








[Discover Packages](#) > [Standard library](#) > [crypto](#) > [rsa](#) **rsa**

package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: [14](#) |Imported by: [31,329](#)**Details** Valid [go.mod](#) file   Redistributable license   Tagged version  Stable version [Learn more](#)**Repository**cs.opensource.google/go/go**Links** [Report a Vulnerability](#) Documentation <> **Documentation****Overview**

Package `rsa` implements RSA encryption as specified in PKCS #1 and [RFC 8017](#).

RSA is a single, fundamental operation that is used in this package to implement either public-key encryption or public-key signatures.

The original specification for encryption and signatures with RSA is PKCS #1 and the terms "RSA encryption" and "RSA signatures" by default refer to PKCS #1 version 1.5. However, that specification has flaws and new designs should use version 2, usually called by just OAEP and PSS, where possible.

Two sets of interfaces are included in this package. When a more abstract interface isn't necessary, there are functions for encrypting/decrypting with v1.5/OAEP and signing/verifying with v1.5/PSS. If one needs to abstract over the public key primitive, the `PrivateKey` type implements the `Decrypter` and `Signer` interfaces from the `crypto` package.

Operations in this package are implemented using constant-time algorithms, except for [GenerateKey](#), [PrivateKey.Precompute](#), and [PrivateKey.Validate](#). Every other operation only leaks the bit size of the involved values, which all depend on the selected key size.

Index[Constants](#)[Variables](#)

```
func DecryptOAEP(hash hash.Hash, random io.Reader, priv *PrivateKey, ciphertext []byte, ...) ([]byte, error)
```

```

func DecryptPKCS1v15(random io.Reader, priv *PrivateKey, ciphertext []byte) ([]byte, error)
func DecryptPKCS1v15SessionKey(random io.Reader, priv *PrivateKey, ciphertext []byte, key []byte)
error
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, msg []byte, label []byte) ([]byte,
error)
func EncryptPKCS1v15(random io.Reader, pub *PublicKey, msg []byte) ([]byte, error)
func SignPKCS1v15(random io.Reader, priv *PrivateKey, hash crypto.Hash, hashed []byte) ([]byte, error)
func SignPSS(rand io.Reader, priv *PrivateKey, hash crypto.Hash, digest []byte, ...) ([]byte, error)
func VerifyPKCS1v15(pub *PublicKey, hash crypto.Hash, hashed []byte, sig []byte) error
func VerifyPSS(pub *PublicKey, hash crypto.Hash, digest []byte, sig []byte, opts *PSSOptions) error
type CRTValue
type OAEPOptions
type PKCS1v15DecryptOptions
type PSSOptions
    func (opts *PSSOptions) HashFunc() crypto.Hash
type PrecomputedValues
type PrivateKey
    func GenerateKey(random io.Reader, bits int) (*PrivateKey, error)
    func GenerateMultiPrimeKey(random io.Reader, nprimes int, bits int) (*PrivateKey, error)
    func (priv *PrivateKey) Decrypt(rand io.Reader, ciphertext []byte, opts crypto.DecrypterOpts)
(plaintext []byte, err error)
    func (priv *PrivateKey) Equal(x crypto.PrivateKey) bool
    func (priv *PrivateKey) Precompute()
    func (priv *PrivateKey) Public() crypto.PublicKey
    func (priv *PrivateKey) Sign(rand io.Reader, digest []byte, opts crypto.SignerOpts) ([]byte, error)
    func (priv *PrivateKey) Validate() error
type PublicKey
    func (pub *PublicKey) Equal(x crypto.PublicKey) bool
    func (pub *PublicKey) Size() int

```

Examples

DecryptOAEP

DecryptPKCS1v15SessionKey

EncryptOAEP

SignPKCS1v15

VerifyPKCS1v15

Constants

[View Source](#)

```

const (
    // PSSSaltLengthAuto causes the salt in a PSS signature to be as large
    // as possible when signing, and to be auto-detected when verifying.
    PSSSaltLengthAuto = 0
    // PSSSaltLengthEqualsHash causes the salt length to equal the length
    // of the hash used in the signature.

```

```
PSSSaltLengthEqualsHash = -1
)
```

Variables

[View Source](#)

```
var ErrDecryption = errors.New("crypto/rsa: decryption error")
```

ErrDecryption represents a failure to decrypt a message. It is deliberately vague to avoid adaptive attacks.

