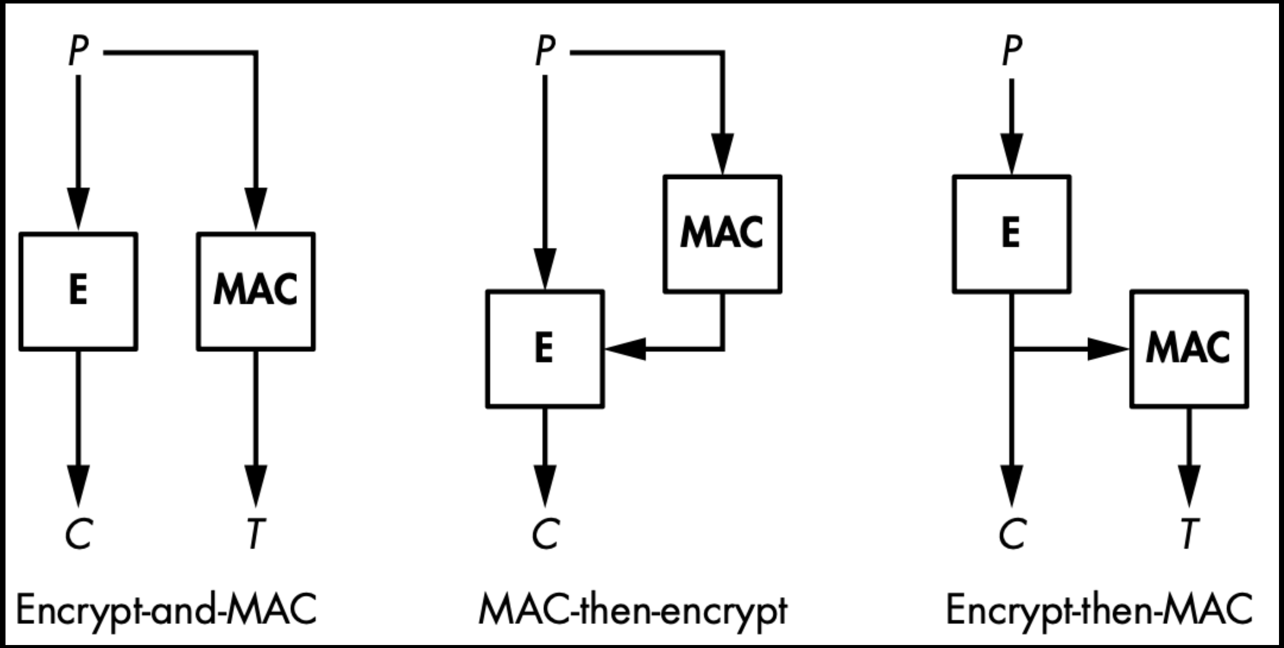
Cryptography

Authenticated Encryption

Content

Authenticated Encryption using MACs

- MACs protects messages' authenticity and integrity by generating tags
- Authenticated encryption protects confidentiality + integrity + authenticity
- AE can be constructed in three ways



Authenticated Encryption using MACs
Encrypt-and-MAC

- EaM computes the ciphertext and the tag separately
  - They can be computed in **parallel**
- Encryption: $C = E(K_1, P)$
- Authentication: $T = MAC(K_2, P)$
- The recipient receives C and T:
  1. Decrypts C to obtain P: $P = D(K_1, C)$
  2. Checks the tag of the message: $T = MAC(K_2, P)$
  3. Compares the computed tag with the received tag
  4. Verification fails if C or T is corrupted

Authenticated Encryption using MACs
Encrypt-and-MAC

- EaM is the least secure composition – a MAC may leak info on the plaintext

- Even a secure MAC may leak the plaintext:
  - The goal of the MACs is to make **unforgeable tags** $\boldsymbol{T}$ of P
  - Tags are not necessarily pseudorandom
  - Hence, tags may reveal information about the plaintext

