS 标准并没有指定 protocols 应该如何通过 TLS 来增强安全性, 这使得设计者以及运行在 TLS 之上的 protocols 的实现必须得自己考虑如何 Initiate TLS handshaking、如何解析 authentication certificates.

1.1 TLS 协议 1.2 VS 1.1

相比于 TLS 协议 1.1, TLS 协议 1.2 具有更大的灵活性，改进如下:

- 原先在伪随机函数 PRF 中所使用的 MD5/SHA-1 组合被替换成了 cipher-suite-specified PRFs
- 数字签名元素中的 MD5/SHA-1 组合被替换成大衣的哈希函数, 元素现在可以增加一个 field 来致命哈希算法
- 在发送完 certificate_request 之后, 如果没有可用证书的话，clients 必须发送空的证书列表
- cipher suite 的实现者必须得实现 TLS_RSA_WITH_AES_128_CBC_SHA
- cipher suites 添加了 HMAC-SHA256, 移除掉了 IDEA 和 DES

2 TLS 协议的目的

TLS 协议的目的在于:

1. Crytographic security: 应该使用 TLS 为双方的通信建立起安全连接
2. Interoperability: 在不知道对方的代码的情况下，不同的程序员在他们编写的应用程序中都使用了 TLS, 那么这些应用程序是可以成功交换 cryptographic parameters 的.
3. Extensibility: TLS 的目的在于提供一种新的公钥和 bulk encryption methods 可以并入的协议框架中, 这样可以避免实现一个全新的安全库的可能性.
4. Relative efficiency: 由于 Cryptographic operations 常常非常耗费 CPU, 特别是其中的 public key operations, 为此, TLS 协议增加了 session caching scheme, 从而可以减少从零开始建立连接的数量.

3 Cryptographic Attributes

总共有五种 cryptographic operations:

1. digital signing
2. stream cipher encryption
3. block cipher encryption
4. authenticated encryption with additional data (AEAD) encryption
5. public key encryption

注意对称加密主要有两种模式, 一种是序列加密 (亦称为流加密 strema cipher encryption), 另外一种是分组加密 (即 block cipher encryption).

Listing 1: TLS package 中对于 net.Conn 接口的实现

```go
// Copyright 2010 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

// TLS low level connection and record layer

package tls

import (
	"bytes"
	"crypto/cipher"
	"crypto/subtle"
	"crypto/x509"
	"errors"
	"fmt"
	"io"
	"net"
	"sync"
	"sync/atomic"
	"time"
)

// A Conn represents a secured connection.
// It implements the net.Conn interface.
type Conn struct {
	// constant
	conn     net.Conn
	isClient bool

	// constant after handshake; protected by handshakeMutex
	handshakeMutex sync.Mutex // handshakeMutex < in.Mutex, out.Mutex, errMutex
	handshakeErr error     // error resulting from handshake
```

```
	vers        uint16   // TLS version
	haveVers    bool     // version has been negotiated
	config      *Config  // configuration passed to constructor
	// handshakeComplete is true if the connection is currently transfering
	// application data (i.e. is not currently processing a handshake).
	handshakeComplete bool
	// handshakes counts the number of handshakes performed on the
	// connection so far. If renegotiation is disabled then this is either
	// zero or one.
	handshakes      int
	didResume       bool // whether this connection was a session resumption
	cipherSuite     uint16
	ocspResponse    []byte  // stapled OCSP response
	scts            [][]byte // signed certificate timestamps from server
	peerCertificates []*x509.Certificate
	// verifiedChains contains the certificate chains that we built, as
	// opposed to the ones presented by the server.
	verifiedChains [][]*x509.Certificate
	// serverName contains the server name indicated by the client, if any.
	serverName string
	// secureRenegotiation is true if the server echoed the secure
	// renegotiation extension. (This is meaningless as a server because
	// renegotiation is not supported in that case.)
	secureRenegotiation bool

	// clientFinishedIsFirst is true if the client sent the first Finished
	// message during the most recent handshake. This is recorded because
	// the first transmitted Finished message is the tls-unique
	// channel-binding value.
	clientFinishedIsFirst bool
	// clientFinished and serverFinished contain the Finished message sent
	// by the client or server in the most recent handshake. This is
	// retained to support the renegotiation extension and tls-unique
	// channel-binding.
	clientFinished [12]byte
	serverFinished [12]byte

	clientProtocol         string
	clientProtocolFallback bool

	// input/output
	in, out  halfConn     // in.Mutex < out.Mutex
	rawInput *block       // raw input, right off the wire
	input    *block       // application data waiting to be read
	hand     bytes.Buffer // handshake data waiting to be read
	buffering bool        // whether records are buffered in sendBuf
	sendBuf  []byte       // a buffer of records waiting to be sent
```

```
	// bytesSent counts the bytes of application data sent.
	// packetsSent counts packets.
	bytesSent int64
	packetsSent int64

	// activeCall is an atomic int32; the low bit is whether Close has
	// been called. the rest of the bits are the number of goroutines
	// in Conn.Write.
	activeCall int32

	tmp [16]byte
}

// Access to net.Conn methods.
// Cannot just embed net.Conn because that would
// export the struct field too.

// LocalAddr returns the local network address.
func (c *Conn) LocalAddr() net.Addr {
	return c.conn.LocalAddr()
}

// RemoteAddr returns the remote network address.
func (c *Conn) RemoteAddr() net.Addr {
	return c.conn.RemoteAddr()
}

// SetDeadline sets the read and write deadlines associated with the connection.
// A zero value for t means Read and Write will not time out.
// After a Write has timed out, the TLS state is corrupt and all future writes will return the same
    error.
func (c *Conn) SetDeadline(t time.Time) error {
	return c.conn.SetDeadline(t)
}

// SetReadDeadline sets the read deadline on the underlying connection.
// A zero value for t means Read will not time out.
func (c *Conn) SetReadDeadline(t time.Time) error {
	return c.conn.SetReadDeadline(t)
}

// SetWriteDeadline sets the write deadline on the underlying connection.
// A zero value for t means Write will not time out.
// After a Write has timed out, the TLS state is corrupt and all future writes will return the same
    error.
func (c *Conn) SetWriteDeadline(t time.Time) error {
	return c.conn.SetWriteDeadline(t)
}
```

```
// A halfConn represents one direction of the record layer
// connection, either sending or receiving.
type halfConn struct {
	sync.Mutex

	err            error       // first permanent error
	version        uint16      // protocol version
	cipher         interface{} // cipher algorithm
	mac            macFunction
	seq            [8]byte // 64-bit sequence number
	bfree          *block  // list of free blocks
	additionalData [13]byte // to avoid allocs; interface method args escape

	nextCipher interface{} // next encryption state
	nextMac    macFunction // next MAC algorithm

	// used to save allocating a new buffer for each MAC.
	inDigestBuf, outDigestBuf []byte
}

func (hc *halfConn) setErrorLocked(err error) error {
	hc.err = err
	return err
}

// prepareCipherSpec sets the encryption and MAC states
// that a subsequent changeCipherSpec will use.
func (hc *halfConn) prepareCipherSpec(version uint16, cipher interface{}, mac macFunction) {
	hc.version = version
	hc.nextCipher = cipher
	hc.nextMac = mac
}

// changeCipherSpec changes the encryption and MAC states
// to the ones previously passed to prepareCipherSpec.
func (hc *halfConn) changeCipherSpec() error {
	if hc.nextCipher == nil {
		return alertInternalError
	}
	hc.cipher = hc.nextCipher
	hc.mac = hc.nextMac
	hc.nextCipher = nil
	hc.nextMac = nil
	for i := range hc.seq {
		hc.seq[i] = 0
	}
	return nil
```

```
}

// incSeq increments the sequence number.
func (hc *halfConn) incSeq() {
	for i := 7; i >= 0; i-- {
		hc.seq[i]++
		if hc.seq[i] != 0 {
			return
		}
	}

	// Not allowed to let sequence number wrap.
	// Instead, must renegotiate before it does.
	// Not likely enough to bother.
	panic("TLS: sequence number wraparound")
}

// removePadding returns an unpadded slice, in constant time, which is a prefix
// of the input. It also returns a byte which is equal to 255 if the padding
// was valid and 0 otherwise. See RFC 2246, section 6.2.3.2
func removePadding(payload []byte) ([]byte, byte) {
	if len(payload) < 1 {
		return payload, 0
	}

	paddingLen := payload[len(payload)-1]
	t := uint(len(payload)-1) - uint(paddingLen)
	// if len(payload) >= (paddingLen - 1) then the MSB of t is zero
	good := byte(int32(^t) >> 31)

	toCheck := 255 // the maximum possible padding length
	// The length of the padded data is public, so we can use an if here
	if toCheck+1 > len(payload) {
		toCheck = len(payload) - 1
	}

	for i := 0; i < toCheck; i++ {
		t := uint(paddingLen) - uint(i)
		// if i <= paddingLen then the MSB of t is zero
		mask := byte(int32(^t) >> 31)
		b := payload[len(payload)-1-i]
		good &^= mask&paddingLen ^ mask&b
	}

	// We AND together the bits of good and replicate the result across
	// all the bits.
	good &= good << 4
	good &= good << 2
```

```
	good &= good << 1
	good = uint8(int8(good) >> 7)

	toRemove := good&paddingLen + 1
	return payload[:len(payload)-int(toRemove)], good
}

// removePaddingSSL30 is a replacement for removePadding in the case that the
// protocol version is SSLv3. In this version, the contents of the padding
// are random and cannot be checked.
func removePaddingSSL30(payload []byte) ([]byte, byte) {
	if len(payload) < 1 {
		return payload, 0
	}

	paddingLen := int(payload[len(payload)-1]) + 1
	if paddingLen > len(payload) {
		return payload, 0
	}

	return payload[:len(payload)-paddingLen], 255
}

func roundUp(a, b int) int {
	return a + (b-a%b)%b
}

// cbcMode is an interface for block ciphers using cipher block chaining.
type cbcMode interface {
	cipher.BlockMode
	SetIV([]byte)
}

// decrypt checks and strips the mac and decrypts the data in b. Returns a
// success boolean, the number of bytes to skip from the start of the record in
// order to get the application payload, and an optional alert value.
func (hc *halfConn) decrypt(b *block) (ok bool, prefixLen int, alertValue alert) {
	// pull out payload
	payload := b.data[recordHeaderLen:]

	macSize := 0
	if hc.mac != nil {
		macSize = hc.mac.Size()
	}

	paddingGood := byte(255)
	explicitIVLen := 0

