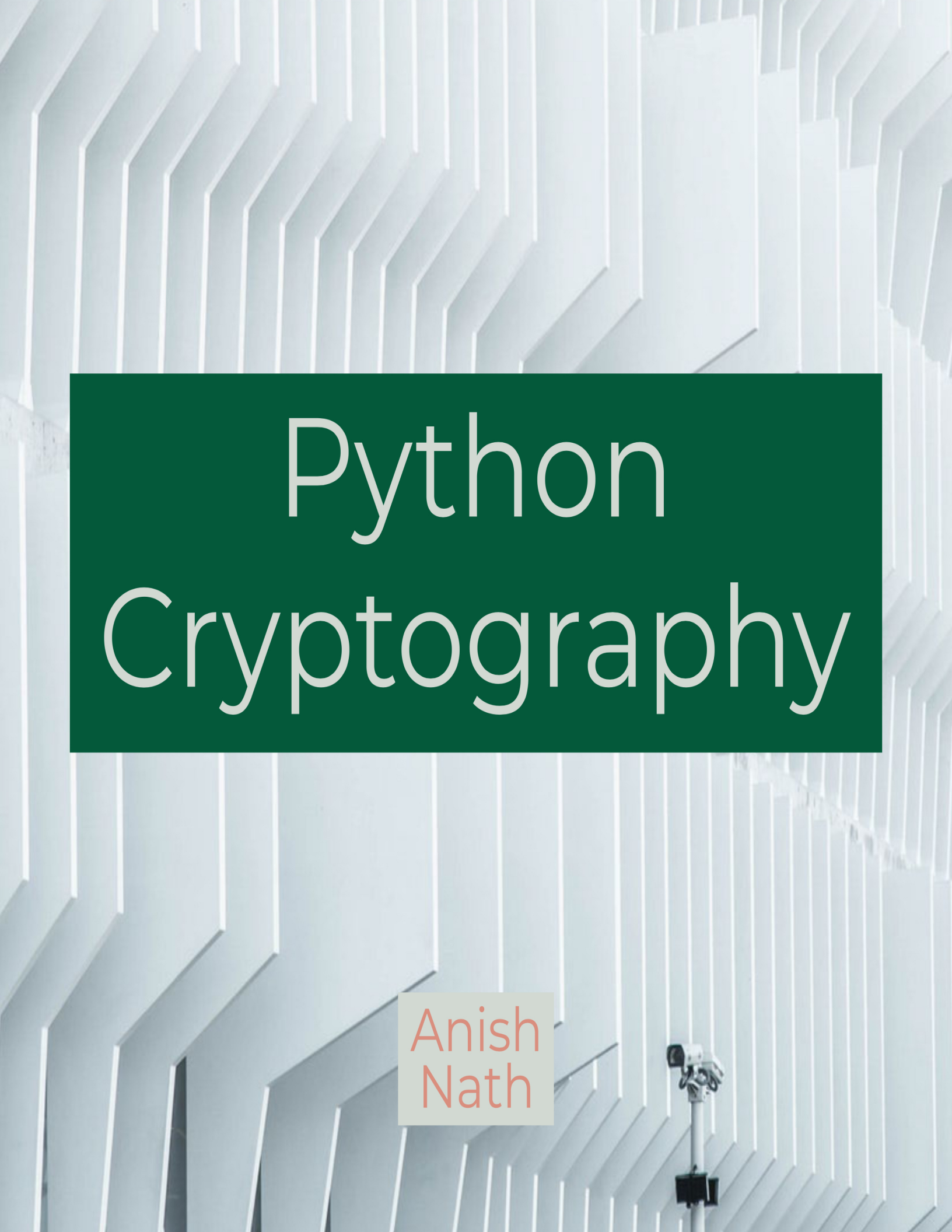

Python
Cryptography
Anish
Nath

Python Cryptography

Python Cryptography

Anish Nath

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

This version was published on 2018-10-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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.