- If a MAC is a PRF, tags will be pseudorandom → secure tags → no leaks on P

Authenticated Encryption using MACs
Encrypt-and-MAC

- SSH protocol uses the EaM construction

- The used MAC is secure: HMAC-SHA-256 → no leaks on the plaintext

- $T = MAC(K, N||P)$
  - N is a sequence number that is incremented for each sent packet
  - N is concatenated with the plaintext P

Authenticated Encryption using MACs
MAC-then-Encrypt

- MtE first computes the tag then do encryption of the plaintext with the tag:
  1. $T = MAC(K_2, P)$
  2. $C = E(K_1, P||T)$
- The sender transmits C only – it contains the ciphertext and the tag
- The recipient do the following to get the plaintext:
  1. $P||T = D(K_1, C)$
  2. $T` = MAC(K_2, P)$
  3. $T == T`?$
- MtE is more secure that EaM:
  - Hides the authentication tag → Prevents the tag from leaking info about the plaintext

Authenticated Encryption using MACs Encrypt-then-MAC

- EtM sends two values to the recipient:
  1. $C = E(K_1, P)$
  2. $T = MAC(K_2, C)$
- The recipient receives (C, T):
  1. $T` = MAC(K_2, C)$
  2. $T` == T?$
  3. If true, compute $P = D(K_1, C)$
  4. Otherwise, discard the ciphertext
- There is no need to perform decryption if the message is corrupted
- Used in IPSec protocol with VPN tunnels

Content

Authenticated Ciphers

- Authenticated ciphers are alternatives to the cipher and MAC combination
- They are normal ciphers, but they return a ciphertext and a tag

- Encryption: $AE(K, P) = (C, T)$
- Decryption: $AD(K, P, T) = P$
  - If either or both C and T are invalid, AD will return an error

- Security requirement: cannot forge (C, T) that the AD accepts and decrypts

Authenticated Ciphers
Authenticated Encryption with Associated Data

- Encryption: $AEAD(K, P, A) = (C, A, T)$
- Decryption: $ADAD(K, C, A, T) = (P, A)$
- Associated data are authenticated but not encrypted
- Useful when encrypting network packets
  - Encrypt and authenticate the content of the packets
  - Authenticate the headers of the packets
  - Headers are kept in clear to send the packet to its recipient
- If any of the cleartext, the ciphertext, or the tag is corrupted, the packet will be discarded.

Authenticated Ciphers
Authenticated Encryption with Associated Data

- You can leave A or P empty
  - Empty A → normal authenticate ciphers
  - Empty P → a MAC on cleartext data
- AEAD is the current norm for authenticated encryption
- Google has its SQL, called **GoogleSQL**, in **BigQuery**
- BigQuery is an AI-driven autonomous data warehouse solution
  - https://cloud.google.com/bigquery?hl=en
- Tink supports AEAD algorithms → open-source crypto library by Google
  - https://developers.google.com/tink
- TLS uses AEAD

Authenticated Ciphers
Authenticated Encryption with Associated Data

- What makes a good authenticated cipher?

Security
- Confidentiality = strongest cipher
- Authenticity = best MAC
- Misuse resistance: still secure when the nonce is reused

Performance
- No. of bits processed/sec
- Parallelizable?
- Internal structure: one layer or two layers
- Streamable: discard any already-processed blocks.

Functionality
- The format of the input
- Use the same algorithm for encryption and decryption

Content

Authenticated Encryption
Authenticated Encryption using MACs
Authenticated Ciphers
AES-GCM: The Authenticated Cipher Standard
OCB: Authenticated Cipher Faster than GCM
SIV: The Safest Authentication Cipher?
Permutation-Based AEAD