```

```
// decrypt
if hc.cipher != nil {
    switch c := hc.cipher.(type) {
    case cipher.Stream:
        c.XORKeyStream(payload, payload)
    case cipher.AEAD:
        explicitIVLen = 8
        if len(payload) < explicitIVLen {
            return false, 0, alertBadRecordMAC
        }
        nonce := payload[:8]
        payload = payload[8:]

        copy(hc.additionalData[:], hc.seq[:])
        copy(hc.additionalData[8:], b.data[:3])
        n := len(payload) - c.Overhead()
        hc.additionalData[11] = byte(n >> 8)
        hc.additionalData[12] = byte(n)
        var err error
        payload, err = c.Open(payload[:0], nonce, payload, hc.additionalData[:])
        if err != nil {
            return false, 0, alertBadRecordMAC
        }
        b.resize(recordHeaderLen + explicitIVLen + len(payload))
    case cbcMode:
        blockSize := c.BlockSize()
        if hc.version >= VersionTLS11 {
            explicitIVLen = blockSize
        }

        if len(payload)%blockSize != 0 || len(payload) < roundUp(explicitIVLen+macSize
            +1, blockSize) {
            return false, 0, alertBadRecordMAC
        }

        if explicitIVLen > 0 {
            c.SetIV(payload[:explicitIVLen])
            payload = payload[explicitIVLen:]
        }
        c.CryptBlocks(payload, payload)
        if hc.version == VersionSSL30 {
            payload, paddingGood = removePaddingSSL30(payload)
        } else {
            payload, paddingGood = removePadding(payload)
        }
        b.resize(recordHeaderLen + explicitIVLen + len(payload))

        // note that we still have a timing side-channel in the
```

```
                        // MAC check, below. An attacker can align the record
                        // so that a correct padding will cause one less hash
                        // block to be calculated. Then they can iteratively
                        // decrypt a record by breaking each byte. See
                        // "Password Interception in a SSL/TLS Channel", Brice
                        // Canvel et al.
                        //
                        // However, our behavior matches OpenSSL, so we leak
                        // only as much as they do.
                default:
                        panic("unknown cipher type")
                }
        }

        // check, strip mac
        if hc.mac != nil {
                if len(payload) < macSize {
                        return false, 0, alertBadRecordMAC
                }

                // strip mac off payload, b.data
                n := len(payload) - macSize
                b.data[3] = byte(n >> 8)
                b.data[4] = byte(n)
                b.resize(recordHeaderLen + explicitIVLen + n)
                remoteMAC := payload[n:]
                localMAC := hc.mac.MAC(hc.inDigestBuf, hc.seq[0:], b.data[:recordHeaderLen], payload[:n
                    ])

                if subtle.ConstantTimeCompare(localMAC, remoteMAC) != 1 || paddingGood != 255 {
                        return false, 0, alertBadRecordMAC
                }
                hc.inDigestBuf = localMAC
        }
        hc.incSeq()

        return true, recordHeaderLen + explicitIVLen, 0
}

// padToBlockSize calculates the needed padding block, if any, for a payload.
// On exit, prefix aliases payload and extends to the end of the last full
// block of payload. finalBlock is a fresh slice which contains the contents of
// any suffix of payload as well as the needed padding to make finalBlock a
// full block.
func padToBlockSize(payload []byte, blockSize int) (prefix, finalBlock []byte) {
        overrun := len(payload) % blockSize
        paddingLen := blockSize - overrun
        prefix = payload[:len(payload)-overrun]
```

```
	finalBlock = make([]byte, blockSize)
	copy(finalBlock, payload[len(payload)-overrun:])
	for i := overrun; i < blockSize; i++ {
		finalBlock[i] = byte(paddingLen - 1)
	}
	return
}

// encrypt encrypts and macs the data in b.
func (hc *halfConn) encrypt(b *block, explicitIVLen int) (bool, alert) {
	// mac
	if hc.mac != nil {
		mac := hc.mac.MAC(hc.outDigestBuf, hc.seq[0:], b.data[:recordHeaderLen], b.data[
		    recordHeaderLen+explicitIVLen:])

		n := len(b.data)
		b.resize(n + len(mac))
		copy(b.data[n:], mac)
		hc.outDigestBuf = mac
	}

	payload := b.data[recordHeaderLen:]

	// encrypt
	if hc.cipher != nil {
		switch c := hc.cipher.(type) {
		case cipher.Stream:
			c.XORKeyStream(payload, payload)
		case cipher.AEAD:
			payloadLen := len(b.data) - recordHeaderLen - explicitIVLen
			b.resize(len(b.data) + c.Overhead())
			nonce := b.data[recordHeaderLen : recordHeaderLen+explicitIVLen]
			payload := b.data[recordHeaderLen+explicitIVLen:]
			payload = payload[:payloadLen]

			copy(hc.additionalData[:], hc.seq[:])
			copy(hc.additionalData[8:], b.data[:3])
			hc.additionalData[11] = byte(payloadLen >> 8)
			hc.additionalData[12] = byte(payloadLen)

			c.Seal(payload[:0], nonce, payload, hc.additionalData[:])
		case cbcMode:
			blockSize := c.BlockSize()
			if explicitIVLen > 0 {
				c.SetIV(payload[:explicitIVLen])
				payload = payload[explicitIVLen:]
			}
			prefix, finalBlock := padToBlockSize(payload, blockSize)
```

```
			b.resize(recordHeaderLen + explicitIVLen + len(prefix) + len(finalBlock))
			c.CryptBlocks(b.data[recordHeaderLen+explicitIVLen:], prefix)
			c.CryptBlocks(b.data[recordHeaderLen+explicitIVLen+len(prefix):], finalBlock)
		default:
			panic("unknown cipher type")
		}
	}

	// update length to include MAC and any block padding needed.
	n := len(b.data) - recordHeaderLen
	b.data[3] = byte(n >> 8)
	b.data[4] = byte(n)
	hc.incSeq()

	return true, 0
}

// A block is a simple data buffer.
type block struct {
	data []byte
	off  int // index for Read
	link *block
}

// resize resizes block to be n bytes, growing if necessary.
func (b *block) resize(n int) {
	if n > cap(b.data) {
		b.reserve(n)
	}
	b.data = b.data[0:n]
}

// reserve makes sure that block contains a capacity of at least n bytes.
func (b *block) reserve(n int) {
	if cap(b.data) >= n {
		return
	}
	m := cap(b.data)
	if m == 0 {
		m = 1024
	}
	for m < n {
		m *= 2
	}
	data := make([]byte, len(b.data), m)
	copy(data, b.data)
	b.data = data
}
```

```
// readFromUntil reads from r into b until b contains at least n bytes
// or else returns an error.
func (b *block) readFromUntil(r io.Reader, n int) error {
	// quick case
	if len(b.data) >= n {
		return nil
	}

	// read until have enough.
	b.reserve(n)
	for {
		m, err := r.Read(b.data[len(b.data):cap(b.data)])
		b.data = b.data[0 : len(b.data)+m]
		if len(b.data) >= n {
			// TODO(bradfitz,agl): slightly suspicious
			// that we're throwing away r.Read's err here.
			break
		}
		if err != nil {
			return err
		}
	}
	return nil
}

func (b *block) Read(p []byte) (n int, err error) {
	n = copy(p, b.data[b.off:])
	b.off += n
	return
}

// newBlock allocates a new block, from hc's free list if possible.
func (hc *halfConn) newBlock() *block {
	b := hc.bfree
	if b == nil {
		return new(block)
	}
	hc.bfree = b.link
	b.link = nil
	b.resize(0)
	return b
}

// freeBlock returns a block to hc's free list.
// The protocol is such that each side only has a block or two on
// its free list at a time, so there's no need to worry about
// trimming the list, etc.
```

```
func (hc *halfConn) freeBlock(b *block) {
	b.link = hc.bfree
	hc.bfree = b
}

// splitBlock splits a block after the first n bytes,
// returning a block with those n bytes and a
// block with the remainder. the latter may be nil.
func (hc *halfConn) splitBlock(b *block, n int) (*block, *block) {