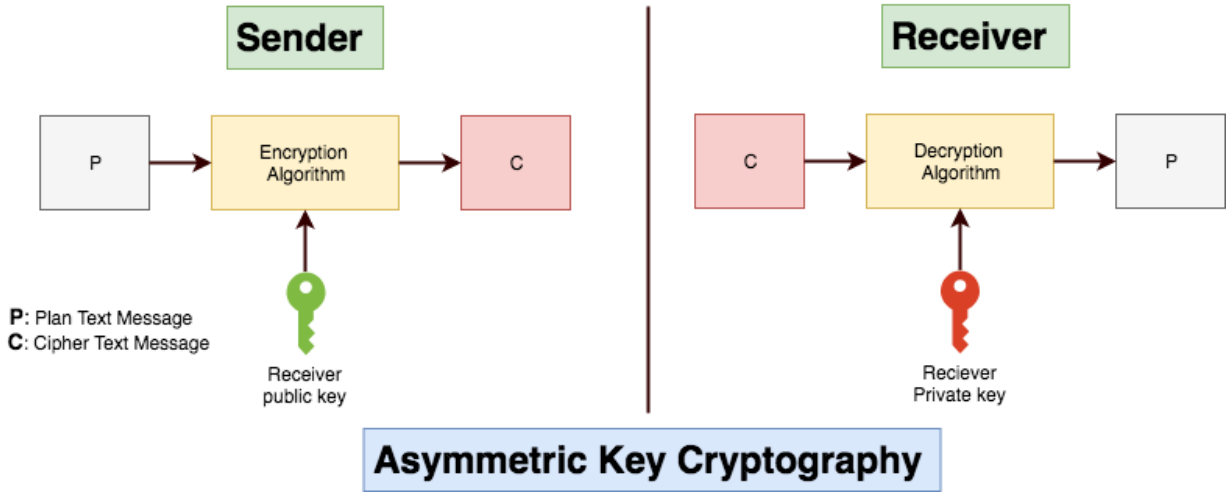
I would also thanks to reader who spend their money and time to read this book, without them there is no Inspiration.