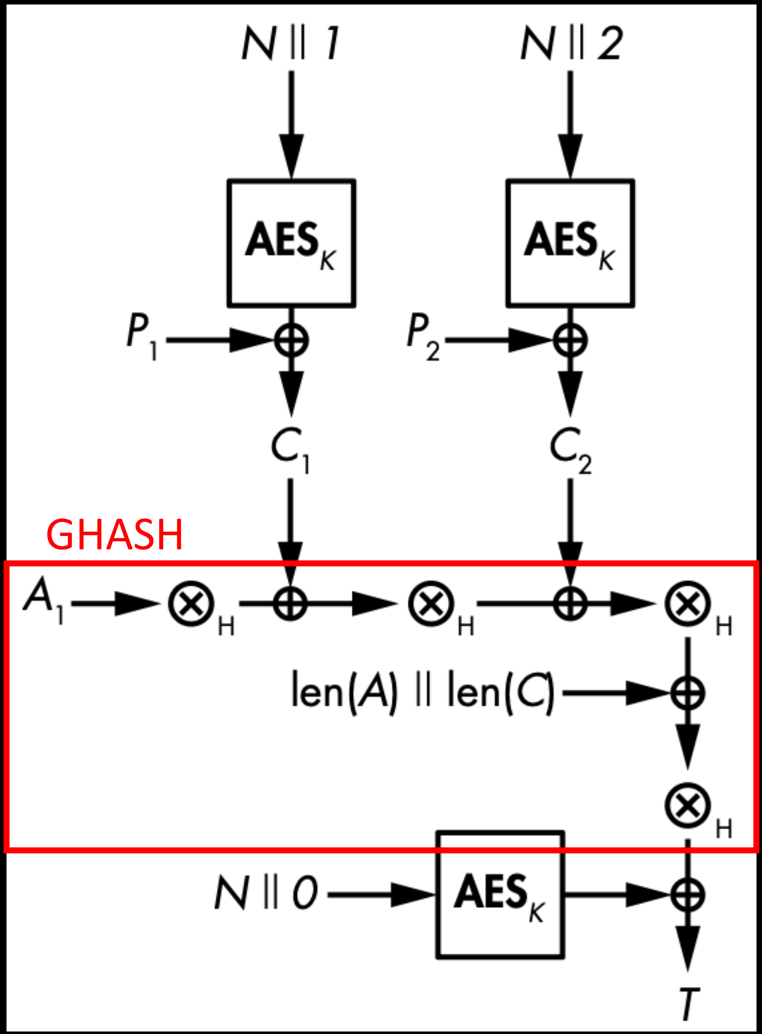
GNU Privacy Guard

AES-GCM: The Authenticated Cipher Standard

- AES-GCM is the most widely used authenticated cipher
  - NIST standard
- It’s based on the AES cipher and the Galois counter mode
- GCM is a modified CTR mode that supports computing a tag
- Used in Apple’s Cryptokit: http://developer.apple.com/documentation/cryptokit/aes/gcm

AES-GCM: The Authenticated Cipher Standard

- GCM Internals: CTR and GHASH
  - AES with secret key K
  - AES encrypts a block of 96-bit nonce concatenated with a counter
  - XOR the result with the plaintext
  - Mixes the ciphertext with the associated data using XORs and multiplications
  - Wagman-Carter MAC for authentication
    - Uses a hash function called GHASH
    - $T = GHASH(H, A, C) \oplus AES(K, N||0)$
    - H is a hash key → $H = AES(K, 0)$



AES-GCM: The Authenticated Cipher Standard

- AES-GCM is **fragile** against nonce reuse
  - If the nonce is reused twice, an attacker can get the authentication key H
- Compromised H → forge tags
  - $T_1 = GHASH(H, A_1, C_1) \oplus AES(K, N||0)$, $T_2 = GHASH(H, A_2, C_2) \oplus AES(K, N||0)$
  - $T_1 \oplus T_2 = GHASH(H, A_1, C_1) \oplus GHASH(H, A_2, C_2)$
  - A and C are known, GHASH is linear function → recover H
- In 2016, 184 HTTPs servers had repeated nonces: https://eprint.iacr.org/2016/475/

