Go Lang Cryptography for Developers

Anish Nath

GoLang Cryptography for Developer

Anish Nath

This book is for sale at http://leanpub.com/cryptog

This version was published on 2018-12-05

I would like to dedicate this book to my 8-year old son Arjun Nath , Grandfather Sri Rajeshwar Prasad; wife Manisha Prasad; mother Indu Sinha; and all my family members (my father Anil Kumar Sinha; chote papa Sunil Kumar Sinha; choti mummy: Poonam Sinha; and friends). Without them, this would not have been possible.

I would also thanks to Cisco where I work, I got most of the learning from there, and not least the opensource community.

I would also thanks to reader who spend their money and time to read this book, without them there is no Inspiration.

Contents

Argon2

Argon2 is a key derivation function[1] that was selected as the winner of the Password Hashing Competition[2] in July 2015. It was designed by Alex Biryukov[3], Daniel Dinu, and Dmitry Khovratovich from the University of Luxembourg[4].

There are two flavors of Argon2 – **Argon2d** and **Argon2i**. The former one uses data-dependent memory access to thwart tradeoff attacks. However, this makes it vulnerable for side-channel attacks, so Argon2d is **recommended** primarily for cryptocurrencies and backend servers.

Argon2i uses data-independent memory access, which is recommended for password hashing and password-based key derivation

Go Package Argon2[5] implements the key derivation function Argon2.

If you aren't sure which function you need, use `Argon2id (IDKey)`

Argon2 has two types of **inputs**: **primary** inputs and **secondary** inputs, or parameters. Primary inputs are message P and nonce S, which are password and salt, respectively, for the password hashing

- Message **P** may have any length from 0 to 232 – 1 bytes;
- Nonce **S** may have any length from 8 to 2 ³² – 1 bytes
- Degree of parallelism **p** : - integer value from 1 to 2²⁴ – 1.
- Tag length **τ** : integer value from 4 to 2 ³² – 1.
- Memory size **m**
- Number of iterations **t** : 1 to 2³² – 1.
- Version number **v** : - 0x13
- Secret value **K** :
- Associated data **X**
- Type **y** of Argon2: 0 for Argon2d, 1 for Argon2i, 2 for Argon2id

Argon2 supported functions

- Go `IDKey` derives a key from the password, salt, and cost parameters using **Argon2id** returning a byte slice of length keyLen that can be used as cryptographic key..
- Go `Key` derives a key from the password, salt, and cost parameters using **Argon2i** returning a byte slice of length keyLen that can be used as cryptographic key

[1] https://en.wikipedia.org/wiki/Key_derivation_function
[2] https://en.wikipedia.org/wiki/Password_Hashing_Competition
[3] https://en.wikipedia.org/wiki/Alex_Biryukov
[4] https://en.wikipedia.org/wiki/University_of_Luxembourg
[5] https://godoc.org/golang.org/x/crypto/argon2

```
func IDKey(password, salt []byte, time, memory uint32, threads uint8, keyLen uint32)\
 []byte
func Key(password, salt []byte, time, memory uint32, threads uint8, keyLen uint32) [\
]byte
```

Argon2d

The draft RFC recommends `time=3`, and `memory=32*1024` is a sensible number, If using that amount of memory (32 MB) is not possible in some contexts then the time parameter can be increased to compensate.

The number of threads can be adjusted to the number of available CPUs. The cost parameters should be increased as memory latency and CPU parallelism increases

For example, you can get a derived key for e.g. AES-256 (which needs a 32-byte key) by doing:

```
package main

import (
  "crypto/rand"
  "encoding/base64"
  "fmt"
  "golang.org/x/crypto/argon2"
  )

func main() {

   P := "my secret Password 8gwifi.org"
   m := 32*1024
   keyLen :=32

  // Generatign Random Salt of 8 byte
  salt := make([]byte, 8)
   if _, err := rand.Read(salt); err != nil {
      panic(err)
   }

   key := argon2.Key([]byte(P), salt, 3, uint32(m), 4, uint32(keyLen))
   fmt.Println("Argon2 Derive Key of length 32 Base64 Value: ", base64.StdEncoding.E\
ncodeToString(key))
}
```

The output : It will vary when run again, as for security best practice a **random** salt value is used.

```
$ go run argon2.go
Argon2 Derive Key of length 32 Base64 Value:  jjVeKTAKLBe49NN3rg8lWwq5EqVXPTlK3smog3\
zHLgY=
```

Argon2id

The draft RFC recommends `time=1`, and `memory=64*1024` is a sensible number, If using that amount of memory (64 MB) is not possible in some contexts then the time parameter can be increased to compensate.