Contents

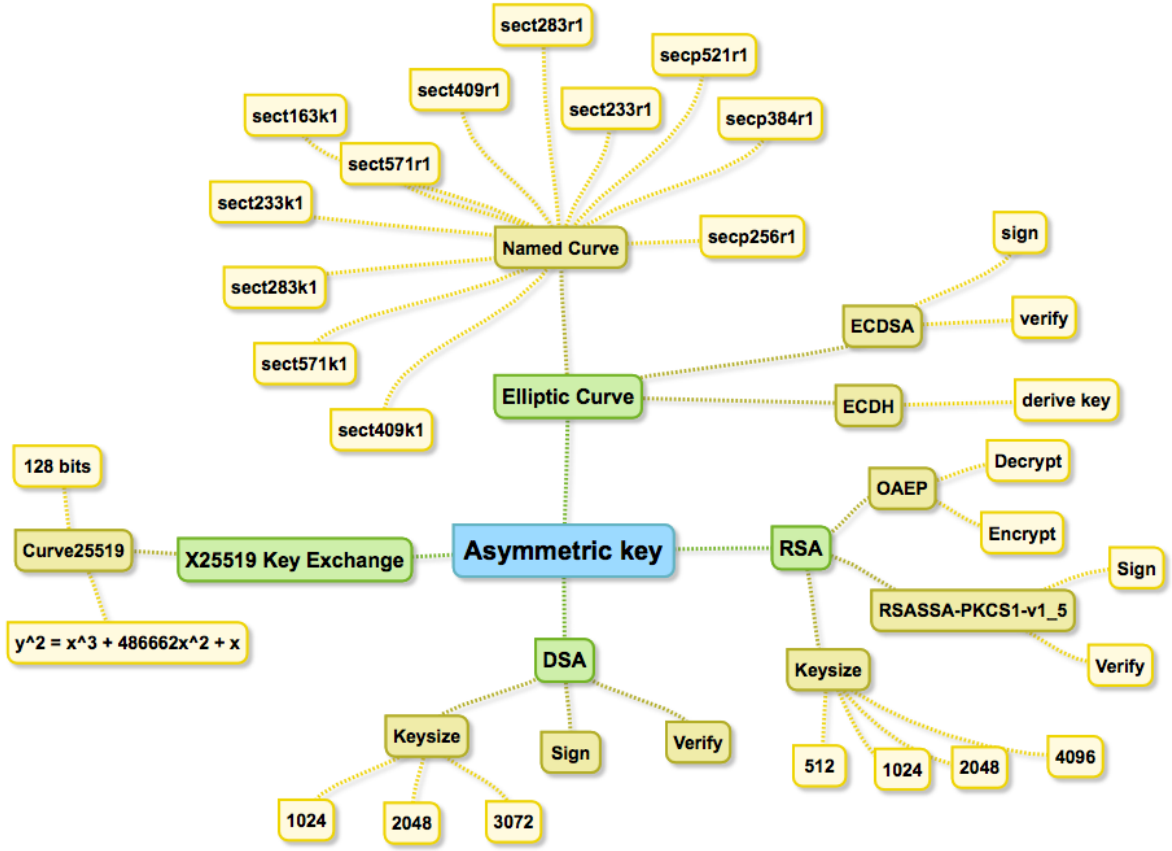
Asymmetric-key algorithms

Public-key cryptography, or asymmetric cryptography, is any cryptographic system that uses pairs of keys: public keys which is distributed widely, and private keys which are known only to the owner.

This accomplishes two functions: authentication, where the public key verifies that a holder of the paired private key sent the message, and encryption, where only the paired private key holder can decrypt the message encrypted with the public key.



Asymmetric Key Algorithm MindMap



X25519 key exchange

X25519 is an elliptic curve Diffie-Hellman key exchange[1] using Curve25519[2]. It allows two parties to jointly agree on a shared secret using an insecure channel.

In this example the `shared_key` is passed to KDF function, to further derive a secure key exchange.

[1] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange
[2] https://en.wikipedia.org/wiki/Curve25519

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric.x25519 import X25519PrivateKey
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from binascii import hexlify, unhexlify
import os


# Generate a private key for use in the exchange.
private_key = X25519PrivateKey.generate()

peer_public_key = X25519PrivateKey.generate().public_key()
shared_key = private_key.exchange(peer_public_key)

print "Shared Key " + hexlify(shared_key)

# Perform key derivation.
derived_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=os.urandom(13),
    info=b'Some Data ',
    backend=default_backend()
).derive(shared_key)


print "Derive Key " +  hexlify(derived_key)
```

RSA

RSA stands for Ron Rivest, Adi Shamir, and Leonard Adleman, who first publicly described the algorithm in 1978. A user of RSA creates and publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The private KEY (prime factors) MUST BE KEPT SECRET. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key enough it is virtually impossible to decode the message.

Generate RSA Keys

Generate RSA private/public Key and save in PEM format

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

encryptedpass = "myverystrongpassword"

# Generate an RSA Keys
private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )

public_key = private_key.public_key()

# Save the RSA key in PEM format
with open("/tmp/rsakey.pem", "wb") as f:
    f.write(private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.BestAvailableEncryption(encryptedpass),
    )
    )

# Save the Public key in PEM format
with open("/tmp/rsapub.pem", "wb") as f:
    f.write(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo,
    )
)
```

RSA-OAEP encryption/ decryption example

Optimal Asymmetric Encryption Padding is a padding scheme often used together with RSA encryption, standardized in PKCS#1 v2

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.serialization import load_pem_private_key
from cryptography.hazmat.primitives.serialization import load_pem_public_key

encryptedpass = "myverystrongpassword"
plaintextMessage = "Hello 8gwifi.org"

alicePubKey = load_pem_public_key(open('/tmp/alicepub.pem', 'rb').read(),default_bac\
kend())

ciphertext = alicePubKey.encrypt(
    plaintextMessage,
    padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
  )
)


alicePrivKey = load_pem_private_key(open('/tmp/alice.pem', 'rb').read(),encryptedpas\
s,default_backend())

d = alicePrivKey.decrypt(
    ciphertext,
    padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
  )
)