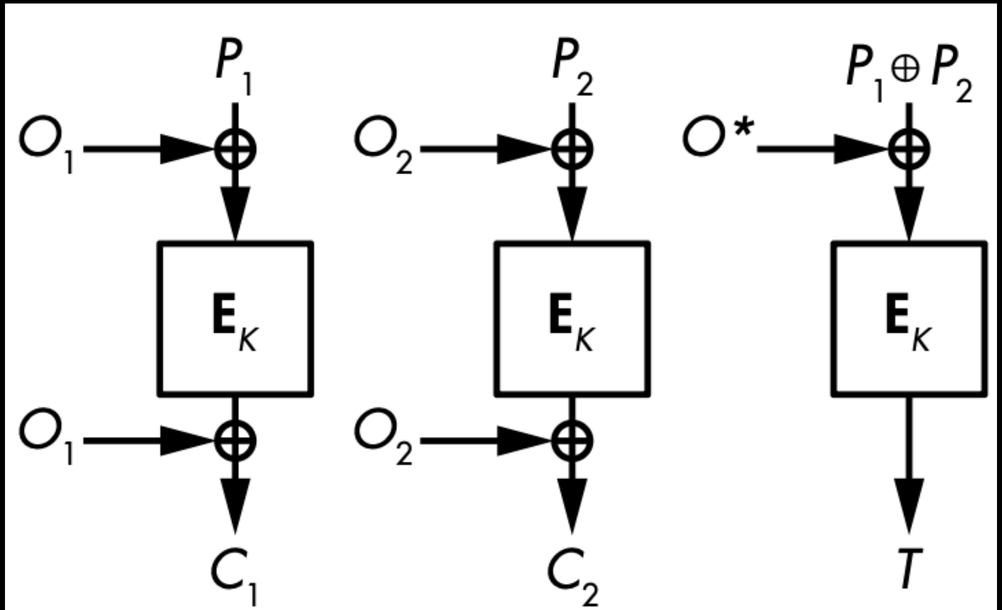
Content

OCB: Authenticated Cipher Faster than GCM

- OCB = offset codebook
- Simpler than GCM
- Paralleizable and streamable as GCM
- OCB is a bit less fragile than GCM against repeated nonces
    - Attackers may see identical ciphertext blocks – cannot recover secret keys
- Not a formal standard – requires licenses from the inventor
    - Free license
- One layer and 1 key to produce the ciphertext and the tag

OCB: Authenticated Cipher Faster than GCM

- $C = E(K, P \oplus O) \oplus O$
- O is the offset, a value that depends on the key and the nonce incremented for each new block processed
- $T = E(K, S \oplus O^*)$
  - $S = P_1 \oplus P_2 \oplus P_3 \oplus \cdots$
  - O^* is an offset value computed from the offset of the last plaintext block processed



- OCB supports associated data:
  - $T = E(K, S \oplus O^*) \oplus E(K, A_1 \oplus O_1) \oplus E(K, A_2 \oplus O_2) \oplus$ …

Content

Authenticated Encryption
Authenticated Encryption using MACs
Authenticated Ciphers
AES-GCM: The Authenticated Cipher Standard
OCB: Authenticated Cipher Faster than GCM
SIV: The Safest Authentication Cipher?
Permutation-Based AEAD

GNU Privacy Guard

SIV: The Safest Authenticated Cipher

- Synthetic IV is an authenticated cipher mode used with AES
- SIV is secure even if you use the same nonce twice
  - Attackers will only be able to learn whether the same plaintext was encrypted twice
- SIV combines a cipher E and a PRF:
  - $T = PRF(K_1, N||P)$
  - $C = E(K_2, T, P)$ - the tag acts as the nonce of E
- Not streamable: it must keep the entire plaintext in memory:
  - To encrypt a 100GB plaintext, you must first store the 100GB in memory
- Common constructions: CMAC-AES as a PRF + AES-CTR as a cipher, GCM-SIV

