CSCI968 Advance Network Security: Assignment #1

Mei Wangzhihui 2019124044

Problem 1

One-time password SecureID system
Google Authenticator. User enter a initializing code to generator a one-time password changing each period. It use the AES-128 algorithm.
S/Key System The authentication to Unix-like operating system replacing long-term password. A user's real password is combined in an offline device with a short set of characters and a decrementing counter to form a single-use password. Because each password is only used once, they are useless to password sniffers. After password generation, the user has a sheet of paper with n passwords on it. It use the Random Oracle.
Challenge response protocol Challenge–response authentication can help solve the problem of exchanging session keys for encryption. Using a key derivation function, the challenge value and the secret may be combined to generate an unpredictable encryption key for the session. This is particularly effective against a man-in-the-middle attack, because the attacker will not be able to derive the session key from the challenge without knowing the secret, and therefore will not be able to decrypt the data stream.

Problem 2

```go
package main

import (
    "crypto/elliptic"
    "crypto/rand"
    "crypto/sha256"
    "fmt"
    "io"
    "math/big"
)

type PublicKey struct {
    elliptic.Curve
    X, Y *big.Int
}

type PrivateKey struct {
    PublicKey
    D *big.Int
}

var one = new(big.Int).SetInt64(1)

func randFieldElement(c elliptic.Curve, rand io.Reader) (k *big.Int, err error) {
    params := c.Params()
    b := make([]byte, params.BitSize/8+8)
    _, err = io.ReadFull(rand, b)
    if err != nil {
        return
    }

    k = new(big.Int).SetBytes(b)
    n := new(big.Int).Sub(params.N, one)
    k.Mod(k, n)
    k.Add(k, one)
    return
}

func GenerateKey(c elliptic.Curve, rand io.Reader) (*PrivateKey, error) {
    k, err := randFieldElement(c, rand)
    if err != nil {
        return nil, err
    }
    priv := new(PrivateKey)
    priv.PublicKey.Curve = c
    priv.D = k
    priv.PublicKey.X, priv.PublicKey.Y = c.ScalarBaseMult(k.Bytes())
    return priv, nil
```

Problem 2 continued on next page...

```go
}

func hashIt(x, y *big.Int, message string) *big.Int {
    tempHash := sha256.Sum256([]byte(message))
    hash := tempHash[:]
    tempInput := make([]byte, len(hash))
    tempInput = append(tempInput, hash...)
    tempInput = append(tempInput, x.Bytes()...)
    tempInput = append(tempInput, y.Bytes()...)
    temp := sha256.Sum256(tempInput)
    cTemp := temp[:]
    return new(big.Int).SetBytes(cTemp)
}

func Sign(rand io.Reader, priv *PrivateKey, message string) (msg string, x, y, a *big.Int, err error) {

    var k *big.Int

    for {

        k, err = randFieldElement(priv.PublicKey.Curve, rand)
        if err == nil {
            break
        }
    }

    x, y = priv.Curve.ScalarBaseMult(k.Bytes())
    cV := hashIt(x, y, message)
    temp := new(big.Int).Mul(priv.D, cV)
    a = new(big.Int).Add(k, temp)
    msg = message
    return
}

func Verify(pub *PublicKey, message string, x, y, a *big.Int) bool {

    cV := hashIt(x, y, message)
    x1, y1 := pub.Curve.ScalarMult(pub.X, pub.Y, cV.Bytes())
    x, y = pub.Curve.Add(x1, y1, x, y)
    x2, y2 := pub.Curve.ScalarBaseMult(a.Bytes())
    return (x.Cmp(x2) == 0 && y.Cmp(y2) == 0)
}

func Sign_opt(rand io.Reader, priv *PrivateKey, message string) (msg string, cv, a *big.Int, err error)
      {

    var k *big.Int
    for {

        k, err = randFieldElement(priv.PublicKey.Curve, rand)
        if err == nil {
            break
        }
    }

    x, y := priv.Curve.ScalarBaseMult(k.Bytes())
    cv = hashIt(x, y, message)
    temp := new(big.Int).Mul(priv.D, cv)
    a = new(big.Int).Add(k, temp)
    msg = message
    return
}

func Verify_opt(pub *PublicKey, message string, cV, a *big.Int) bool {
    x, y := pub.Curve.ScalarBaseMult(a.Bytes())
    x1, y1 := pub.Curve.ScalarMult(pub.X, pub.Y, cV.Bytes())
    negY1 := new(big.Int).Neg(y1)
    x2, y2 := pub.Curve.Add(x, y, x1, negY1)
    x3, y3 := pub.Curve.Add(x2, y2, x1, y1)
    return (x3.Cmp(x) == 0 && y3.Cmp(y) == 0)
}

func main() {
```

```
    privateKey, err := GenerateKey(elliptic.P256(), rand.Reader)
    if err != nil {
        panic(err)
    }
    msg := "CSCI468/968AdvancedNetworkSecurity,Spring2020"

    message, x, y, a, err := Sign(rand.Reader, privateKey, msg)
    if err != nil {
        panic(err)
    }

    fmt.Println("message:", message)
    fmt.Println("signature: u_t:", x, y)
    fmt.Println("signature: a_z:", a)

    valid := Verify(&privateKey.PublicKey, message, x, y, a)
    fmt.Println("signature verified:", valid)

    fmt.Println(" optimized versions:")

    message1, cv1, a1, err1 := Sign_opt(rand.Reader, privateKey, msg)
    if err1 != nil {
        panic(err)
    }

    fmt.Println("message:", message1)
    fmt.Println("signature: c:", cv1)
    fmt.Println("signature: a_z:", a1)
    valid1 := Verify_opt(&privateKey.PublicKey, message1, cv1, a1)
    fmt.Println("signature verified:", valid1)
}
```

