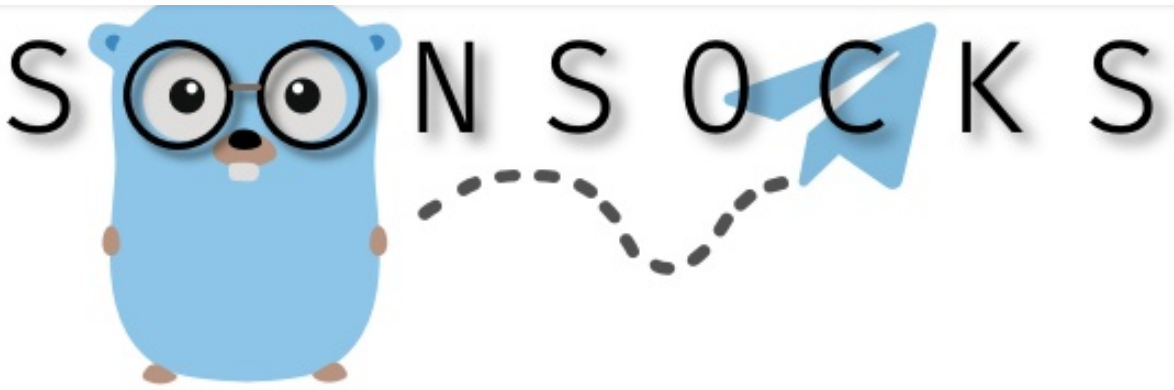


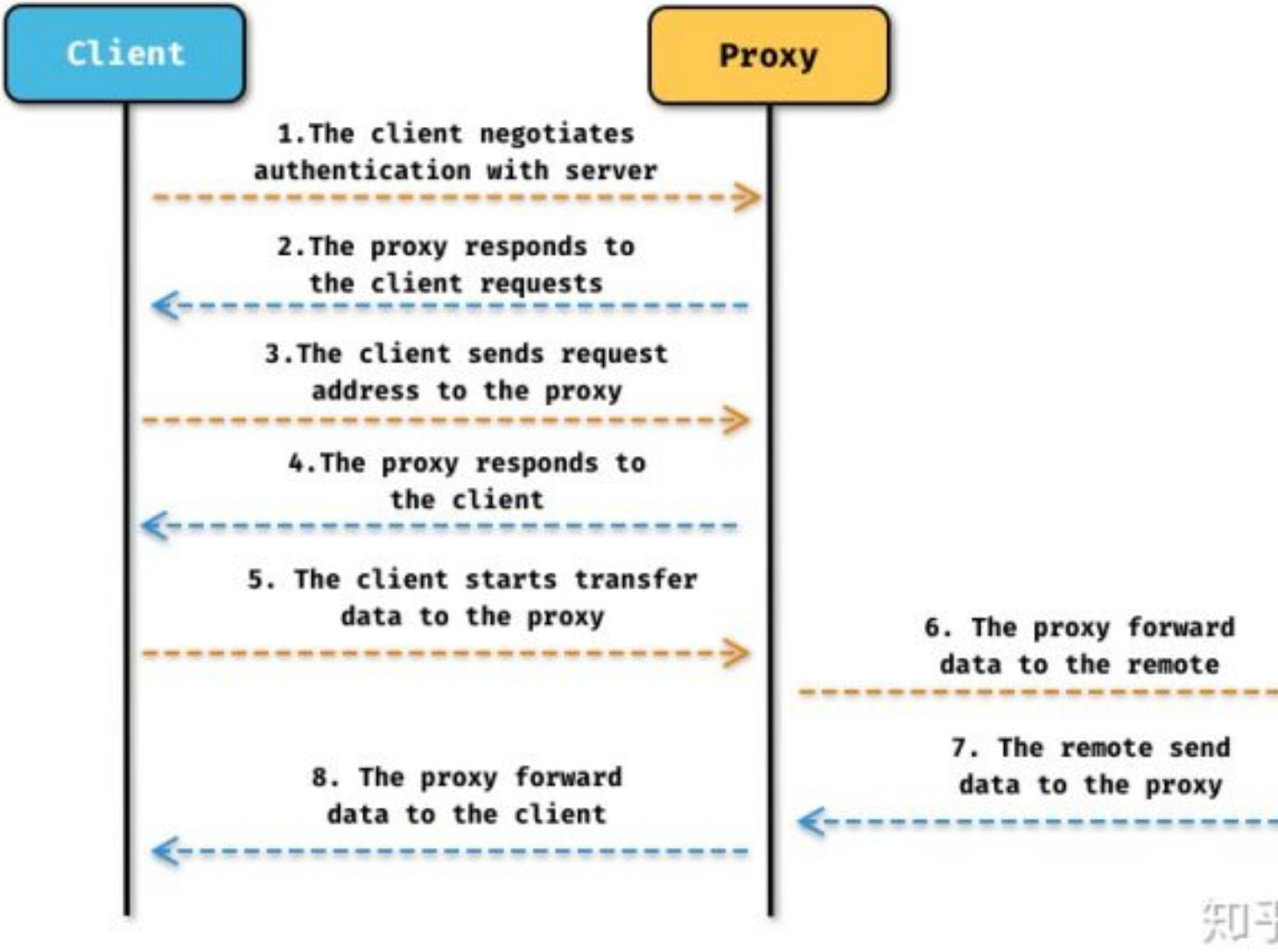
深入理解Socks5加密传输实现原理

yangsoon

> 声明，本文属于源码学习文章，主要学习golang相关的网络编程。

socks5协议

关于socks5协议，网上的讲解^1有很多，而且sock5协议的RFC 1928很短，包
展示了socks5客户端和代理采用无认证的方式(ss就是采用无认证的方式)进行
流程。主要是为了和socks加密的实现做对比。



主要流程分为socks5协商、建立连接和传输阶段:

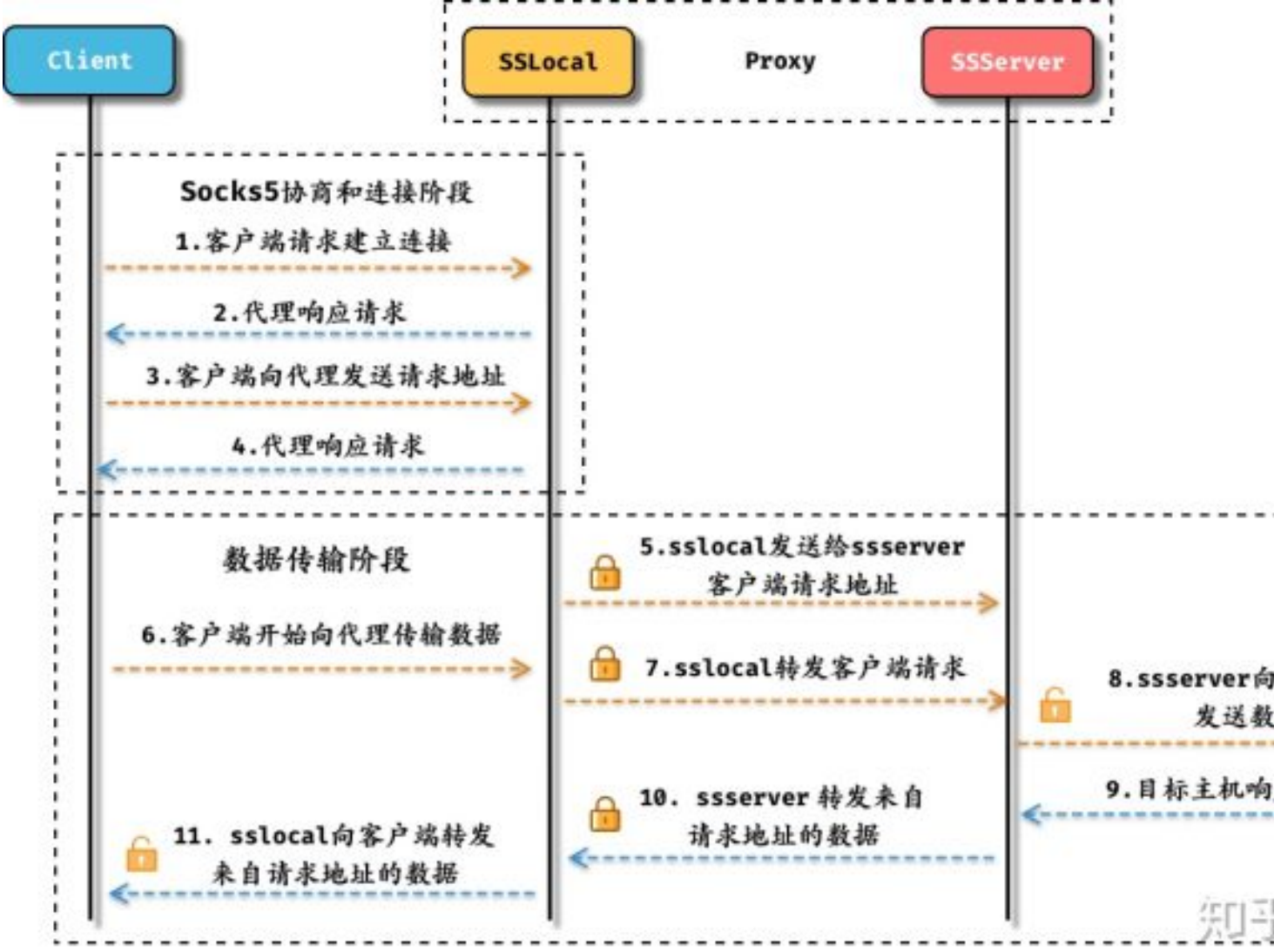
在协商阶段，客户端向代理发送请求协商认证方式，代理端告诉客户端采用开

在连接阶段，客户端向代理发送要请求的目的地址，代理端会响应客户端是否

SOCKS5 协议只负责建立连接，在完成握手阶段和建立连接之后，SOCKS5 服
转发了。客户端就开始传输数据，代理端将来自remote的响应返回给客户端。

socks加密实现如何使用socks5建立连接

socks5因为简单易用的特性被用来来实现代理功能，其实socks5只被用来建立
使用一系列加密算法加密数据以及使用代理服务器做转发来实现一些网站的访
站访问的整体流程。



和传统的socks5不同，Proxy划分为SSLocal和SSServer两个部分，其中SSLo
科学上网的主机上，SSServer部署在国外的主机上，其中SSLocal可以和SSe

主要流程也是分为协商连接和数据传输阶段，协商和连接阶段和传统的socks
传输阶段，SSLocal会像SSServer传输加密后的客户端请求访问的网站地址，
密数据之后解密确定将要代理访问的目标地址。SSLocal发送完请求地址之后
户端的请求数据加密转发给SSServer。开始数据传输。

SSLocal模块实现

SSLocal模块的实现严格按照上面的连接流程。首先SSLocal模块读取配置文
件中指定的端口，并监听来自客户端的请求，每接收到一个请求就开启一个g
的数据请求和响应。

```
func main() {
    var configPath string
    flag.StringVar(&configPath, "c", "config.json", "json file w
    flag.Parse()

    var err error
    config, err = ss.ParseConfig(configPath)
    if err != nil {
        ss.Logger.Fatalf("parse %s failed %v \n", configPath, er
    }
    ss.Logger.Printf("SSLocal is running at %v\n", config.LocalA
    ss.Logger.Printf("config info: \n"+
        "--------------------------------\n"+
        "LocalAddr: %v\n"+
        "ServerAddr: %v\n"+
        "Method: %v\n"+
        "--------------------------------\n",
        config.LocalAddr,
        config.ServerAddr,
        config.Method)

    l, err := net.Listen("tcp", config.LocalAddr)
    if err != nil {
        ss.Logger.Printf("SSLocal listen faild %v\n", err)
        panic(err)
    }

    for {