	if len(b.data) <= n {
		return b, nil
	}
	bb := hc.newBlock()
	bb.resize(len(b.data) - n)
	copy(bb.data, b.data[n:])
	b.data = b.data[0:n]
	return b, bb
}

// RecordHeaderError results when a TLS record header is invalid.
type RecordHeaderError struct {
	// Msg contains a human readable string that describes the error.
	Msg string
	// RecordHeader contains the five bytes of TLS record header that
	// triggered the error.
	RecordHeader [5]byte
}

func (e RecordHeaderError) Error() string { return "tls: " + e.Msg }

func (c *Conn) newRecordHeaderError(msg string) (err RecordHeaderError) {
	err.Msg = msg
	copy(err.RecordHeader[:], c.rawInput.data)
	return err
}

// readRecord reads the next TLS record from the connection
// and updates the record layer state.
// c.in.Mutex <= L; c.input == nil.
func (c *Conn) readRecord(want recordType) error {
	// Caller must be in sync with connection:
	// handshake data if handshake not yet completed,
	// else application data.
	switch want {
	default:
		c.sendAlert(alertInternalError)
		return c.in.setErrorLocked(errors.New("tls: unknown record type requested"))
	case recordTypeHandshake, recordTypeChangeCipherSpec:
```

```
		if c.handshakeComplete {
			c.sendAlert(alertInternalError)
			return c.in.setErrorLocked(errors.New("tls: handshake or ChangeCipherSpec requested while not in handshake"))
		}
	case recordTypeApplicationData:
		if !c.handshakeComplete {
			c.sendAlert(alertInternalError)
			return c.in.setErrorLocked(errors.New("tls: application data record requested while in handshake"))
		}
	}

Again:
	if c.rawInput == nil {
		c.rawInput = c.in.newBlock()
	}
	b := c.rawInput

	// Read header, payload.
	if err := b.readFromUntil(c.conn, recordHeaderLen); err != nil {
		// RFC suggests that EOF without an alertCloseNotify is
		// an error, but popular web sites seem to do this,
		// so we can't make it an error.
		// if err == io.EOF {
		//     err = io.ErrUnexpectedEOF
		// }
		if e, ok := err.(net.Error); !ok || !e.Temporary() {
			c.in.setErrorLocked(err)
		}
		return err
	}
	typ := recordType(b.data[0])

	// No valid TLS record has a type of 0x80, however SSLv2 handshakes
	// start with a uint16 length where the MSB is set and the first record
	// is always < 256 bytes long. Therefore typ == 0x80 strongly suggests
	// an SSLv2 client.
	if want == recordTypeHandshake && typ == 0x80 {
		c.sendAlert(alertProtocolVersion)
		return c.in.setErrorLocked(c.newRecordHeaderError("unsupported SSLv2 handshake received"))
	}

	vers := uint16(b.data[1])<<8 | uint16(b.data[2])
	n := int(b.data[3])<<8 | int(b.data[4])
	if c.haveVers && vers != c.vers {
		c.sendAlert(alertProtocolVersion)
```

```
		msg := fmt.Sprintf("received record with version %x when expecting version %x", vers, c
		    .vers)
		return c.in.setErrorLocked(c.newRecordHeaderError(msg))
	}
	if n > maxCiphertext {
		c.sendAlert(alertRecordOverflow)
		msg := fmt.Sprintf("oversized record received with length %d", n)
		return c.in.setErrorLocked(c.newRecordHeaderError(msg))
	}
	if !c.haveVers {
		// First message, be extra suspicious: this might not be a TLS
		// client. Bail out before reading a full 'body', if possible.
		// The current max version is 3.3 so if the version is >= 16.0,
		// it's probably not real.
		if (typ != recordTypeAlert && typ != want) || vers >= 0x1000 {
			c.sendAlert(alertUnexpectedMessage)
			return c.in.setErrorLocked(c.newRecordHeaderError("first record does not look
			    like a TLS handshake"))
		}
	}
	if err := b.readFromUntil(c.conn, recordHeaderLen+n); err != nil {
		if err == io.EOF {
			err = io.ErrUnexpectedEOF
		}
		if e, ok := err.(net.Error); !ok || !e.Temporary() {
			c.in.setErrorLocked(err)
		}
		return err
	}

	// Process message.
	b, c.rawInput = c.in.splitBlock(b, recordHeaderLen+n)
	ok, off, err := c.in.decrypt(b)
	if !ok {
		c.in.setErrorLocked(c.sendAlert(err))
	}
	b.off = off
	data := b.data[b.off:]
	if len(data) > maxPlaintext {
		err := c.sendAlert(alertRecordOverflow)
		c.in.freeBlock(b)
		return c.in.setErrorLocked(err)
	}

	switch typ {
	default:
		c.in.setErrorLocked(c.sendAlert(alertUnexpectedMessage))
```

```
	case recordTypeAlert:
		if len(data) != 2 {
			c.in.setErrorLocked(c.sendAlert(alertUnexpectedMessage))
			break
		}
		if alert(data[1]) == alertCloseNotify {
			c.in.setErrorLocked(io.EOF)
			break
		}
		switch data[0] {
		case alertLevelWarning:
			// drop on the floor
			c.in.freeBlock(b)
			goto Again
		case alertLevelError:
			c.in.setErrorLocked(&net.OpError{Op: "remote error", Err: alert(data[1])})
		default:
			c.in.setErrorLocked(c.sendAlert(alertUnexpectedMessage))
		}

	case recordTypeChangeCipherSpec:
		if typ != want || len(data) != 1 || data[0] != 1 {
			c.in.setErrorLocked(c.sendAlert(alertUnexpectedMessage))
			break
		}
		err := c.in.changeCipherSpec()
		if err != nil {
			c.in.setErrorLocked(c.sendAlert(err.(alert)))
		}

	case recordTypeApplicationData:
		if typ != want {
			c.in.setErrorLocked(c.sendAlert(alertUnexpectedMessage))
			break
		}
		c.input = b
		b = nil

	case recordTypeHandshake:
		// TODO(rsc): Should at least pick off connection close.
		if typ != want && !(c.isClient && c.config.Renegotiation != RenegotiateNever) {
			return c.in.setErrorLocked(c.sendAlert(alertNoRenegotiation))
		}
		c.hand.Write(data)
	}

	if b != nil {
		c.in.freeBlock(b)
```

```
	}
	return c.in.err
}

// sendAlert sends a TLS alert message.
// c.out.Mutex <= L.
func (c *Conn) sendAlertLocked(err alert) error {
	switch err {
	case alertNoRenegotiation, alertCloseNotify:
		c.tmp[0] = alertLevelWarning
	default:
		c.tmp[0] = alertLevelError
	}
	c.tmp[1] = byte(err)

	_, writeErr := c.writeRecordLocked(recordTypeAlert, c.tmp[0:2])
	if err == alertCloseNotify {
		// closeNotify is a special case in that it isn't an error.
		return writeErr
	}

	return c.out.setErrorLocked(&net.OpError{Op: "local error", Err: err})
}

// sendAlert sends a TLS alert message.
// L < c.out.Mutex.
func (c *Conn) sendAlert(err alert) error {
	c.out.Lock()
	defer c.out.Unlock()
	return c.sendAlertLocked(err)
}

const (
	// tcpMSSEstimate is a conservative estimate of the TCP maximum segment
	// size (MSS). A constant is used, rather than querying the kernel for
	// the actual MSS, to avoid complexity. The value here is the IPv6
	// minimum MTU (1280 bytes) minus the overhead of an IPv6 header (40
	// bytes) and a TCP header with timestamps (32 bytes).
	tcpMSSEstimate = 1208

	// recordSizeBoostThreshold is the number of bytes of application data
	// sent after which the TLS record size will be increased to the
	// maximum.
	recordSizeBoostThreshold = 128 * 1024
)

// maxPayloadSizeForWrite returns the maximum TLS payload size to use for the
// next application data record. There is the following trade-off:
```

```go
//
//   - For latency-sensitive applications, such as web browsing, each TLS
//     record should fit in one TCP segment.
//   - For throughput-sensitive applications, such as large file transfers,
//     larger TLS records better amortize framing and encryption overheads.
//
// A simple heuristic that works well in practice is to use small records for
// the first 1MB of data, then use larger records for subsequent data, and
// reset back to smaller records after the connection becomes idle. See "High
// Performance Web Networking", Chapter 4, or:
// https://www.igvita.com/2013/10/24/optimizing-tls-record-size-and-buffering-latency/
//
// In the interests of simplicity and determinism, this code does not attempt
// to reset the record size once the connection is idle, however.