assert plaintextMessage, d
```

RSA Sign Verify Example

Valid paddings for signatures are `PSS`[3] and `PKCS1v15`[4]. `PSS` is the recommended choice for any new protocols or applications, `PKCS1v15` should only be used to support legacy protocols.

[3] https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#cryptography.hazmat.primitives.asymmetric.padding.PSS
[4] https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#cryptography.hazmat.primitives.asymmetric.padding.PKCS1v15

Probabilistic Signature Scheme (PSS) is a cryptographic[5] signature scheme[6] designed by Mihir Bellare[7] and Phillip Rogaway[8]

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.serialization import load_pem_private_key
from cryptography.hazmat.primitives.serialization import load_pem_public_key

encryptedpass = "myverystrongpassword"
plaintextMessage = "Hello 8gwifi.org"

alicePrivKey = load_pem_private_key(open('/tmp/alice.pem', 'rb').read(),encryptedpas\
s,default_backend())

sig = alicePrivKey.sign(
    plaintextMessage,
    padding.PSS(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH,
    ),
    hashes.SHA256()
)

alicePubKey = load_pem_public_key(open('/tmp/alicepub.pem', 'rb').read(),default_bac\
kend())

ciphertext = alicePubKey.verify(
    sig,
    plaintextMessage,
    padding.PSS(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH,
    ),
    hashes.SHA256()
)
```

[5] https://en.wikipedia.org/wiki/Cryptography
[6] https://en.wikipedia.org/wiki/Digital_signature
[7] https://en.wikipedia.org/wiki/Mihir_Bellare
[8] https://en.wikipedia.org/wiki/Phillip_Rogaway

DSA

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**.

- DSA Private Key : Used for Signature generation
- DSA Public Key: Used for Signature Verification

DSA Key Size : 1024, 2048 or 3072

Generate DSA Public/Private key and Store in Encrypted PEM format

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes

encryptedpass = "myverystrongpassword"

# Generate an RSA Keys
private_key = dsa.generate_private_key(
        key_size=2048,
        backend=default_backend()
    )

public_key = private_key.public_key()

# Save the RSA key in PEM format
with open("/tmp/dsakey.pem", "wb") as f:
    f.write(private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.BestAvailableEncryption(encryptedpass),
    )
    )

# Save the Public key in PEM format
with open("/tmp/dsapub.pem", "wb") as f:
    f.write(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo,
```

```
    )
)
```

DSA Sign/verify Message

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.serialization import load_pem_private_key
from cryptography.hazmat.primitives.serialization import load_pem_public_key

encryptedpass = "myverystrongpassword"
plaintextMessage = "Hello 8gwifi.org"


privateKey = load_pem_private_key(open('/tmp/dsakey.pem', 'rb').read(),encryptedpass\
,default_backend())

sig = privateKey.sign(
    plaintextMessage,
    hashes.SHA256()
)


publicKey = load_pem_public_key(open('/tmp/dsapub.pem', 'rb').read(),default_backend\
())

ciphertext = publicKey.verify(
    sig,
    plaintextMessage,
    hashes.SHA256()
)
```

Generate DSA public/private key using DH param-part1

This example will generate DSA public/private key using the DH parameters and then store the key into PEM format

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import dsa

encryptedpass = "myverystrongpassword"

parameters = dsa.generate_parameters(2048, default_backend())
skey = parameters.generate_private_key()
numbers = skey.private_numbers()
skey_parameters = numbers.public_numbers.parameter_numbers
pkey = skey.public_key()
parameters = pkey.parameters()
parameter_numbers = parameters.parameter_numbers()

print parameter_numbers.p
print parameter_numbers.q
print parameter_numbers.g

# Save the RSA key in PEM format
with open("/tmp/dsakey.pem", "wb") as f:
    f.write(skey.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.BestAvailableEncryption(encryptedpass),
    )
    )

# Save the Public key in PEM format
with open("/tmp/dsapub.pem", "wb") as f:
    f.write(pkey.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo,
    )
)
```

Generate DSA public/private key from the defined DH Parameter

This example will generate DSA public/private key using the defined DH parameters

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dsa

p=175628339748224396140181355652569699881813470248510623740591206685224250268384255\
2
401419485673544365374694912582204800573435683225744140869248656128095022289257162507\
481366119761008955265309887839899181516660497951731739548267151470640724403365873861\
794410445516360488355961960135907665134353497502483699328 91
q=735357924361709222913818385151798699202077448881
g=107176209836536718962801284808131605075363838277339744565850617301534676770726222 7\
429126104367980592496186789708393242880893038278225437661505051592469514979453793392\
558506224886088633646579656346293418470374116067654366991809721156366574446879171569\
569606790376439216739270626035503243344737499461101393771 51
```

```
parameters = dsa.DSAParameterNumbers(
            p,
            q,
            g
        ).parameters(default_backend())


skey = parameters.generate_private_key()
numbers = skey.private_numbers()
skey_parameters = numbers.public_numbers.parameter_numbers
pkey = skey.public_key()
parameters = pkey.parameters()
parameter_numbers = parameters.parameter_numbers()
```

Diffie-Hellman key exchange

Diffie-Hellman is an algorithm used to establish a shared secret between two parties. It is primarily used as a method of exchanging cryptography keys for use in symmetric encryption algorithms like AES.