        conn, err := l.Accept()
        if err != nil {
            ss.Logger.Printf("SSLocal accept client error: %v\n"
            continue
        }

        go handleConnection(conn)
    }
}
```

handleConnection函数实现这样的功能：首先完成和客户端的socks5连接，在中完成，建立连接之后，和SSServer建立tcp连接，并传输客户端想要访问的就充当一个转发者的角色，将来自客户端的请求加密转发给SSServer，同时将应请求解密转发给客户端。

```
func handleConnection(conn net.Conn) {
    rawaddr, host, err := ss.HandleShake(conn)
    if err != nil {
        ss.Logger.Printf("socks negotiate host %s error: %v\n",
        return
    }

    cipher, err := ss.NewCipher(config.Method, config.Password)

    if err != nil {
        ss.Logger.Printf("create cipher error: %v\n", err)
        return
    }

    serverCConn, err := ss.DialWithCipher(config.ServerAddr, cip
    if err != nil {
        ss.Logger.Printf("connect to server %s error: %v\n", con
        return
    }

    ss.Logger.Printf("connecting to server %v (request host %v)\
    _, err = serverCConn.Write(rawaddr)
    if err != nil {
        ss.Logger.Printf("write to server %s error: %v\n", confi
    }

    go func() {
        defer conn.Close()
        _, err := ss.CopyBuffer(conn, serverCConn)
        if err != nil {
```

```
            ss.Logger.Printf("connecting to %v error: %v\n", hos
        }
    }()

    _, err = ss.CopyBuffer(serverCConn, conn)
    if err != nil {
        ss.Logger.Printf("connecting to %v error: %v", host, err
    }
    serverCConn.Close()
}
```