//
// c.out.Mutex <= L.
func (c *Conn) maxPayloadSizeForWrite(typ recordType, explicitIVLen int) int {
	if c.config.DynamicRecordSizingDisabled || typ != recordTypeApplicationData {
		return maxPlaintext
	}

	if c.bytesSent >= recordSizeBoostThreshold {
		return maxPlaintext
	}

	// Subtract TLS overheads to get the maximum payload size.
	macSize := 0
	if c.out.mac != nil {
		macSize = c.out.mac.Size()
	}

	payloadBytes := tcpMSSEstimate - recordHeaderLen - explicitIVLen
	if c.out.cipher != nil {
		switch ciph := c.out.cipher.(type) {
		case cipher.Stream:
			payloadBytes -= macSize
		case cipher.AEAD:
			payloadBytes -= ciph.Overhead()
		case cbcMode:
			blockSize := ciph.BlockSize()
			// The payload must fit in a multiple of blockSize, with
			// room for at least one padding byte.
			payloadBytes = (payloadBytes & ^(blockSize - 1)) - 1
			// The MAC is appended before padding so affects the
			// payload size directly.
			payloadBytes -= macSize
		default:
			panic("unknown cipher type")
```

```
			}
		}

		// Allow packet growth in arithmetic progression up to max.
		pkt := c.packetsSent
		c.packetsSent++
		if pkt > 1000 {
			return maxPlaintext // avoid overflow in multiply below
		}

		n := payloadBytes * int(pkt+1)
		if n > maxPlaintext {
			n = maxPlaintext
		}
		return n
}

// c.out.Mutex <= L.
func (c *Conn) write(data []byte) (int, error) {
		if c.buffering {
			c.sendBuf = append(c.sendBuf, data...)
			return len(data), nil
		}

		n, err := c.conn.Write(data)
		c.bytesSent += int64(n)
		return n, err
}

func (c *Conn) flush() (int, error) {
		if len(c.sendBuf) == 0 {
			return 0, nil
		}

		n, err := c.conn.Write(c.sendBuf)
		c.bytesSent += int64(n)
		c.sendBuf = nil
		c.buffering = false
		return n, err
}

// writeRecordLocked writes a TLS record with the given type and payload to the
// connection and updates the record layer state.
// c.out.Mutex <= L.
func (c *Conn) writeRecordLocked(typ recordType, data []byte) (int, error) {
		b := c.out.newBlock()
		defer c.out.freeBlock(b)
```

```go
	var n int
	for len(data) > 0 {
		explicitIVLen := 0
		explicitIVIsSeq := false

		var cbc cbcMode
		if c.out.version >= VersionTLS11 {
			var ok bool
			if cbc, ok = c.out.cipher.(cbcMode); ok {
				explicitIVLen = cbc.BlockSize()
			}
		}
		if explicitIVLen == 0 {
			if _, ok := c.out.cipher.(cipher.AEAD); ok {
				explicitIVLen = 8
				// The AES-GCM construction in TLS has an
				// explicit nonce so that the nonce can be
				// random. However, the nonce is only 8 bytes
				// which is too small for a secure, random
				// nonce. Therefore we use the sequence number
				// as the nonce.
				explicitIVIsSeq = true
			}
		}
		m := len(data)
		if maxPayload := c.maxPayloadSizeForWrite(typ, explicitIVLen); m > maxPayload {
			m = maxPayload
		}
		b.resize(recordHeaderLen + explicitIVLen + m)
		b.data[0] = byte(typ)
		vers := c.vers
		if vers == 0 {
			// Some TLS servers fail if the record version is
			// greater than TLS 1.0 for the initial ClientHello.
			vers = VersionTLS10
		}
		b.data[1] = byte(vers >> 8)
		b.data[2] = byte(vers)
		b.data[3] = byte(m >> 8)
		b.data[4] = byte(m)
		if explicitIVLen > 0 {
			explicitIV := b.data[recordHeaderLen : recordHeaderLen+explicitIVLen]
			if explicitIVIsSeq {
				copy(explicitIV, c.out.seq[:])
			} else {
				if _, err := io.ReadFull(c.config.rand(), explicitIV); err != nil {
					return n, err
				}
```

```
			}
		}
		copy(b.data[recordHeaderLen+explicitIVLen:], data)
		c.out.encrypt(b, explicitIVLen)
		if _, err := c.write(b.data); err != nil {
			return n, err
		}
		n += m
		data = data[m:]
	}

	if typ == recordTypeChangeCipherSpec {
		if err := c.out.changeCipherSpec(); err != nil {
			return n, c.sendAlertLocked(err.(alert))
		}
	}

	return n, nil
}

// writeRecord writes a TLS record with the given type and payload to the
// connection and updates the record layer state.
// L < c.out.Mutex.
func (c *Conn) writeRecord(typ recordType, data []byte) (int, error) {
	c.out.Lock()
	defer c.out.Unlock()

	return c.writeRecordLocked(typ, data)
}

// readHandshake reads the next handshake message from
// the record layer.
// c.in.Mutex < L; c.out.Mutex < L.
func (c *Conn) readHandshake() (interface{}, error) {
	for c.hand.Len() < 4 {
		if err := c.in.err; err != nil {
			return nil, err
		}
		if err := c.readRecord(recordTypeHandshake); err != nil {
			return nil, err
		}
	}

	data := c.hand.Bytes()
	n := int(data[1])<<16 | int(data[2])<<8 | int(data[3])
	if n > maxHandshake {
		c.sendAlertLocked(alertInternalError)
		return nil, c.in.setErrorLocked(fmt.Errorf("tls: handshake message of length %d bytes
```

```
                    exceeds maximum of %d bytes", n, maxHandshake))
        }
        for c.hand.Len() < 4+n {
                if err := c.in.err; err != nil {
                        return nil, err
                }
                if err := c.readRecord(recordTypeHandshake); err != nil {
                        return nil, err
                }
        }
        data = c.hand.Next(4 + n)
        var m handshakeMessage
        switch data[0] {
        case typeHelloRequest:
                m = new(helloRequestMsg)
        case typeClientHello:
                m = new(clientHelloMsg)
        case typeServerHello:
                m = new(serverHelloMsg)
        case typeNewSessionTicket:
                m = new(newSessionTicketMsg)
        case typeCertificate:
                m = new(certificateMsg)
        case typeCertificateRequest:
                m = &certificateRequestMsg{
                        hasSignatureAndHash: c.vers >= VersionTLS12,
                }
        case typeCertificateStatus:
                m = new(certificateStatusMsg)
        case typeServerKeyExchange:
                m = new(serverKeyExchangeMsg)
        case typeServerHelloDone:
                m = new(serverHelloDoneMsg)
        case typeClientKeyExchange:
                m = new(clientKeyExchangeMsg)
        case typeCertificateVerify:
                m = &certificateVerifyMsg{
                        hasSignatureAndHash: c.vers >= VersionTLS12,
                }
        case typeNextProtocol:
                m = new(nextProtoMsg)
        case typeFinished:
                m = new(finishedMsg)
        default:
                return nil, c.in.setErrorLocked(c.sendAlert(alertUnexpectedMessage))
        }

        // The handshake message unmarshallers
```

```
	// expect to be able to keep references to data,
	// so pass in a fresh copy that won't be overwritten.
	data = append([]byte(nil), data...)

	if !m.unmarshal(data) {
		return nil, c.in.setErrorLocked(c.sendAlert(alertUnexpectedMessage))
	}
	return m, nil
}

var errClosed = errors.New("tls: use of closed connection")

// Write writes data to the connection.
func (c *Conn) Write(b []byte) (int, error) {
	// interlock with Close below
	for {
		x := atomic.LoadInt32(&c.activeCall)
		if x&1 != 0 {
			return 0, errClosed
		}
		if atomic.CompareAndSwapInt32(&c.activeCall, x, x+2) {
			defer atomic.AddInt32(&c.activeCall, -2)
			break
		}
	}

	if err := c.Handshake(); err != nil {
		return 0, err
	}

	c.out.Lock()
	defer c.out.Unlock()

	if err := c.out.err; err != nil {
		return 0, err
	}

	if !c.handshakeComplete {
		return 0, alertInternalError
	}

	// SSL 3.0 and TLS 1.0 are susceptible to a chosen-plaintext
	// attack when using block mode ciphers due to predictable IVs.
	// This can be prevented by splitting each Application Data
	// record into two records, effectively randomizing the IV.
	//
	// http://www.openssl.org/~bodo/tls-cbc.txt