For example, you can get a derived key for e.g. AES-256 (which needs a 32-byte key) by doing:

```
package main

import (
   "crypto/rand"
   "encoding/base64"
   "fmt"
   "golang.org/x/crypto/argon2")


func main() {

  P := "my secret Password 8gwifi.org"
  m := 64*1024
  keyLen :=32

  // Generatign Random Salt of 8 byte
  salt := make([]byte, 8)
  if _, err := rand.Read(salt); err != nil {
      panic(err)
   }
   key := argon2.IDKey([]byte(P), salt, 1, uint32(m), 4, uint32(keyLen))
   fmt.Println("Argon2 IDkey of length 32 Base64 Value: ", base64.StdEncoding.Encode\
ToString(key))

}
```

Output: It will vary when run again, as for security best practice a **random** salt value is used.

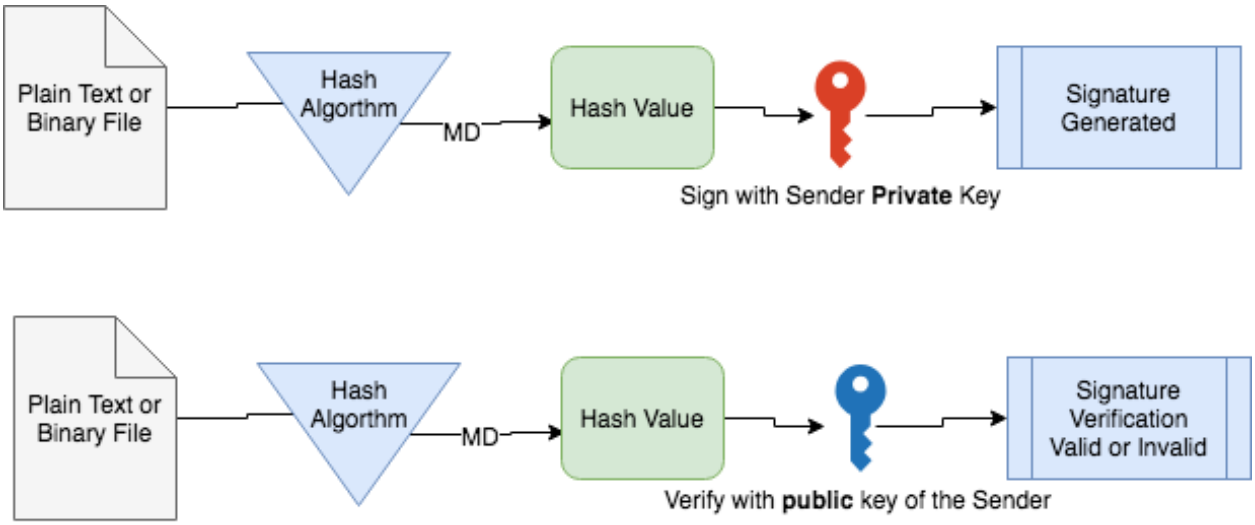
```
$ go run argon2id.go
Argon2 IDkey of length 32 Base64 Value:  boUwmz3xctikIU+y7yt+v6qWShhdUziqaSwML8EkQhE=
```

DSA

The **Digital Signature Algorithm** (**DSA**) is a Federal Information Processing Standard[6] for **digital signatures**[7], based on the mathematical concept of modular exponentiations[8] and the discrete logarithm problem[9].

DSA is a variant on the ElGamal and Schnorr algorithms creates a 320 bit signature, but with 512-1024 bit security again rests on difficulty of computing discrete logarithms has been quite widely accepted.

The digital signature can be used to provide assurance that the claimed signatory signed the information.



Digital Signature Process

A digital signature algorithm includes a **signature generation** process and a **signature verification** process. A signatory uses the generation process to generate a digital signature on data; a verifier uses the verification process to verify the authenticity of the signature. Each signatory has a public and private key and is the owner of that key pair.

Go standard library dsa[10] implements the Digital Signature Algorithm, as defined in FIPS 186-3.

DSA KeyGen, Sign & Verify

`GenerateKey` GenerateKey generates a public and private key pair.