函数HandleShake完成了客户端和SSLocal的连接的建立，分为4个步骤和soc
一对应。

```
func HandleShake(conn net.Conn) (rawaddr []byte, host string, er

    rawaddr = []byte{}
    host = ""

    // 1. get pkg from client
    if _, err = extractNegotiation(conn); err != nil {
        return
    }
    Logger.Println("get conn from client")

    // 2. reply to client build connect
    if err = replyNegotiation(conn); err != nil {
        return
    }
    Logger.Println("reply to client")

    // 3. get request pkg from client
    var socks5r Socks5Request
    if socks5r, err = extractRequest(conn); err != nil {
        return
    }
```

```
    Logger.Printf("request %s\n", socks5r.Host)

    // 4. reply to client
    if err = replyRequest(conn); err != nil {
        return
    }
    Logger.Println("reply to client request")

    rawaddr = socks5r.RawAddr
    host = socks5r.Host
    return
}
```

SSServer模块实现

该模块的实现逻辑和SSLocal基本相同，可以直接看代码实现[^2] 。其中需要注
在传输数据的时候，首先SSlocal会发送一个数据包，包括接下来client将要请
个目标地址是裁剪自socks5协商连接阶段，client发送给proxy请求目标地址的
址字段，因此，SSServer还要需要先对目标数据包做解析。

数据加密模块

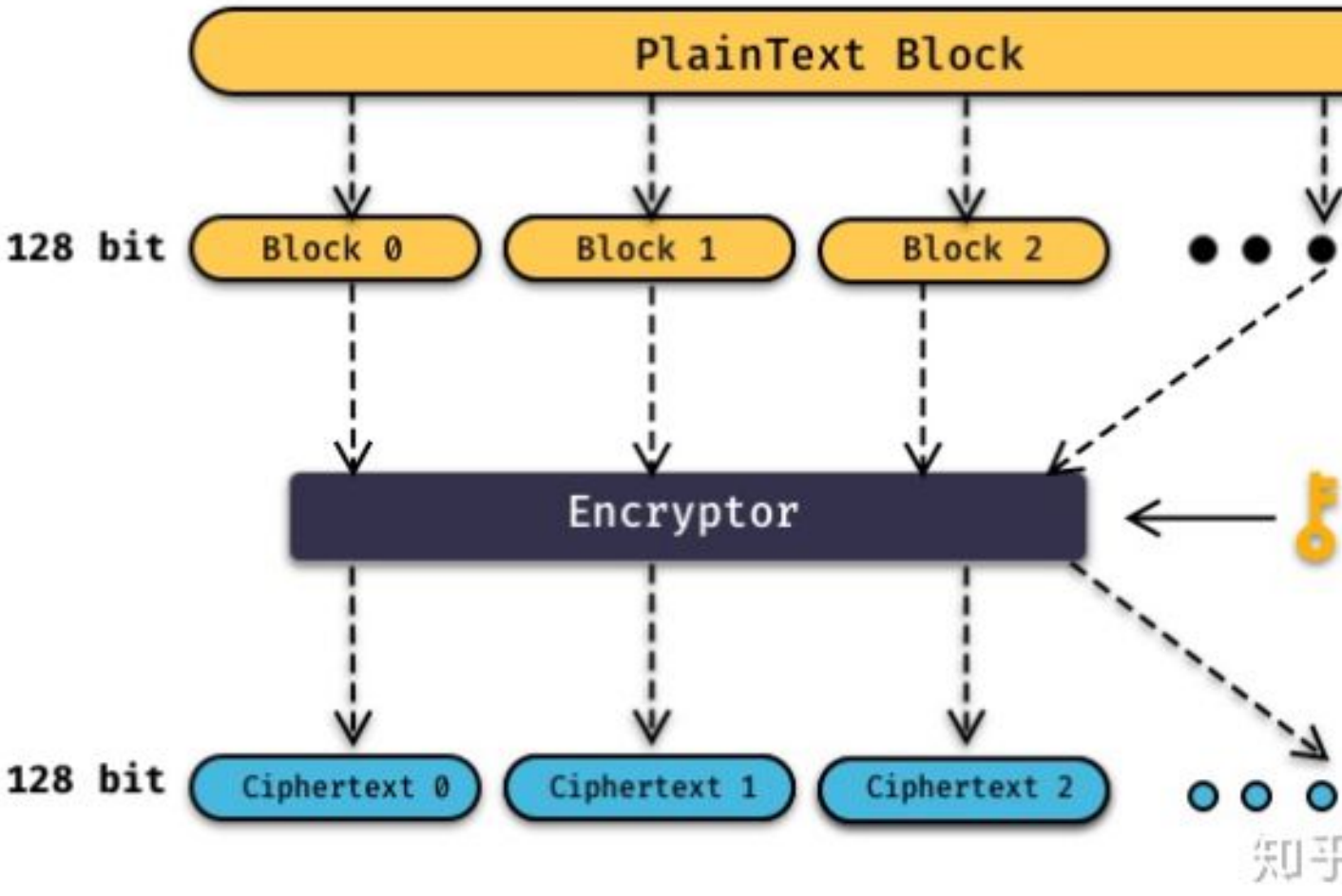
支持rc4md和aes-128-cfb和aes-256-cfb三种加密算法，本部分主要讲解代理
密算法进行数据加密传输。其中对aes原理只是简单一提，具体的加密算法请