	// https://bugzilla.mozilla.org/show_bug.cgi?id=665814
```

```
	// http://www.imperialviolet.org/2012/01/15/beastfollowup.html

	var m int
	if len(b) > 1 && c.vers <= VersionTLS10 {
		if _, ok := c.out.cipher.(cipher.BlockMode); ok {
			n, err := c.writeRecordLocked(recordTypeApplicationData, b[:1])
			if err != nil {
				return n, c.out.setErrorLocked(err)
			}
			m, b = 1, b[1:]
		}
	}

	n, err := c.writeRecordLocked(recordTypeApplicationData, b)
	return n + m, c.out.setErrorLocked(err)
}

// handleRenegotiation processes a HelloRequest handshake message.
// c.in.Mutex <= L
func (c *Conn) handleRenegotiation() error {
	msg, err := c.readHandshake()
	if err != nil {
		return err
	}

	_, ok := msg.(*helloRequestMsg)
	if !ok {
		c.sendAlert(alertUnexpectedMessage)
		return alertUnexpectedMessage
	}

	if !c.isClient {
		return c.sendAlert(alertNoRenegotiation)
	}

	switch c.config.Renegotiation {
	case RenegotiateNever:
		return c.sendAlert(alertNoRenegotiation)
	case RenegotiateOnceAsClient:
		if c.handshakes > 1 {
			return c.sendAlert(alertNoRenegotiation)
		}
	case RenegotiateFreelyAsClient:
		// Ok.
	default:
		c.sendAlert(alertInternalError)
		return errors.New("tls: unknown Renegotiation value")
	}
```

```
	c.handshakeMutex.Lock()
	defer c.handshakeMutex.Unlock()

	c.handshakeComplete = false
	if c.handshakeErr = c.clientHandshake(); c.handshakeErr == nil {
		c.handshakes++
	}
	return c.handshakeErr
}

// Read can be made to time out and return a net.Error with Timeout() == true
// after a fixed time limit; see SetDeadline and SetReadDeadline.
func (c *Conn) Read(b []byte) (n int, err error) {
	if err = c.Handshake(); err != nil {
		return
	}
	if len(b) == 0 {
		// Put this after Handshake, in case people were calling
		// Read(nil) for the side effect of the Handshake.
		return
	}

	c.in.Lock()
	defer c.in.Unlock()

	// Some OpenSSL servers send empty records in order to randomize the
	// CBC IV. So this loop ignores a limited number of empty records.
	const maxConsecutiveEmptyRecords = 100
	for emptyRecordCount := 0; emptyRecordCount <= maxConsecutiveEmptyRecords; emptyRecordCount++ {
		for c.input == nil && c.in.err == nil {
			if err := c.readRecord(recordTypeApplicationData); err != nil {
				// Soft error, like EAGAIN
				return 0, err
			}
			if c.hand.Len() > 0 {
				// We received handshake bytes, indicating the
				// start of a renegotiation.
				if err := c.handleRenegotiation(); err != nil {
					return 0, err
				}
			}
		}
		if err := c.in.err; err != nil {
			return 0, err
		}
```

```
		n, err = c.input.Read(b)
		if c.input.off >= len(c.input.data) {
			c.in.freeBlock(c.input)
			c.input = nil
		}

		// If a close-notify alert is waiting, read it so that
		// we can return (n, EOF) instead of (n, nil), to signal
		// to the HTTP response reading goroutine that the
		// connection is now closed. This eliminates a race
		// where the HTTP response reading goroutine would
		// otherwise not observe the EOF until its next read,
		// by which time a client goroutine might have already
		// tried to reuse the HTTP connection for a new
		// request.
		// See https://codereview.appspot.com/76400046
		// and https://golang.org/issue/3514
		if ri := c.rawInput; ri != nil &&
			n != 0 && err == nil &&
			c.input == nil && len(ri.data) > 0 && recordType(ri.data[0]) == recordTypeAlert {
			if recErr := c.readRecord(recordTypeApplicationData); recErr != nil {
				err = recErr // will be io.EOF on closeNotify
			}
		}

		if n != 0 || err != nil {
			return n, err
		}
	}

	return 0, io.ErrNoProgress
}

// Close closes the connection.
func (c *Conn) Close() error {
	// Interlock with Conn.Write above.
	var x int32
	for {
		x = atomic.LoadInt32(&c.activeCall)
		if x&1 != 0 {
			return errClosed
		}
		if atomic.CompareAndSwapInt32(&c.activeCall, x, x|1) {
			break
		}
	}
	if x != 0 {
```

```
		// io.Writer and io.Closer should not be used concurrently.
		// If Close is called while a Write is currently in-flight,
		// interpret that as a sign that this Close is really just
		// being used to break the Write and/or clean up resources and
		// avoid sending the alertCloseNotify, which may block
		// waiting on handshakeMutex or the c.out mutex.
		return c.conn.Close()
	}

	var alertErr error

	c.handshakeMutex.Lock()
	defer c.handshakeMutex.Unlock()
	if c.handshakeComplete {
		alertErr = c.sendAlert(alertCloseNotify)
	}

	if err := c.conn.Close(); err != nil {
		return err
	}
	return alertErr
}

// Handshake runs the client or server handshake
// protocol if it has not yet been run.
// Most uses of this package need not call Handshake
// explicitly: the first Read or Write will call it automatically.
func (c *Conn) Handshake() error {
	// c.handshakeErr and c.handshakeComplete are protected by
	// c.handshakeMutex. In order to perform a handshake, we need to lock
	// c.in also and c.handshakeMutex must be locked after c.in.
	//
	// However, if a Read() operation is hanging then it'll be holding the
	// lock on c.in and so taking it here would cause all operations that
	// need to check whether a handshake is pending (such as Write) to
	// block.
	//
	// Thus we take c.handshakeMutex first and, if we find that a handshake
	// is needed, then we unlock, acquire c.in and c.handshakeMutex in the
	// correct order, and check again.
	c.handshakeMutex.Lock()
	defer c.handshakeMutex.Unlock()

	for i := 0; i < 2; i++ {
		if i == 1 {
			c.handshakeMutex.Unlock()
			c.in.Lock()
			defer c.in.Unlock()
```

```
			c.handshakeMutex.Lock()
		}

		if err := c.handshakeErr; err != nil {
			return err
		}
		if c.handshakeComplete {
			return nil
		}
	}

	if c.isClient {
		c.handshakeErr = c.clientHandshake()
	} else {
		c.handshakeErr = c.serverHandshake()
	}
	if c.handshakeErr == nil {
		c.handshakes++
	}
	return c.handshakeErr
}

// ConnectionState returns basic TLS details about the connection.
func (c *Conn) ConnectionState() ConnectionState {
	c.handshakeMutex.Lock()
	defer c.handshakeMutex.Unlock()

	var state ConnectionState
	state.HandshakeComplete = c.handshakeComplete
	if c.handshakeComplete {
		state.Version = c.vers
		state.NegotiatedProtocol = c.clientProtocol
		state.DidResume = c.didResume
		state.NegotiatedProtocolIsMutual = !c.clientProtocolFallback
		state.CipherSuite = c.cipherSuite
		state.PeerCertificates = c.peerCertificates
		state.VerifiedChains = c.verifiedChains
		state.ServerName = c.serverName
		state.SignedCertificateTimestamps = c.scts
		state.OCSPResponse = c.ocspResponse
		if !c.didResume {
			if c.clientFinishedIsFirst {
				state.TLSUnique = c.clientFinished[:]
			} else {
				state.TLSUnique = c.serverFinished[:]
			}
		}
	}
```

```
	return state
}

// OCSPResponse returns the stapled OCSP response from the TLS server, if
// any. (Only valid for client connections.)
func (c *Conn) OCSPResponse() []byte {
	c.handshakeMutex.Lock()
	defer c.handshakeMutex.Unlock()

	return c.ocspResponse
}

// VerifyHostname checks that the peer certificate chain is valid for
// connecting to host. If so, it returns nil; if not, it returns an error
// describing the problem.
func (c *Conn) VerifyHostname(host string) error {
	c.handshakeMutex.Lock()
	defer c.handshakeMutex.Unlock()
	if !c.isClient {
		return errors.New("tls: VerifyHostname called on TLS server connection")
	}
	if !c.handshakeComplete {
		return errors.New("tls: handshake has not yet been performed")
	}
	if len(c.verifiedChains) == 0 {
		return errors.New("tls: handshake did not verify certificate chain")
	}
	return c.peerCertificates[0].VerifyHostname(host)
}
```