[6] https://en.wikipedia.org/wiki/Federal_Information_Processing_Standards
[7] https://en.wikipedia.org/wiki/Digital_signature
[8] https://en.wikipedia.org/wiki/Modular_exponentiation
[9] https://en.wikipedia.org/wiki/Discrete_logarithm
[10] https://godoc.org/crypto/dsa

Go Lang function `GenerateKey` .

```
func GenerateKey(priv *PrivateKey, rand io.Reader) error
```

`GenerateParameters` puts a random, valid set of DSA parameters into params.

```
func GenerateParameters(params *Parameters, rand io.Reader, sizes ParameterSizes) er\
ror
```

Valid Params

- L1024N160 (L=1024, N=160)
- L2048N224 (L=2048, N=224)
- L2048N256 (L=2048, N=256)
- L3072N256 (L=3072, N=256)

`Sign` signs an arbitrary length hash (which should be the result of hashing a larger message) using the private key, priv. It returns the signature as a pair of integers

```
func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r, s *big.Int, err error)
```

`Verify` verifies the signature in r, s of hash using the public key, pub. It reports whether the signature is valid.

```
func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
```

for example the following go snippet will generate DSA key pair of parameter L1024N160 and then perform message signature generation and verification.

```
package main

import (
   "crypto/dsa"
   "crypto/md5"
   "crypto/rand"
   "encoding/base64"
   "fmt"
   "os"
   )

 func main() {
```

```
   // Message to Be signed and Verify
  secretMessage := "Hello 8gwifi.org"

   var priv dsa.PrivateKey
   params := &priv.Parameters

   err := dsa.GenerateParameters(params, rand.Reader, dsa.L1024N160)
   if err != nil {
      fmt.Println("Error Generating Parameter %s", err)
      os.Exit(1)
   }

   err = dsa.GenerateKey(&priv, rand.Reader)

   var pubkey dsa.PublicKey
   pubkey = priv.PublicKey

   fmt.Printf("Private Key :%x \n" , priv)
   fmt.Printf("Public Key %x \n:", pubkey)


   // Always Sign the Hash of the Message
   hashed := md5.Sum([]byte(secretMessage))
   r, s, err := dsa.Sign(rand.Reader, &priv, hashed[:])
   if err != nil {
      fmt.Println("Error signing: %s", err)
      return
    }

   //Signature Value in Base64 Encoded
   signature := base64.StdEncoding.EncodeToString(r.Bytes())
   fmt.Printf("Signature  %s\n",signature)

   // Signature Verification
   if !dsa.Verify(&priv.PublicKey, hashed[:], r, s) {
      fmt.Println("Error Verification ", err)
      return
  }

   fmt.Println("Signature Verification Passed  ")

}
```

The Output : This is sample output, it will vary when the program run again.

```
$ go run dsasignverify.go
Private Key :{{{fe5b0ec6f649687eeb34e6a72fbd000180a276098035d7752ac3144a921d89ab12ed\
407957153c7b5cbf98fb3b9f810372b7551acac4fba2157d2d77a48f74e0f5447aed957fa4aedca41b8a\
195e2788d0683b0cbf9ec57304ef7f09f58de3e45b28c339de9718a134857070b4530ebaee56e8bca20f\
c5c5006fe0eb2e566f85 d6862d150366601f2b7d1355cebea8be01159a95 4668192524ea7dbb5fa175\
4dfa9a3eecc886ceded22e1d894e3e99803c65585acfe1c61f6ffda2824b2499b88618867867fdbb420f\
513bc5bec571db1061a714e971da81db40e926b7b2dc81b4b36e5d4514fe2725eca5561da2368280925d\
78491252aa1f3873cf22f7a064b9fc15da93b15a01721c62de5fce021e81e93554} 40ae9f7b9c6936a0\
75e57978a2401084b576cdae03fdb66125eb967b7659faaded569f19d834add6ca76ec0d57251f32199b\
407bc2e625c5d01470aee1913bf678bc92421b49419d06df55190b05b375357bb703a115c7fab4c4edfe\
f28b030862950f8e1aac0feca55b368f888dafe0a0b140dbe7878c08ffb7fe46f6c13972} 59154b8dee\
635f86787f0993126fcb92dda7a9fe}
Public Key {{fe5b0ec6f649687eeb34e6a72fbd000180a276098035d7752ac3144a921d89ab12ed407\
957153c7b5cbf98fb3b9f810372b7551acac4fba2157d2d77a48f74e0f5447aed957fa4aedca41b8a195\
e2788d0683b0cbf9ec57304ef7f09f58de3e45b28c339de9718a134857070b4530ebaee56e8bca20fc5c\
5006fe0eb2e566f85 d6862d150366601f2b7d1355cebea8be01159a95 4668192524ea7dbb5fa1754df\
a9a3eecc886ceded22e1d894e3e99803c65585acfe1c61f6ffda2824b2499b88618867867fdbb420f513\
bc5bec571db1061a714e971da81db40e926b7b2dc81b4b36e5d4514fe2725eca5561da2368280925d784\
91252aa1f3873cf22f7a064b9fc15da93b15a01721c62de5fce021e81e93554} 40ae9f7b9c6936a075e\
57978a2401084b576cdae03fdb66125eb967b7659faaded569f19d834add6ca76ec0d57251f32199b407\
bc2e625c5d01470aee1913bf678bc92421b49419d06df55190b05b375357bb703a115c7fab4c4edfef28\
b030862950f8e1aac0feca55b368f888dafe0a0b140dbe7878c08ffb7fe46f6c13972}
:Signature  Odc2dUiEwXHIqMYxvj5A+agmca4=
Signature Verification Passed
```

Curve25519

Curve25519: **New Diffie-Hellman Speed Records**, is an elliptic curve[11] offering 128 bits of security and designed for use with the elliptic curve Diffie–Hellman[12] (ECDH) key agreement scheme.