[View Source](#)

```
var ErrMessageTooLong = errors.New("crypto/rsa: message too long for RSA key size")
```

ErrMessageTooLong is returned when attempting to encrypt or sign a message which is too large for the size of the key. When using SignPSS, this can also be returned if the size of the salt is too large.

[View Source](#)

```
var ErrVerification = errors.New("crypto/rsa: verification error")
```

ErrVerification represents a failure to verify a signature. It is deliberately vague to avoid adaptive attacks.

Functions

func DecryptOAEP

```
func DecryptOAEP(hash hash.Hash, random io.Reader, priv *PrivateKey, ciphertext []byte, label []byte) ([]byte, error)
```

DecryptOAEP decrypts ciphertext using RSA-OAEP.

OAEP is parameterised by a hash function that is used as a random oracle. Encryption and decryption of a given message must use the same hash function and sha256.New() is a reasonable choice.

The random parameter is legacy and ignored, and it can be as nil.

The label parameter must match the value given when encrypting. See EncryptOAEP for details.

► [Example](#)

func DecryptPKCS1v15

```
func DecryptPKCS1v15(random io.Reader, priv *PrivateKey, ciphertext []byte) ([]byte, error)
```

DecryptPKCS1v15 decrypts a plaintext using RSA and the padding scheme from PKCS #1 v1.5. The random parameter is legacy and ignored, and it can be as nil.

Note that whether this function returns an error or not discloses secret information. If an attacker can cause this function to run repeatedly and learn whether each instance returned an error then they can decrypt and forge signatures as if they had the private key. See `DecryptPKCS1v15SessionKey` for a way of solving this problem.

func `DecryptPKCS1v15SessionKey`

```
func DecryptPKCS1v15SessionKey(random io.Reader, priv *PrivateKey, ciphertext []byte,
key []byte) error
```

`DecryptPKCS1v15SessionKey` decrypts a session key using RSA and the padding scheme from PKCS #1 v1.5. The `random` parameter is legacy and ignored, and it can be `nil`. It returns an error if the ciphertext is the wrong length or if the ciphertext is greater than the public modulus. Otherwise, no error is returned. If the padding is valid, the resulting plaintext message is copied into `key`. Otherwise, `key` is unchanged. These alternatives occur in constant time. It is intended that the user of this function generate a random session key beforehand and continue the protocol with the resulting value. This will remove any possibility that an attacker can learn any information about the plaintext. See “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1”, Daniel Bleichenbacher, *Advances in Cryptology (Crypto '98)*.

Note that if the session key is too small then it may be possible for an attacker to brute-force it. If they can do that then they can learn whether a random value was used (because it'll be different for the same ciphertext) and thus whether the padding was correct. This defeats the point of this function. Using at least a 16-byte key will protect against this attack.

► Example

func `EncryptOAEP`

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, msg []byte, label
[]byte) ([]byte, error)
```

`EncryptOAEP` encrypts the given message with RSA-OAEP.

OAEP is parameterised by a hash function that is used as a random oracle. Encryption and decryption of a given message must use the same hash function and `sha256.New()` is a reasonable choice.

The `random` parameter is used as a source of entropy to ensure that encrypting the same message twice doesn't result in the same ciphertext.

The `label` parameter may contain arbitrary data that will not be encrypted, but which gives important context to the message. For example, if a given public key is used to encrypt two types of messages then distinct label values could be used to ensure that a ciphertext for one purpose cannot be used for another by an attacker. If not required it can be empty.

The message must be no longer than the length of the public modulus minus twice the hash length, minus a further 2.

► Example

func EncryptPKCS1v15

```
func EncryptPKCS1v15(random io.Reader, pub *PublicKey, msg []byte) ([]byte, error)
```

EncryptPKCS1v15 encrypts the given message with RSA and the padding scheme from PKCS #1 v1.5. The message must be no longer than the length of the public modulus minus 11 bytes.

The random parameter is used as a source of entropy to ensure that encrypting the same message twice doesn't result in the same ciphertext.

WARNING: use of this function to encrypt plaintexts other than session keys is dangerous. Use RSA OAEP in new protocols.

func SignPKCS1v15