3.1 signature

主要有两种数字签名算法 DSA: 一种 RSA, 另外一种 ECDSA. 对于 RSA 签名, 会使用 RSASSA-PKCS1-v1_5 签名机制，其中 DigestInfo 必须是 DER 编码的. 对于 DSA 签名, SHA-1 hash 的 20 个字节是直接通过数字签名算法运行得到，会生成 r 和 s 这两个大整数.

3.2 encyption

对于 stream cipher encryption, 明文的每一个比特位都会和由一个 cryptographically secure keyed pseudorandom number generator 生成的数据 (比特位数与明文的位数相同) 进行异或. 换句话说, 序列密码法是一种文本加密方法，其中，在数据流的各个二进制数字中应用加密秘钥以及算法，每次处理一个比特位;

对于 block cipher encryption, 即分组密码法, 明文的每一个分组 (块) 都被加密成一个密文块. CBC (Cipher Block Chaining) 模式中会完成所有的 block cipher encryption, 而且 all items that are block-ciphered will be an exact multiple of the cipher block length.

对于 AEAD encryption, 明文在被加密的同时保证了明文的数据完整性. 输入可以是任意长度的, 然后 AEAD-ciphered output 通常要比输入的长度大一些，毕竟它加上了完整性校验值.

对于 public key encryption, 使用公钥算法可以对明文进行加密, 但是得到的密文只能被对应的秘钥进行解密. A public-key-encrypted element is encoded as an opague vector $< 0..2^{16} - 1 >$, 其中长度由加密算法和 key 决定.

而对于 RSA encryption, 这个是使用 RSAES-PKCS1-v1_5 encryption scheme 实现的.

4 HMAC and the Pseudorandom Function

为了保护消息的完整性, TLS record layer 使用了 a keyed Message Authentication Code (MAC). 而该 RFC 中定义的 cipher suites 使用了 HMAC 来生成 MAC . HMAC 是基于哈希函数的.