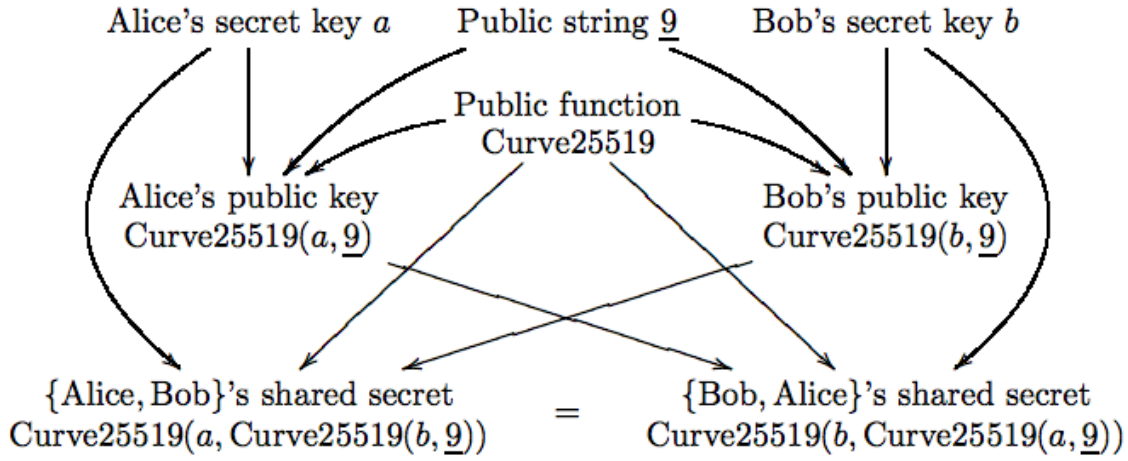
- Each Curve25519 user has a 32- byte secret key and a 32-byte public key
- Each set of two Curve25519 users has a 32-byte shared secret used to authenticate and encrypt messages between the two users

golang.org/x/crypto/curve25519[13] implementation of scalar multiplication on the elliptic curve known as curve25519. See https://cr.yp.to/ecdh.html[14]

`func ScalarBaseMult` sets dst to the product in*base where dst and base are the x coordinates of group points and all values are in little-endian form {line-numbers=off}* `go func ScalarBaseMult(dst, in *[32]byte) func ScalarMult` *sets dst to the product in*base where dst and base are the x coordinates of group points and all values are in little-endian form.

```
func ScalarMult(dst, in, base *[32]byte)
```

The following picture shows the data flow from secret keys through public keys to a shared secret. Reference Spec Curve25519[15]



[11] https://en.wikipedia.org/wiki/Elliptic_curve_cryptography
[12] https://en.wikipedia.org/wiki/Elliptic_curve_Diffie%E2%80%93Hellman
[13] https://godoc.org/golang.org/x/crypto/curve25519
[14] https://cr.yp.to/ecdh.html
[15] https://link.springer.com/content/pdf/10.1007/11745853_14.pdf

```
package main

import (
  "fmt"
  "golang.org/x/crypto/curve25519"
  "io"
  "crypto/rand"
  )
func main() {

   alicePublicKey, _, _ := GenerateKey(rand.Reader)
   _, bobPrivateKey, _ := GenerateKey(rand.Reader)

   var sharedKey [32]byte
   curve25519.ScalarMult(&sharedKey, alicePublicKey, bobPrivateKey)
   result := fmt.Sprintf("%x", sharedKey[:])
   fmt.Println("Arrived Secret Key ", result)

}

func GenerateKey(rand io.Reader) (publicKey, privateKey *[32]byte, err error) {
   publicKey = new([32]byte)
   privateKey = new([32]byte)
   _, err = io.ReadFull(rand, privateKey[:])
   if err != nil {
      publicKey = nil
      privateKey = nil
      return
  }

   curve25519.ScalarBaseMult(publicKey, privateKey)
   return
}
```

The Output

```
$ go run curve25519.go
Arrived Secret Key
8a942fb500ba8441071af9c7297e8aa183deadcd68668bc5320dba065654fd17
```