```
func SignPKCS1v15(random io.Reader, priv *PrivateKey, hash crypto.Hash, hashed []byte) ([]byte, error)
```

SignPKCS1v15 calculates the signature of hashed using RSASSA-PKCS1-V1_5-SIGN from RSA PKCS #1 v1.5. Note that hashed must be the result of hashing the input message using the given hash function. If hash is zero, hashed is signed directly. This isn't advisable except for interoperability.

The random parameter is legacy and ignored, and it can be as nil.

This function is deterministic. Thus, if the set of possible messages is small, an attacker may be able to build a map from messages to signatures and identify the signed messages. As ever, signatures provide authenticity, not confidentiality.

► Example

func SignPSS

added in go1.2

```
func SignPSS(rand io.Reader, priv *PrivateKey, hash crypto.Hash, digest []byte, opts *PSSOptions) ([]byte, error)
```

SignPSS calculates the signature of digest using PSS.

digest must be the result of hashing the input message using the given hash function. The opts argument may be nil, in which case sensible defaults are used. If opts.Hash is set, it overrides hash.

func VerifyPKCS1v15

```
func VerifyPKCS1v15(pub *PublicKey, hash crypto.Hash, hashed []byte, sig []byte) error
```

VerifyPKCS1v15 verifies an RSA PKCS #1 v1.5 signature. hashed is the result of hashing the input message using the given hash function and sig is the signature. A valid signature is indicated by returning a nil error. If hash is zero then hashed is used directly. This isn't advisable except for interoperability.

func VerifyPSS

added in go1.2

```
func VerifyPSS(pub *PublicKey, hash crypto.Hash, digest []byte, sig []byte, opts *PSSOptions) error
```

VerifyPSS verifies a PSS signature.

A valid signature is indicated by returning a nil error. digest must be the result of hashing the input message using the given hash function. The opts argument may be nil, in which case sensible defaults are used. opts.Hash is ignored.

Types**type CRTValue**

```
type CRTValue struct {
    Exp    *big.Int // D mod (prime-1).
    Coeff  *big.Int // R·Coeff ≡ 1 mod Prime.
    R      *big.Int // product of primes prior to this (inc p and q).
}
```

CRTValue contains the precomputed Chinese remainder theorem values.

type OAEPOptions

added in go1.5

```
type OAEPOptions struct {
    // Hash is the hash function that will be used when generating the mask.
    Hash crypto.Hash

    // MGFHash is the hash function used for MGF1.
    // If zero, Hash is used instead.
    MGFHash crypto.Hash

    // Label is an arbitrary byte string that must be equal to the value
    // used when encrypting.
    Label []byte
}
```

OAEPOptions is an interface for passing options to OAEP decryption using the crypto.Decrypter interface.

type PKCS1v15DecryptOptions

added in go1.5

```
type PKCS1v15DecryptOptions struct {
    // SessionKeyLen is the length of the session key that is being
    // decrypted. If not zero, then a padding error during decryption will
    // cause a random plaintext of this length to be returned rather than
```

```
// an error. These alternatives happen in constant time.
SessionKeyLen int
}
```

PKCS1v15DecryptOptions is for passing options to PKCS #1 v1.5 decryption using the crypto.Decrypter interface.

type PSSOptions

added in go1.2

```
type PSSOptions struct {
    // SaltLength controls the length of the salt used in the PSS signature. It
    // can either be a positive number of bytes, or one of the special
    // PSSSaltLength constants.
    SaltLength int

    // Hash is the hash function used to generate the message digest. If not
    // zero, it overrides the hash function passed to SignPSS. It's required
    // when using PrivateKey.Sign.
    Hash crypto.Hash
}
```

PSSOptions contains options for creating and verifying PSS signatures.

func (*PSSOptions) HashFunc

added in go1.4

```
func (opts *PSSOptions) HashFunc() crypto.Hash
```

HashFunc returns opts.Hash so that PSSOptions implements crypto.SignerOpts.

type PrecomputedValues

```
type PrecomputedValues struct {
    Dp, Dq *big.Int // D mod (P-1) (or mod Q-1)
    Qinv   *big.Int // Q^-1 mod P

    // CRTValues is used for the 3rd and subsequent primes. Due to a
    // historical accident, the CRT for the first two primes is handled
    // differently in PKCS #1 and interoperability is sufficiently
    // important that we mirror this.
    //
    // Note: these values are still filled in by Precompute for
    // backwards compatibility but are not used. Multi-prime RSA is very rare,
    // and is implemented by this package without CRT optimizations to limit
    // complexity.
    CRTValues []CRTValue
    // contains filtered or unexported fields
}
```

type PrivateKey

```

type PrivateKey struct {
    PublicKey          // public part.
    D                  *big.Int // private exponent
    Primes              []*big.Int // prime factors of N, has >= 2 elements.