高级加密标准 AES

AES 密码学中的高级加密标准(Advanced Encryption Standard，AES)又称高级
加密法。加密算法分为对称加密和非对称加密，两者的区别在于加密和解密所
一个密钥，其中Rijndael加密法属于对称加密算法。

AES的基本要求是，采用对称分组密码体制，密钥长度的最少支持为128、19
必须为128比特，密钥长度可以是128比特、192比特、256比特中的任意一个

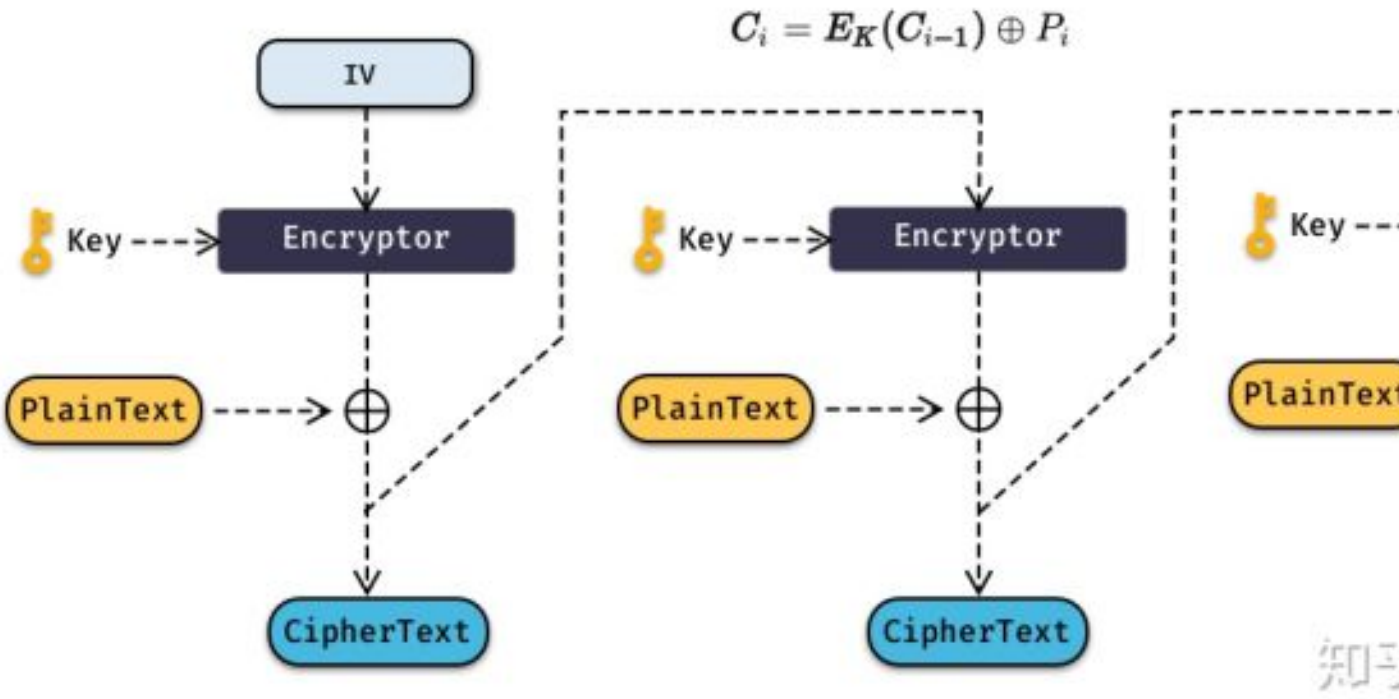
钥长度不足时，会补齐）。aes-128-cfb和aes-256-cfb的不同大家也能看出来
度不同。