For security and performance reasons use of ECDH[9] instead of DH is recommended.

[9] https://cryptography.io/en/latest/hazmat/primitives/asymmetric/ec/#cryptography.hazmat.primitives.asymmetric.ec.ECDH

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from binascii import hexlify, unhexlify
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import load_pem_public_key
import os

# DH generator must be 2 or 5
# Minumun DH key size is 512
parameters = dh.generate_parameters(generator=2, key_size=512,backend=default_backen\
d())

#You must generate a new private key using generate_private_key() for each exchange(\
) when performing an DHE key exchange
private_key = parameters.generate_private_key()

#peer_public_key = parameters.generate_private_key().public_key()

#Public Key Extracted
peer_public_key = load_pem_public_key(open('/tmp/shpub.pem', 'rb').read(),default_ba\
ckend())

shared_key = private_key.exchange(peer_public_key)

print "Shared Key " + hexlify(shared_key)

# Perform key derivation.
derived_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=os.urandom(13),
    info=b'Some Data ',
    backend=default_backend()
).derive(shared_key)


print "Derive Key " +  hexlify(derived_key)
```

- DHE (or EDH), the ephemeral form of this exchange, is **strongly preferred** over simple DH

and provides forward secrecy[10] .

- For the next handshake RE-GENERATE another private key, but we can reuse the same parameters.

DH Key exchange with predefined parameter

This example will generate DH private key using the defined DH parameters, this can also be extended to generate DH public key using p,g.

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from binascii import hexlify, unhexlify
from cryptography.hazmat.primitives.serialization import load_pem_public_key
import os

# DH generator must be 2 or 5
# Minumun DH key size is 512

p=175628339748224396140181355652569698818134702485106237405912066852242502683842552\
401419485673544365374694912582204800573435683225744140869248656128095022289257162507\
481366119761008955265309887839899181516660497951731739548267151470640724403365873861\
794410445516360488355961960135907665134353497502483699328 91
g=107176209836536718962801284808131605075363838277339744565850617301534676770726222 7\
429126104367980592496186789708393242880893038278225437661505051592469514979453793392\
558506224886088633646579656346293418470374116067654366991809721156366574446879171569\
569606790376439216739270626035503243344737499461101393771 51

pn = dh.DHParameterNumbers(p,g)
parameters = pn.parameters(default_backend())
#You must generate a new private key using generate_private_key() for each exchange(\
) when performing an DHE key exchange
private_key = parameters.generate_private_key()

#peer_public_key = parameters.generate_private_key().public_key()

#Public Key Extracted
peer_public_key = load_pem_public_key(open('/tmp/shpub.pem', 'rb').read(),default_ba\
ckend())

shared_key = private_key.exchange(peer_public_key)
```