另外, 为了生成 key 或者进行验证, 需要使用 construction 来将 secrets 扩展到数据块中。这个伪随机函数 (PRF) 的输入是 a secret, a seed, an identifying label, 输出是长度不固定的数据.

下面将基于 HMAC 来定义一个伪随机函数 PRF, 该函数还统一使用了 SHA-256 哈希函数. 当然，也可以通过明确指出 PRF 来定义新的 cipher suites，但是这得用比 SHA-256 安全性更高的哈希函数.

首先，定义一下 P_hash(secret, data) 函数, 该函数是用来扩张数据（其实是增加输出的 MAC 的位数的), 它使用一个函数函数来讲 secret 和 seed 扩张为任意长度的输出:

```
P_Hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...

//其中， A()的定义如下:
A(0) = seed
A(1) = HMAC_hash(secret, A(i-1))
```

其中, P_hash 函数可以被调用任意次数, 从而可以生成特定长度的数据. 比如, P_Hash 使用了 SHA256，则如果要生成 80 字节的数据的话，得迭代三次 (SHA256 生成的哈希是 16 字节，$16 + 32 + 48 = 96$), 得到 96 字节的数据, 然后去掉末尾的 16 字节, 便可以得到 80 字节的数据. 而 TLS 中的 PRF 是通过 P_Hash() 函数生成的，如下

```
//其中label是一个ASCII编码的字符串
PRF(secret, label, seed) = P_[hash](scret, label + seed)
```

下面是 go 的 tls 包对于 P_hash 函数的实现，注意，增加了一个新的参数 hash，把其中要用到的哈希函数作为参数传进来

Listing 2: go 中对于 pHash 的实现

```go
// pHash implements the P_hash function, as defined in RFC 4346, section 5.
func pHash(result, secret, seed []byte, hash func() hash.Hash) {
	h := hmac.New(hash, secret)
	h.Write(seed)
	a := h.Sum(nil)
```

```
	j := 0
	for j < len(result) {
		h.Reset()
		h.Write(a)
		h.Write(seed)
		b := h.Sum(nil)
		todo := len(b)
		if j+todo > len(result) {
			todo = len(result) - j
		}
		copy(result[j:j+todo], b)
		j += todo

		h.Reset()
		h.Write(a)
		a = h.Sum(nil)
	}
}
```

接下来是关于 PRF 算法的, PRF 算法是用来从 master secret 来生成 keys 的. go 的 tls 包对于 PRF 函数的实现，总共有三种 – prf10、prf12、prf30，分别对应 TLS 1.0, TLS 1.2 以及 SSL/3.0 协议, 下面只介绍 prf12 的实现:

Listing 3: prf12 的实现

```
// prf12 implements the TLS 1.2 pseudo-random function, as defined in RFC 5246, section 5.
func prf12(hashFunc func() hash.Hash) func(result, secret, label, seed []byte) {
	return func(result, secret, label, seed []byte) {
		labelAndSeed := make([]byte, len(label)+len(seed))
		copy(labelAndSeed, label)
		copy(labelAndSeed[len(label):], seed)

		pHash(result, secret, labelAndSeed, hashFunc)
	}
}
```

然后这个是 TLS 中对于 HMAC 的使用，

Listing 4: TLS 中 Hmac 的使用

```
func (c *Conn) encryptTicket(state *sessionState) ([]byte, error) {
	serialized := state.marshal()
	encrypted := make([]byte, ticketKeyNameLen+aes.BlockSize+len(serialized)+sha256.Size)
	keyName := encrypted[:ticketKeyNameLen]
	iv := encrypted[ticketKeyNameLen : ticketKeyNameLen+aes.BlockSize]
	macBytes := encrypted[len(encrypted)-sha256.Size:]

	if _, err := io.ReadFull(c.config.rand(), iv); err != nil {
		return nil, err
```

```go
	}
	key := c.config.ticketKeys()[0]
	copy(keyName, key.keyName[:])
	block, err := aes.NewCipher(key.aesKey[:])
	if err != nil {
		return nil, errors.New("tls: failed to create cipher while encrypting ticket: " + err.
		    Error())
	}
	cipher.NewCTR(block, iv).XORKeyStream(encrypted[ticketKeyNameLen+aes.BlockSize:], serialized)

	mac := hmac.New(sha256.New, key.hmacKey[:])
	mac.Write(encrypted[:len(encrypted)-sha256.Size])
	mac.Sum(macBytes[:0])

	return encrypted, nil
}
```

下面是 crypto/hmac 对于 HMAC 的实现源码:

Listing 5: Go 中 Hmac 的实现

```go
type hmac struct {
	size      int
	blocksize int
	opad, ipad []byte
	outer, inner hash.Hash
}

func (h *hmac) Sum(in []byte) []byte {
	origLen := len(in)
	in = h.inner.Sum(in)
	h.outer.Reset()
	h.outer.Write(h.opad)
	h.outer.Write(in[origLen:])
	return h.outer.Sum(in[:origLen])
}

func (h *hmac) Write(p []byte) (n int, err error) {
	return h.inner.Write(p)
}

func (h *hmac) Size() int { return h.size }

func (h *hmac) BlockSize() int { return h.blocksize }

func (h *hmac) Reset() {
	h.inner.Reset()
	h.inner.Write(h.ipad)
}
```

```
// New returns a new HMAC hash using the given hash.Hash type and key.
func New(h func() hash.Hash, key []byte) hash.Hash {
	hm := new(hmac)
	hm.outer = h()
	hm.inner = h()
	hm.size = hm.inner.Size()
	hm.blocksize = hm.inner.BlockSize()
	hm.ipad = make([]byte, hm.blocksize)
	hm.opad = make([]byte, hm.blocksize)
	if len(key) > hm.blocksize {
		// If key is too big, hash it.
		hm.outer.Write(key)
		key = hm.outer.Sum(nil)
	}
	copy(hm.ipad, key)
	copy(hm.opad, key)
	for i := range hm.ipad {
		hm.ipad[i] ^= 0x36
	}
	for i := range hm.opad {
		hm.opad[i] ^= 0x5c
	}
	hm.inner.Write(hm.ipad)
	return hm
}
```

5 TLS Record Protocol

TLS Record Protocol 是一个分层协议，在每一层，消息可能包含长度、备注和内容等信息域。TLS Record Protocol 主要负责携带待传输的消息, 将数据分成分组、可能还会对其进行压缩、对数据生成 MAC、加密、然后传输结果. 在接收到数据的时候, TLS Record Protocol 会解密、验证 MAC、解压、重组分块、然后再发送给上一级的 clients.