AES CFB模式

CFB的加密过程分成两部分，先将前一段加密得到的密文加密，然后加密后的
或。在对第一个块进行加密使用的 IV 即初始向量（Initialization Vector）它的
盐”有些类似，目的是防止同样的明文块始终加密成同样的密文块。

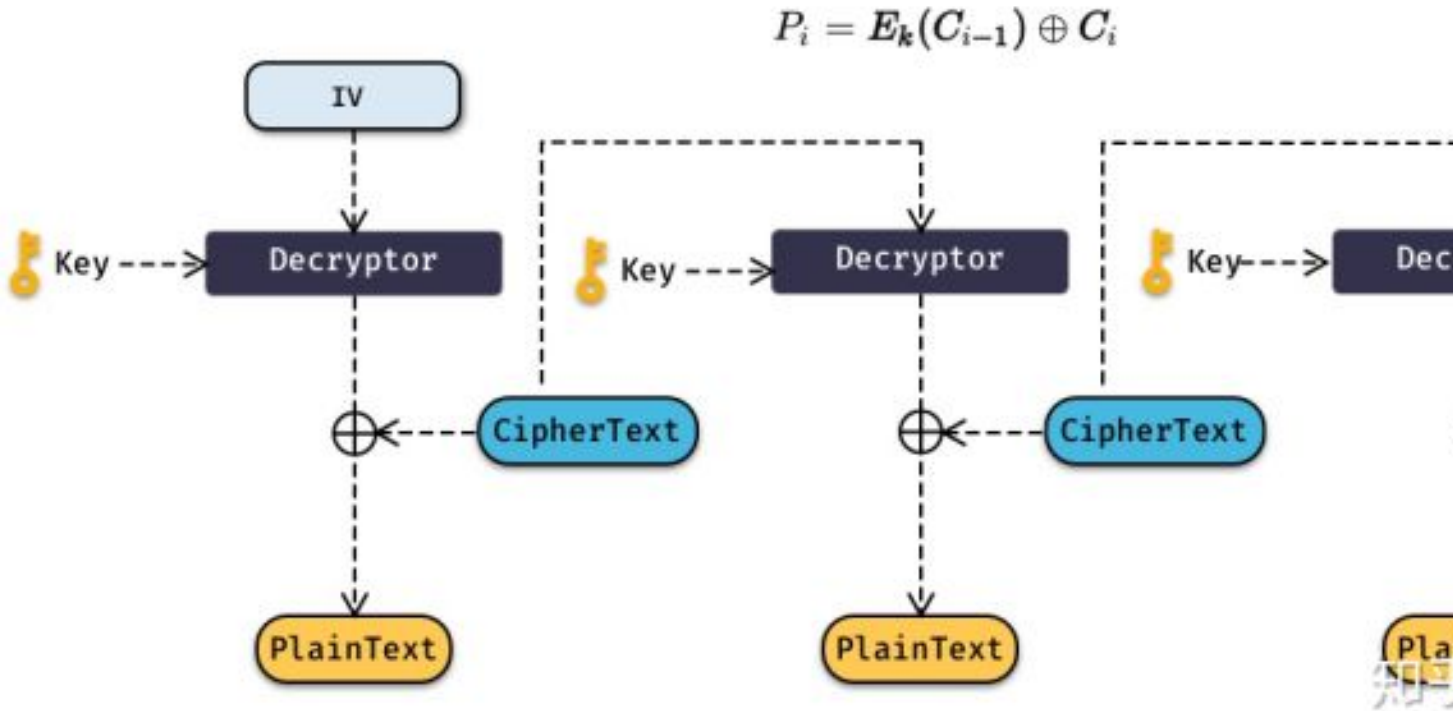
Encryption



CFB的解密过程几乎就是颠倒的CBC的加密过程。

> 图中虽然画的是解密器，但实际上解密器进行的操作仍然是使用和加密过程
> 做加密处理。通过图中的公式可以看到这一点。该部分的图和公式来自于

Decryption



关于AES加密部分的代码实现，golang官方也提供了样例。soonsock中加密解码类似，就不再赘述。

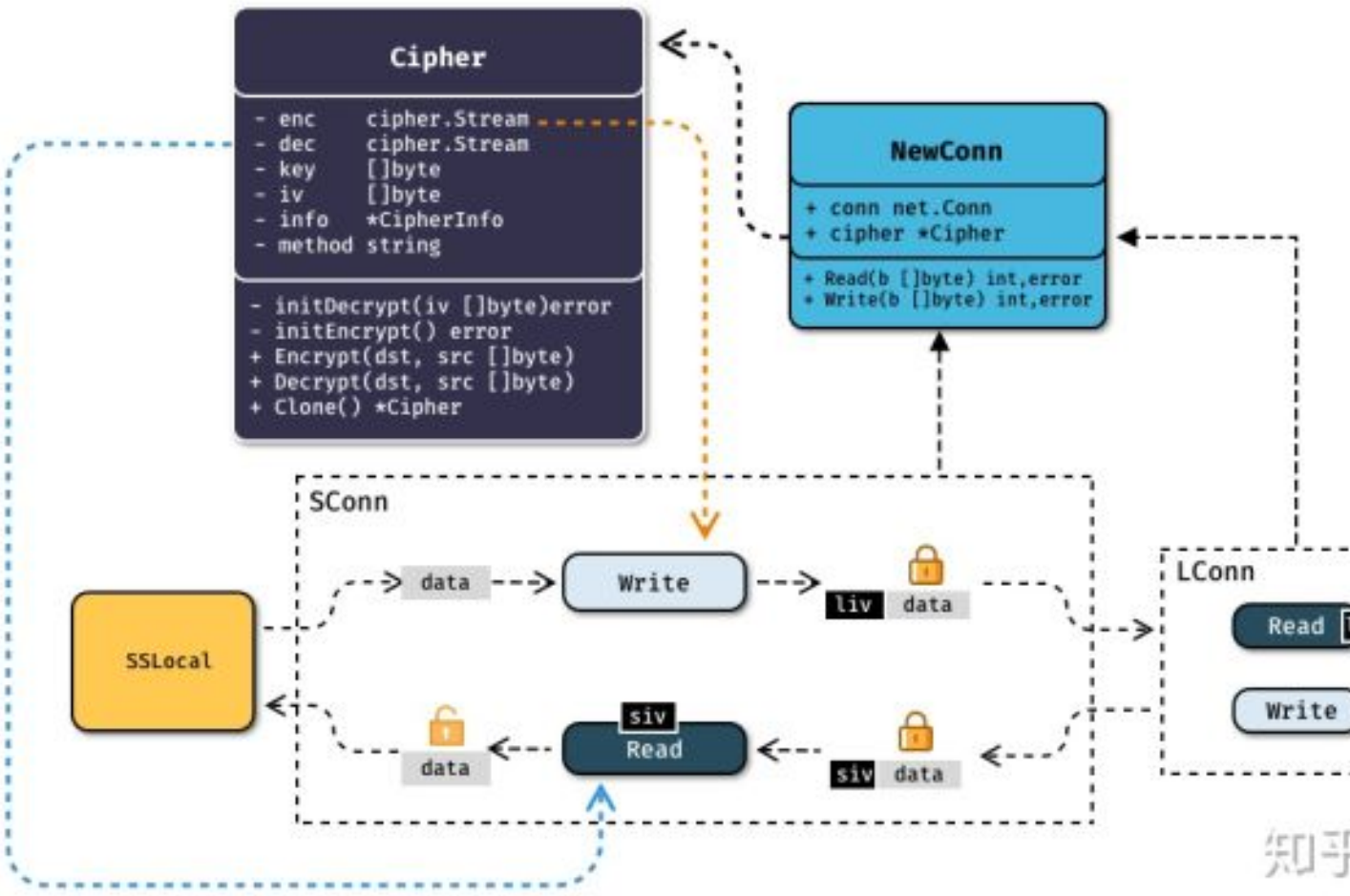
```
func ExampleNewCFBEncrypter() {
    // Load your secret key from a safe place and reuse it acros
    // NewCipher calls. (Obviously don't use this example key fo
    // real.) If you want to convert a passphrase to a key, use
    // package like bcrypt or scrypt.
    key, _ := hex.DecodeString("6368616e676520746869732070617373
    plaintext := []byte("some plaintext")

    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }

    // The IV needs to be unique, but not secure. Therefore it's
    // include it at the beginning of the ciphertext.
    ciphertext := make([]byte, aes.BlockSize+len(plaintext))
    iv := ciphertext[:aes.BlockSize]
    if _, err := io.ReadFull(rand.Reader, iv); err != nil {
        panic(err)
    }

    stream := cipher.NewCFBEncrypter(block, iv)