[10] https://en.wikipedia.org/wiki/Forward_secrecy

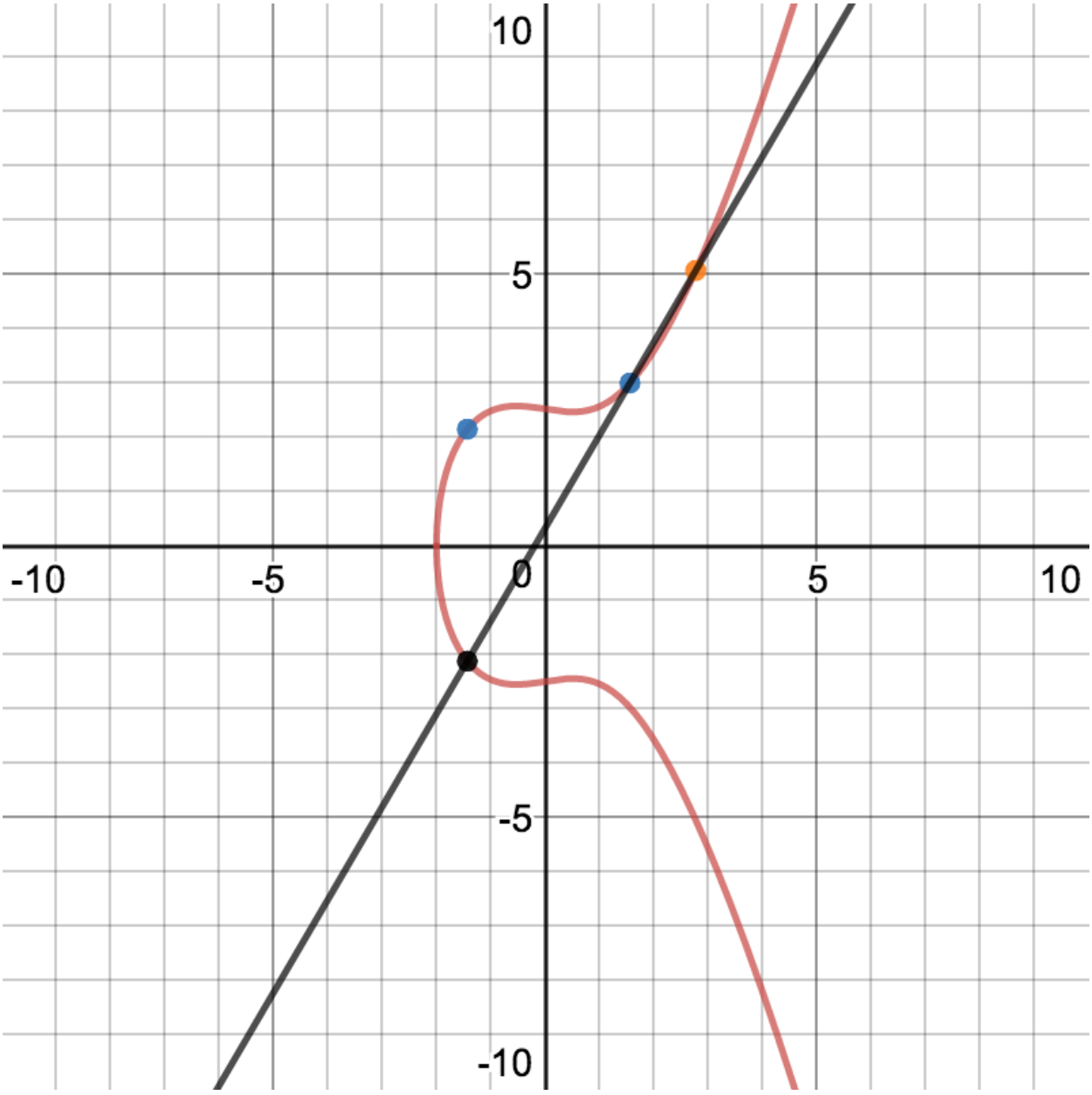
```
print "Shared Key " + hexlify(shared_key)

# Perform key derivation.
derived_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=os.urandom(13),
    info=b'Some Data ',
    backend=default_backend()
).derive(shared_key)


print "Derive Key " +  hexlify(derived_key)
```

Elliptic-curve cryptography

Elliptic-curve cryptography (ECC) is an approach to public-key cryptography[11] based on the algebraic structure[12] of elliptic curves[13] over finite fields[14]

This is a elliptic Curve of **Secp256k1** . The curve equation is $y^2=x^3+3$ and itâ€™s very popular in **Bitcoinâ€™s** public-key cryptography also.

[11] https://en.wikipedia.org/wiki/Public-key_cryptography
[12] https://en.wikipedia.org/wiki/Algebraic_structure
[13] https://en.wikipedia.org/wiki/Elliptic_curve
[14] https://en.wikipedia.org/wiki/Finite_field



This example will use `SECP256K1` curve and generate encrypted EC private/public key in PEM format