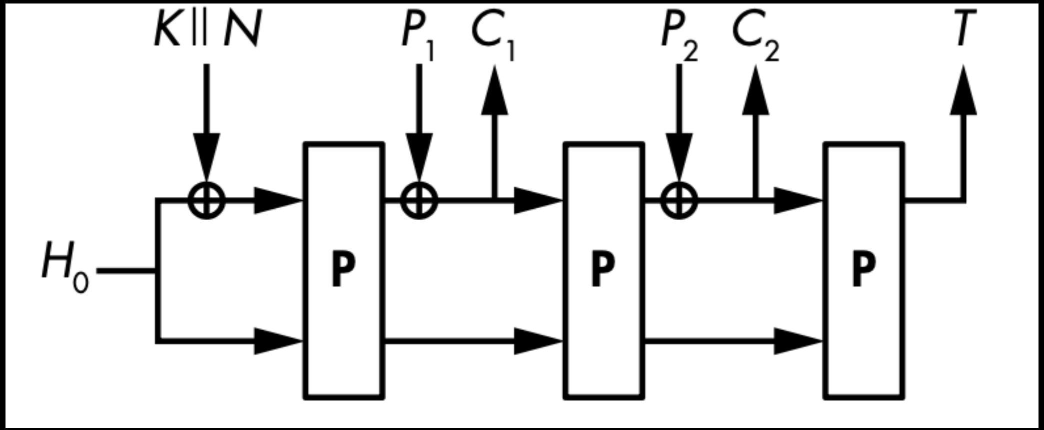
Content

Authenticated Encryption
Authenticated Encryption using MACs
Authenticated Ciphers
AES-GCM: The Authenticated Cipher Standard
OCB: Authenticated Cipher Faster than GCM
SIV: The Safest Authentication Cipher?
Permutation-Based AEAD

GNU Privacy Guard

Permutation-Based AEAD

- A different way to build AEAD – uses permutations instead of block ciphers



- H_0 is a fixed initial state
- All security relies on the secrecy of the internal state
- Hard to design a secure permutation-based AEAD

Permutation-Based AEAD

- Blocks must be padded properly with extra bits to ensure any two different messages will yield different results

- You cannot just add zeros as padding

- Given two messages: $M_1 00$ and $M_1 000$, after padding, $M_1 0000$ and $M_1 0000$ → the last block is the same → same tag

Permutation-Based AEAD

- Reused nonce → the authentication tag won't be compromised; attacker learns that two encrypted messages begin with the same value
- Example:
  - Given two plaintexts: $ABCXYZ$ and $ABCDYZ$
  - $E(ABCXYZ) = JKLTUV$
  - $E(ABCDYZ) = JKLMNO$
  - Attacker cannot learn that the two plaintexts shared the same final two blocks (YZ)
- Pros: single layer of operations, streamable, single algorithm for encryption and decryption
- Cons: not parallelizable

TASK

- Use $pycryptodome$ to implement Encrypt-and-Mac encryption and decryption functions, Encrypt-then-Mac encryption and decryption functions, and Mac-then-Encrypt encryption and decryption function
- Implement AES-GCM encryption and decryption functions that can process associated data
- Implement AES-OCB encryption and decryption functions that can process associated data

Content

GnuPG

- GPG is a cryptography tool for managing public and private keys, and doing encrypt, decrypt, sign, and verify operations.
- It is an open-source version of PGP.
- Most programming languages binds PGP with it.
- Installation on Linux

```
sudo apt-get install gpg
```

```
gpg --help
```