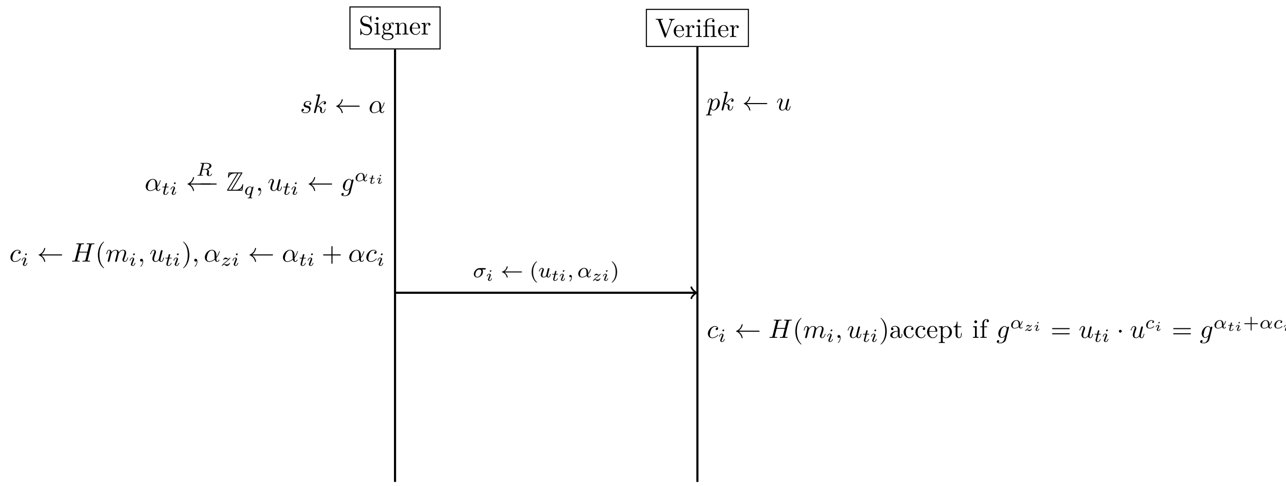
Output

```
message: CSCI468/968AdvancedNetworkSecurity,Spring2020
signature: u_t: 43068305453503561280361870469765841078369704546103943374318459436367629179869
     12359168923261615701252974857854824377642988124531973618366557065204705787927
signature: a_z: 118113457633065229502460821696906849747556761002336714124847642078416851115
48021326205279399598472135606629328989136237936188537873579005908404438188124092
signature verified: true
 optimized versions:
message: CSCI468/968AdvancedNetworkSecurity,Spring2020
signature: c: 83530092752711663050226295389097351853301230250199467408878628416381598386762
signature: a_z: 87804269649572365907801475156995763115439686770266306897454623572750 74160
17615121901592655556616440032132653679278026027408132113773707484264445818883 9833
signature verified: true
```

Problem 3

Schnorr signature scheme

$\alpha \xleftarrow{R} \mathbb{Z}_q,\ u \leftarrow g^\alpha$



When signing m_0, $u_{t0} = g^{\alpha_{t0}}, \alpha_{z0} = \alpha_{t0} + \alpha c_0, c_0 = H(m_0, u_{t0})$.
When signing m_1, $u_{t1} = g^{a\alpha_{t0}+b} = u_{t0}^a \cdot g^b, \alpha_{z1} = a \cdot \alpha_{t0} + b + \alpha c_1, c1 = H(m_1, u_{t1})$.
If adversary obtain (m_0, σ_0) and (m_1, σ_1), he can get $\alpha_{z1} - a\alpha_{z0} = (c_1 - ac_0)\alpha + b$ where c_0, c_1, a, b is clear for adversary.