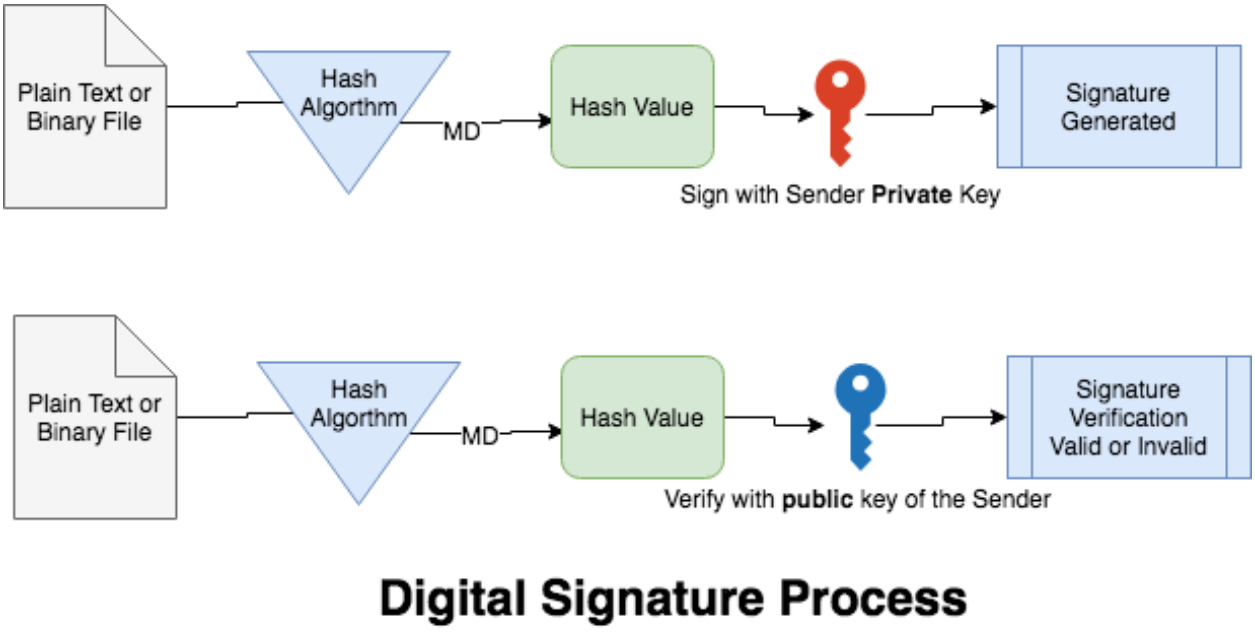
```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization


encryptedpass = "myverystrongpassword"

# Generate an RSA Keys
private_key = ec.generate_private_key(
        ec.SECP256K1,
        backend=default_backend()
    )

public_key = private_key.public_key()

# Save the RSA key in PEM format
with open("/tmp/eckey.pem", "wb") as f:
    f.write(private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.BestAvailableEncryption(encryptedpass),
    )
    )

# Save the Public key in PEM format
with open("/tmp/ecpub.pem", "wb") as f:
    f.write(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo,
    )
)
```

NIST Mapping of Cryptography API Supported Curve

CurveName	NIST Name
sect571k1	K-571
sect409k1	K-409
sect283k1	K-283
sect233k1	K-233
sect163k1	K-163
sect571r1	B-571
sect409r1	B-409
sect283r1	B-283
sect233r1	B-233
sect163r2	B-163
secp521r1	P-521
secp384r1	P-384
secp256r1	P-256
secp224r1	P-224
secp192r1	P-192
secp256k1	P-256

ECDSA Sign/verify Message

The Elliptic Curve Digital Signature Algorithm



This example will load the encrypted EC keys which is stored in PEM format, then generate the message signature value, followed by the verify process.

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.serialization import load_pem_private_key
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives.asymmetric import ec

encryptedpass = "myverystrongpassword"
plaintextMessage = "Hello 8gwifi.org"


# EC DSA Generate Signature
privateKey = load_pem_private_key(open('/tmp/eckey.pem', 'rb').read(),encryptedpass,\
default_backend())

sig = privateKey.sign(
    plaintextMessage,
    ec.ECDSA(hashes.SHA256())
)

publicKey = load_pem_public_key(open('/tmp/ecpub.pem', 'rb').read(),default_backend(\
))

# EC DSA Verify Signature
ciphertext = publicKey.verify(
    sig,
    plaintextMessage,
    ec.ECDSA(hashes.SHA256())
)
```