GnuPG

- To generate new key pairs

```
gpg --gen-key
```

  - It will create a new userID

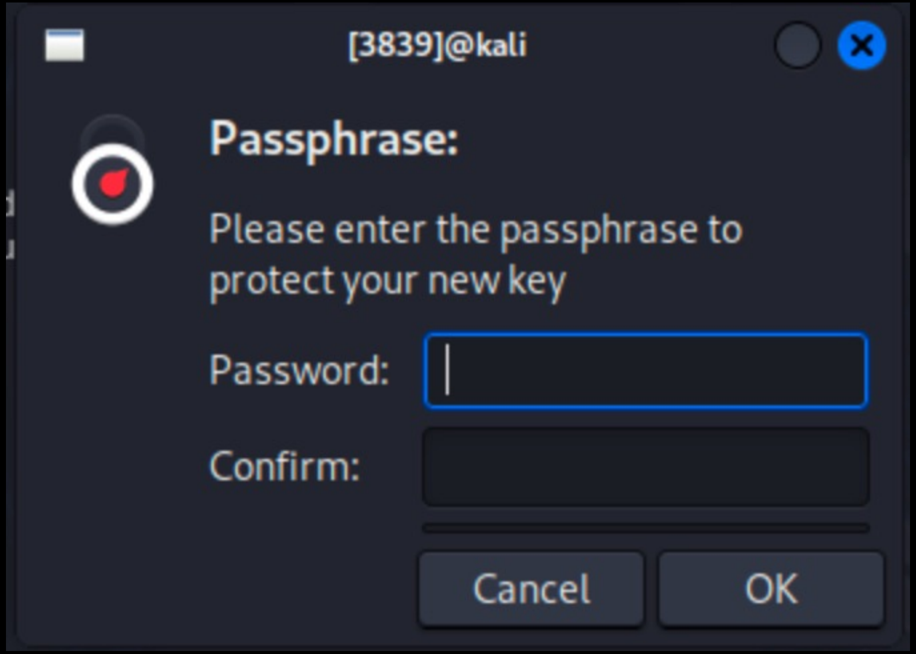
```
└─$ gpg --gen-key
gpg (GnuPG) 2.2.43; Copyright (C) 2023 g10 Code GmbH
This is free software: you are free to change and redistr
There is NO WARRANTY, to the extent permitted by law.

Note: Use "gpg --full-generate-key" for a full featured k

GnuPG needs to construct a user ID to identify your key.

Real name: ThiisOmar
Email address: myemail@email.com
You selected this USER-ID:
    "ThiisOmar <myemail@email.com>"

Change (N)ame, (E)mail, or (O)kay/(Q)uit? o
We need to generate a lot of random bytes. It is a good i
some other action (type on the keyboard, move the mouse,
disks) during the prime generation; this gives the random
generator a better chance to gain enough entropy.
```



GnuPG

- Notice that it will take some time to generate secure pseudorandom bytes
  - It will ask you to move the mouse or do some random things to generate enough entropy
- The public and private keys will be created and signed

```
Change (N)ame, (E)mail, or (O)kay/(Q)uit? o
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /home/kali/.gnupg/trustdb.gpg: trustdb created
gpg: directory '/home/kali/.gnupg/openpgp-revocs.d' created
gpg: revocation certificate stored as '/home/kali/.gnupg/openpgp-revocs.d/699A43AD47F41A01EB1A630783687899CC59B3C9.rev'
public and secret key created and signed.

pub   rsa3072 2025-04-27 [SC] [expires: 2028-04-26]
      699A43AD47F41A01EB1A630783687899CC59B3C9
uid                      ThiisOmar <myemail@email.com>
sub   rsa3072 2025-04-27 [E] [expires: 2028-04-26]
```

GnuPG

- GPG automatically selects the RSA cipher with 3072-bit key.
    - Some versions allows you select which cipher with what key size
    - Some versions have different default settings

- Also, GPG creates a revocation certificate.

- A revocation certificate is useful when you forget your passphrase or if your private key is lost or stolen.
    - It’s published to tell people that your current public key is no longer in use

GnuPG

- An advanced mode to generate key pair

```
gpg --full-generate-key
```

  - Choose the type of the cipher and the signature

```
gpg (GnuPG) 2.2.43; Copyright (C) 2023 g10 Code GmbH
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
  (14) Existing key from card
Your selection?
```