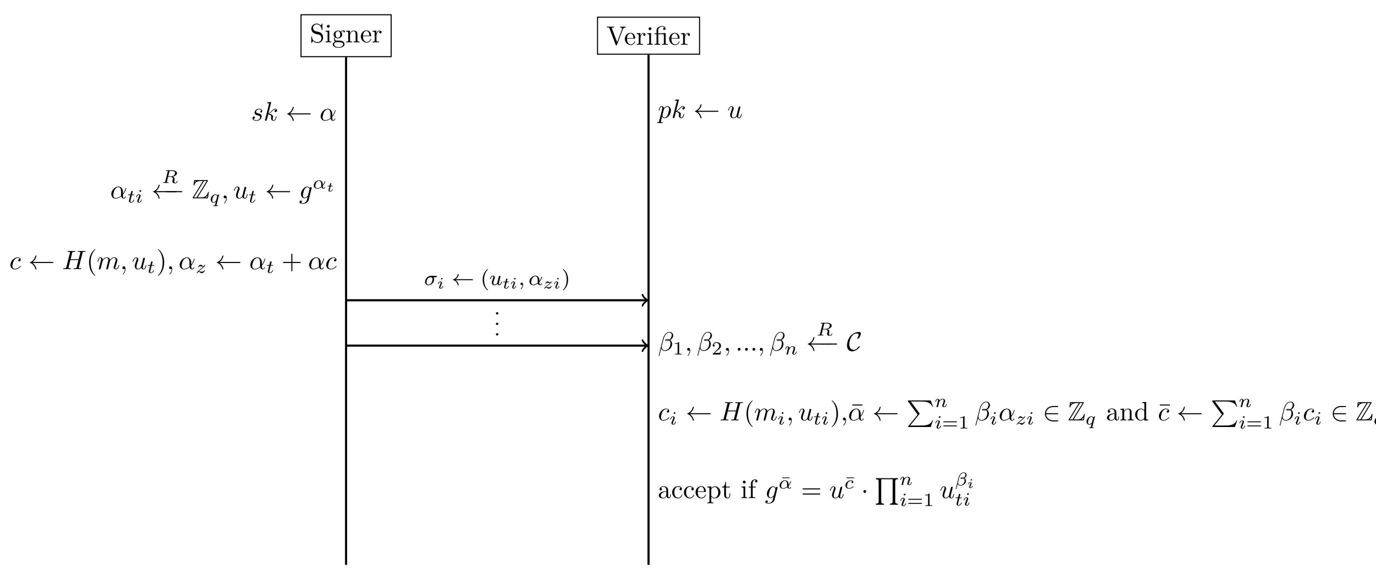
$$Adv_{sk} = P(c_1 - ac_0 \neq 0)$$

so the probability of obtaining secret key α is not negligible.

Problem 4

Batch Schnorr signature scheme

$\alpha \xleftarrow{R} \mathbb{Z}_q,\ u \leftarrow g^\alpha$



Problem 4 continued on next page...

We get

$$\begin{aligned} g^{\bar{\alpha}} &= g^{\sum_{i=1}^{n} \beta_i \alpha_{zi}} \\ &= \prod_{i=1}^{n} (g^{\alpha_{zi}})^{\beta_i} \\ &= g^{\sum_{i=1}^{n} \alpha_{zi} \beta_i} \end{aligned} \tag{1}$$

$$\begin{aligned} u^{\bar{c}} \cdot \prod_{i=1}^{n} u_{ti}^{\beta_i} &= g^{\alpha \cdot \sum_{i=1}^{n} \beta_i c_i} \cdot g^{\sum_{i=1}^{n} \alpha_{ti} \beta_i} \\ &= g^{\sum_{i=1}^{n} (\alpha_{ti} + \alpha c_i) \beta_i} \\ &= \prod_{i=1}^{n} (g^{\alpha_{ti} + \alpha c_i})^{\beta_i} \\ &= g^{\sum_{i=1}^{n} (\alpha_{ti} + \alpha c_i) \beta_i} \end{aligned} \tag{2}$$

Suppose that β_i is not used, then the $g^{\bar{\alpha}} = g^{\sum_{i=1}^{n} \alpha_{zi}}$ and $u^{\bar{c}} \cdot \prod_{i=1}^{n} u_{ti}^{\beta_i} = g^{\sum_{i=1}^{n} (\alpha_{ti} + \alpha c_i)}$

$$P(\exists \alpha_{zi} \neq \alpha_{ti} + \alpha c_i, g^{\sum_{i=1}^{n} \alpha_{zi}} = g^{\sum_{i=1}^{n} (\alpha_{ti} + \alpha c_i)})$$

is not negligible. while

$$P(\exists \alpha_{zi} \neq \alpha_{ti} + \alpha c_i, g^{\sum_{i=1}^{n} \alpha_{zi} \beta_i} = g^{\sum_{i=1}^{n} (\alpha_{ti} + \alpha c_i) \beta_i})$$

is negligible. so advantage is

$$Adv_{BSV} = P(\exists \alpha_{zi} \neq \alpha_{ti} + \alpha c_i, g^{\sum_{i=1}^{n} \alpha_{zi} \beta_i} = g^{\sum_{i=1}^{n} (\alpha_{ti} + \alpha c_i) \beta_i}) P$$