    // Precomputed contains precomputed values that speed up RSA operations,
    // if available. It must be generated by calling PrivateKey.Precompute and
    // must not be modified.
    Precomputed PrecomputedValues
}

```

A PrivateKey represents an RSA key

func GenerateKey

```

func GenerateKey(random io.Reader, bits int) (*PrivateKey, error)

```

GenerateKey generates an RSA keypair of the given bit size using the random source random (for example, crypto/rand.Reader).

func GenerateMultiPrimeKey

```

func GenerateMultiPrimeKey(random io.Reader, nprimes int, bits int) (*PrivateKey, error)

```

GenerateMultiPrimeKey generates a multi-prime RSA keypair of the given bit size and the given random source.

Table 1 in "[On the Security of Multi-prime RSA](#)" suggests maximum numbers of primes for a given bit size.

Although the public keys are compatible (actually, indistinguishable) from the 2-prime case, the private keys are not. Thus it may not be possible to export multi-prime private keys in certain formats or to subsequently import them into other code.

This package does not implement CRT optimizations for multi-prime RSA, so the keys with more than two primes will have worse performance.

Note: The use of this function with a number of primes different from two is not recommended for the above security, compatibility, and performance reasons. Use GenerateKey instead.

func (*PrivateKey) Decrypt

added in go1.5

```

func (priv *PrivateKey) Decrypt(rand io.Reader, ciphertext []byte, opts crypto.DecryptOptions) (plaintext []byte, err error)

```

Decrypt decrypts ciphertext with priv. If opts is nil or of type *PKCS1v15DecryptOptions then PKCS #1 v1.5 decryption is performed. Otherwise opts must have type *OAEPOptions and OAEP decryption is done.

func (*PrivateKey) Equal

added in go1.15

```
func (priv *PrivateKey) Equal(x crypto.PrivateKey) bool
```

Equal reports whether priv and x have equivalent values. It ignores Precomputed values.

func (*PrivateKey) Precompute

```
func (priv *PrivateKey) Precompute()
```

Precompute performs some calculations that speed up private key operations in the future.

func (*PrivateKey) Public

added in go1.4

```
func (priv *PrivateKey) Public() crypto.PublicKey
```

Public returns the public key corresponding to priv.

func (*PrivateKey) Sign

added in go1.4

```
func (priv *PrivateKey) Sign(rand io.Reader, digest []byte, opts crypto.SignerOpts)
([]byte, error)
```

Sign signs digest with priv, reading randomness from rand. If opts is a *PSSOptions then the PSS algorithm will be used, otherwise PKCS #1 v1.5 will be used. digest must be the result of hashing the input message using opts.HashFunc().

This method implements crypto.Signer, which is an interface to support keys where the private part is kept in, for example, a hardware module. Common uses should use the Sign* functions in this package directly.

func (*PrivateKey) Validate

```
func (priv *PrivateKey) Validate() error
```

Validate performs basic sanity checks on the key. It returns nil if the key is valid, or else an error describing a problem.

type PublicKey

```
type PublicKey struct {
    N *big.Int // modulus
    E int      // public exponent
}
```

A PublicKey represents the public part of an RSA key.

func (*PublicKey) Equal

added in go1.15

```
func (pub *PublicKey) Equal(x crypto.PublicKey) bool
```

Equal reports whether pub and x have the same value.

func (*PublicKey) Size

added in go1.11

```
func (pub *PublicKey) Size() int
```

Size returns the modulus size in bytes. Raw signatures and ciphertexts for or by this public key will have the same size.

Source Files

[View all](#) 

[notboring.go](#)
[pkcs1v15.go](#)

[pss.go](#)
[rsa.go](#)

Why Go

[Use Cases](#)
[Case Studies](#)

Get Started

[Playground](#)
[Tour](#)
[Stack Overflow](#)
[Help](#)

Packages

[Standard Library](#)
[About Go Packages](#)

About

[Download](#)
[Blog](#)
[Issue Tracker](#)
[Release Notes](#)
[Brand Guidelines](#)
[Code of Conduct](#)

Connect

[Twitter](#)
[GitHub](#)
[Slack](#)
[r/golang](#)
[Meetup](#)
[Golang Weekly](#)

Copyright

[Terms of Service](#)

[Privacy Policy](#)

[Report an Issue](#)



Google