使用 TLS Record Protocol 的 clients 只要有四种协议:

- the TLS Handshake protocol
- the Alert Protocol
- the change cipher spec protocol
- the application data protocol

5.1 Connection States

TLS connection state 是 TLS Record Protocol 的运作环境. 它指明了压缩算法、加密算法、MAC 算法. 下面是 go 的 tls 包关于 connection state 的定义:

Listing 6: Go 中 connection 的定义

```
// ConnectionState records basic TLS details about the connection.
type ConnectionState struct {
	Version                     uint16   // TLS version used by the connection (e.g. VersionTLS12)
	HandshakeComplete           bool     // TLS handshake is complete
	DidResume                   bool     // connection resumes a previous TLS connection
	CipherSuite                 uint16   // cipher suite in use (TLS_RSA_WITH_RC4_128_SHA, ...)
	NegotiatedProtocol          string   // negotiated next protocol (from Config.NextProtos)
	NegotiatedProtocolIsMutual  bool     // negotiated protocol was advertised by server
	ServerName                  string   // server name requested by client, if any (server side only
	    )
	PeerCertificates            []*x509.Certificate // certificate chain presented by remote peer
	VerifiedChains              [][]*x509.Certificate // verified chains built from PeerCertificates
	SignedCertificateTimestamps [][]byte // SCTs from the server, if any
	OCSPResponse                []byte   // stapled OCSP response from server, if any

	// TLSUnique contains the "tls-unique" channel binding value (see RFC
	// 5929, section 3). For resumed sessions this value will be nil
	// because resumption does not include enough context (see
	// https://secure-resumption.com/#channelbindings). This will change in
	// future versions of Go once the TLS master-secret fix has been
	// standardized and implemented.
	TLSUnique []byte
}
```

5.2 影响 connection states 的参数 – 术语

注意，TLS record layer 会使用下面列出的安全参数生成 client write MAC key, server write MAC key, client write encryption key, server write encryption key, client write IV, server write IV. 对于 client write 参数，server 在接收和处理 records 的时候，会使用 client write 这些参数，反之亦然。当下面列的安全参数设置完毕，并且生成了相关的 keys 之后，则连接状态 connection states 便可以通过转移到 current states 被实例化. 每个连接状态都会包含如下的条目：

- compression state: 即压缩算法的当前状态
- cipher state: 加密算法的当前状态
- MAC key: 使用上面的那些安全参数生成的 MAC 算法的 key
- sequence number: 读写状态各自维护了自己的 sequence number

安全参数：

master secret A 48-byte secret shared between the two peers in the connection

client random A 32-byte value provided by the client

server random A 32-byte value provided by the server

RPF algorithm An algorithm used to generate keys from the master secret

bulk encryption algorithm An algorithm to be used for bulk encryption. 必须指明算法的 key size, 何种 cipher (block, stream, or AEAD cipher), cipher 所使用的 block size, the length of explicit and implicit initialization vectors (or nonces)

MAC algorithm An algorithm to be used for message authentication. 必须要指明算法所返回的 MAC 碼的大小

5.3 Record layer

The TLS record layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

5.3.1 Fragmentation

The record layer fragments informatin blocks into TLSPaintext records carrying data in chunks of $2^1$4 bytes or less. Client message boundaries are not preserved in the record layer (比如, multiple client messages of the same ContentType MAY be coalesced into a single TLSPlaintext record, or a single message MAY be fragmented across several records). 也就是说，多个相同内容类型的 client messages 可能会被重组成一条记录 TLSPlaintext record, 但是一条过大的 client message 也可能会被切分成多条记录。

此外，值得注意的是，由于历史原因，TLS1.0 继承了 SSL/3.0，版本号可能会写作 3.1, 而 TLS1.2 对应 3.3

Listing 7: TLS 协议中的 TLS record 类型以及 handshake 消息类型

```
const (
	VersionSSL30 = 0x0300
	VersionTLS10 = 0x0301
	VersionTLS11 = 0x0302
	VersionTLS12 = 0x0303
)

const (
	maxPlaintext  = 16384        // maximum plaintext payload length
	maxCiphertext = 16384 + 2048 // maximum ciphertext payload length
	recordHeaderLen = 5          // record header length
	maxHandshake  = 65536        // maximum handshake we support (protocol max is 16 MB)

	minVersion = VersionTLS10
	maxVersion = VersionTLS12
)

// TLS record types.
type recordType uint8

const (
	recordTypeChangeCipherSpec recordType = 20
```

```
	recordTypeAlert           recordType = 21
	recordTypeHandshake       recordType = 22
	recordTypeApplicationData recordType = 23
)

// TLS handshake message types.
const (
	typeHelloRequest       uint8 = 0
	typeClientHello        uint8 = 1
	typeServerHello        uint8 = 2
	typeNewSessionTicket uint8 = 4
	typeCertificate        uint8 = 11
	typeServerKeyExchange uint8 = 12
	typeCertificateRequest uint8 = 13
	typeServerHelloDone uint8 = 14
	typeCertificateVerify uint8 = 15
	typeClientKeyExchange uint8 = 16
	typeFinished           uint8 = 20
	typeCertificateStatus uint8 = 22
	typeNextProtocol       uint8 = 67 // Not IANA assigned
)

// TLS compression types.
const (
	compressionNone uint8 = 0
)
```

5.3.2 Record Payload Protection

record 中携带的数据被压缩后，会经历 encryption 和 MAC functions 的处理. decryption functions 会反转此过程. record 中的 MAC 也包含了一个序列号，从而可以检测到消息是否消失、重复或者是额外的. 下面是 record 写的源码实现：

Listing 8: TLS 协议中如何写 record

```
// writeRecordLocked writes a TLS record with the given type and payload to the
// connection and updates the record layer state.
// c.out.Mutex <= L.
func (c *Conn) writeRecordLocked(typ recordType, data []byte) (int, error) {
	b := c.out.newBlock()
	defer c.out.freeBlock(b)

	var n int
	for len(data) > 0 {
		explicitIVLen := 0
		explicitIVIsSeq := false

		var cbc cbcMode
```

```
		if c.out.version >= VersionTLS11 {
			var ok bool
			if cbc, ok = c.out.cipher.(cbcMode); ok {
				explicitIVLen = cbc.BlockSize()
			}
		}
		if explicitIVLen == 0 {
			if _, ok := c.out.cipher.(cipher.AEAD); ok {
				explicitIVLen = 8
				// The AES-GCM construction in TLS has an
				// explicit nonce so that the nonce can be
				// random. However, the nonce is only 8 bytes
				// which is too small for a secure, random
				// nonce. Therefore we use the sequence number
				// as the nonce.
				explicitIVIsSeq = true
			}
		}
		m := len(data)
		if maxPayload := c.maxPayloadSizeForWrite(typ, explicitIVLen); m > maxPayload {
			m = maxPayload
		}
		b.resize(recordHeaderLen + explicitIVLen + m)
		b.data[0] = byte(typ)
		vers := c.vers
		if vers == 0 {
			// Some TLS servers fail if the record version is
			// greater than TLS 1.0 for the initial ClientHello.
			vers = VersionTLS10
		}
		b.data[1] = byte(vers >> 8)
		b.data[2] = byte(vers)
		b.data[3] = byte(m >> 8)
		b.data[4] = byte(m)
		if explicitIVLen > 0 {
			explicitIV := b.data[recordHeaderLen : recordHeaderLen+explicitIVLen]
			if explicitIVIsSeq {
				copy(explicitIV, c.out.seq[:])
			} else {
				if _, err := io.ReadFull(c.config.rand(), explicitIV); err != nil {
					return n, err
				}
			}
		}
		copy(b.data[recordHeaderLen+explicitIVLen:], data)
		c.out.encrypt(b, explicitIVLen)
		if _, err := c.write(b.data); err != nil {
			return n, err
```

```
            }
            n += m
            data = data[m:]
        }

        if typ == recordTypeChangeCipherSpec {
            if err := c.out.changeCipherSpec(); err != nil {
                return n, c.sendAlertLocked(err.(alert))
            }
        }

        return n, nil
}
```

Null or Standard Stream Cipher 注意 MAC 是在加密前计算得到的，stream ciphter 加密了包含 MAC 的 entire block.

CBC Block Cipher 对于分组加密（比如 3DES、AES），

IV The Initialization Vector (IV) SHOULD be chosen at random, and MUST be unpredictable. Note that in versions of TLS prior to 1.1, there was no IV field, and the last ciphertext block of the previous reord (the “CBC residue”) was used a the IV. This was changed to prevent the attacts described in CBCATT. For block ciphers, the IV length is of length 等于安全参数中的

```
record_iv_lenghth
```

, 亦等于

```
block_size
```