    stream.XORKeyStream(ciphertext[aes.BlockSize:], plaintext)

    // It's important to remember that ciphertexts must be authe
    // (i.e. by using crypto/hmac) as well as being encrypted in
    // be secure.
    fmt.Printf("%x\n", ciphertext)
}

func ExampleNewCFBDecrypter() {
    // Load your secret key from a safe place and reuse it acros
    // NewCipher calls. (Obviously don't use this example key fo
```

```
    // real.) If you want to convert a passphrase to a key, use
    // package like bcrypt or scrypt.
    key, _ := hex.DecodeString("6368616e676520746869732070617373
    ciphertext, _ := hex.DecodeString("7dd015f06bec7f1b8f6559dac

    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }

    // The IV needs to be unique, but not secure. Therefore it's
    // include it at the beginning of the ciphertext.
    if len(ciphertext) < aes.BlockSize {
        panic("ciphertext too short")
    }
    iv := ciphertext[:aes.BlockSize]
    ciphertext = ciphertext[aes.BlockSize:]

    stream := cipher.NewCFBDecrypter(block, iv)

    // XORKeyStream can work in-place if the two arguments are t
    stream.XORKeyStream(ciphertext, ciphertext)
    fmt.Printf("%s", ciphertext)
    // Output: some plaintext
}
```

SS加密解密传输流程

下图描述了SSLocal和SSServer之间进行数据加密传输的过程，首先SSServer
定端口，SSLocal连接到SSServer得到了连接conn并将连接包装为对象NewC
含有net.Conn对象和Cipher对象，并且实现了net.Conn的Read和Write接口。
写入数据的时候，使用Cipher进行解密和加密。

同理，当SSServer获取到来自SSLocal的连接的时候，也会将conn包装成New
对数据进行加密和解密。



SSLocal和SSServer建立连接的时候，SSLocal先向连接中写入数据，首次写入成初始向量iv并用来初始化加密器enc，在写入数据的时候，会对要发送的请求加密数据前附加上初始向量iv，因为解密过程需要使用相同的iv进行解密，所以包前附上初始向量iv。

SSServer监听到来自SSLocal的连接之后，同样会将conn包装成LConn，SSS始向量iv的数据包之后，会使用iv来初始化自己的解密器，以便解密数据包。一次使用LConn向SSLocal写入数据的时候，也会初始化自己的加密器，将对数据包中。SSLocal获取到带有iv的数据包并发现自己没有相应的解密器，所以己的解密器，解密数据包。

当SSLocal和SSServer都初始化了自己加密解密器之后，接下来发送的数据包向量了。

```
func (cc *CConn) Read(b []byte) (int, error){
    if cc.dec == nil {
        iv := make([]byte, cc.info.ivLen)
```

```
        if _, err := io.ReadFull(cc.Conn, iv); err != nil {
            return 0, err
        }
        if err := cc.initDecrypt(iv); err!=nil {
            return 0, err
        }

        if len(cc.iv) == 0 {
            cc.iv = iv
        }
    }
    encryptData := make([]byte, len(b))
    n, err := cc.Conn.Read(encryptData)
    if n > 0 {
        cc.Decrypt(b[0:n], encryptData[0:n])
    }
    return n, err
}

func (cc *CConn) Write(b []byte) (int, error) {
    var iv []byte
    if cc.enc == nil {
        if err := cc.initEncrypt(); err != nil {
            return 0, err
        }
        if len(cc.iv) == 0 {
            return 0, errors.New("get iv error")
        }
        iv = cc.iv
    }

    encryptData := make([]byte, len(iv)+len(b))
    if len(iv) > 0 {
        copy(encryptData, iv)
    }

    cc.Encrypt(encryptData[len(iv):], b)
```

```
    n, err := cc.Conn.Write(encryptData)
    return n, err
}
```

原文地址

yangsoon.github.io

编辑于 2020-01-10

socks代理 网络传输 Golang 最佳实践

2 条评论

写下你的评论...

赞同 24 2 条评论

想来知