GnuPG

```
gpg (GnuPG) 2.2.43; Copyright (C) 2023 g10 Code GmbH
This is free software: you are free to change and redistribut
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
  (14) Existing key from card
Your selection? 2
DSA keys may be between 1024 and 3072 bits long.
What keysize do you want? (2048) 2048
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 120
Key expires at Mon Aug 25 11:00:08 2025 EDT
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: ThisisMe1
Email address: email@me.com
Comment: Testing
You selected this USER-ID:
    "ThisisMe1 (Testing) <email@me.com>"
```

```
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: WARNING: some OpenPGP programs can't handle a DSA key with this digest size
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: revocation certificate stored as '/home/kali/.gnupg/openpgp-revocs.d/58512A1B06FD
public and secret key created and signed.

pub   dsa2048 2025-04-27 [SC] [expires: 2025-08-25]
      58512A1B06FDF06AE2CAB5852C263D0D7D768A10
uid                      ThisisMe1 (Testing) <email@me.com>
sub   elg2048 2025-04-27 [E] [expires: 2025-08-25]
```

[SC]: the key for **Sign** and **Certify**

[E]: Subkey for encryption

Fingerprint: The long hexadecimal string is the fingerprint of your key.

GnuPG

- To list the keys on your public keyring
    - The keyring is a file that contains multiple public keys

```
gpg --list-keys
```

```
└─$ gpg --list-keys
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   2  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 2u
gpg: next trustdb check due at 2025-08-25
/home/kali/.gnupg/pubring.kbx
-----------------------------
pub   rsa3072 2025-04-27 [SC] [expires: 2028-04-26]
      699A43AD47F41A01EB1A630783687899CC59B3C9
uid           [ultimate] ThiisOmar <myemail@email.com>
sub   rsa3072 2025-04-27 [E] [expires: 2028-04-26]

pub   dsa2048 2025-04-27 [SC] [expires: 2025-08-25]
      58512A1B06FDF06AE2CAB5852C263D0D7D768A10
uid           [ultimate] ThisisMe1 (Testing) <email@me.com>
sub   elg2048 2025-04-27 [E] [expires: 2025-08-25]
```

GnuPG

- To send your key to another party, you must export it first

```
gpg --output <filename.gpg> --export <userID>
```

```
gpg --output myPubKey.gpg --export myemail@email.com
```

```
file myPubKey.gpg
```

```
└─$ file myPubKey.gpg
myPubKey.gpg: OpenPGP Public Key Version 4, Created Sun Apr 27 14:45:41 2025, RSA (Encrypt or Sign, 3072 bits); User ID; Signature; OpenPGP Certificate
```

- The output file is in binary format, try reading it

```
cat myPubKey.gpg
```

GnuPG

- To export it in ASCII format to be published, use *armor* option

```
gpg --armor --output <filename.gpg> --export <userID>
```

```
gpg --armor --output myPubKey.txt --export myemail@email.com
```

```
cat myPubKey.txt
```

GnuPG

- Upon receiving the public key file, import it your public key ring

```
gpg --import myPubKey.gpg # gpg --import <filename>
```

- Before start using the key, you must verify it:

```
gpg --list-keys
```

```
gpg --edit-key myemail@email.com
```

```
gpg> fpr  #compute the fingerprint
```

- A key's fingerprint is verified with the key's owner.
    - Via a call or other ways to guarantee that you are communicating with the true owner

GnuPG

- After checking the fingerprint with the key’s owner, sign it to validate it

```
gpg> sign
```

GnuPG

- Example:

```
alice% gpg --edit-key blake@cyb.org

pub  1024D/9E98BC16  created: 1999-06-04 expires: never  trust: -/q
sub  1024g/5C8CBD41  created: 1999-06-04 expires: never
(1)  Blake (Executioner) <blake@cyb.org>

Command> fpr
pub  1024D/9E98BC16 1999-06-04 Blake (Executioner) <blake@cyb.org>
     Fingerprint: 268F 448F CCD7 AF34 183E  52D8 9BDE 1A08 9E98 BC16
```

GnuPG

- Example:

```
Command> sign

pub  1024D/9E98BC16  created: 1999-06-04 expires: never   trust: -/q
     Fingerprint: 268F 448F CCD7 AF34 183E  52D8 9BDE 1A08 9E98 BC16

     Blake (Executioner) <blake@cyb.org>

Are you really sure that you want to sign this key
with your key: "Alice (Judge) <alice@cyb.org>"

Really sign?
```

GnuPG

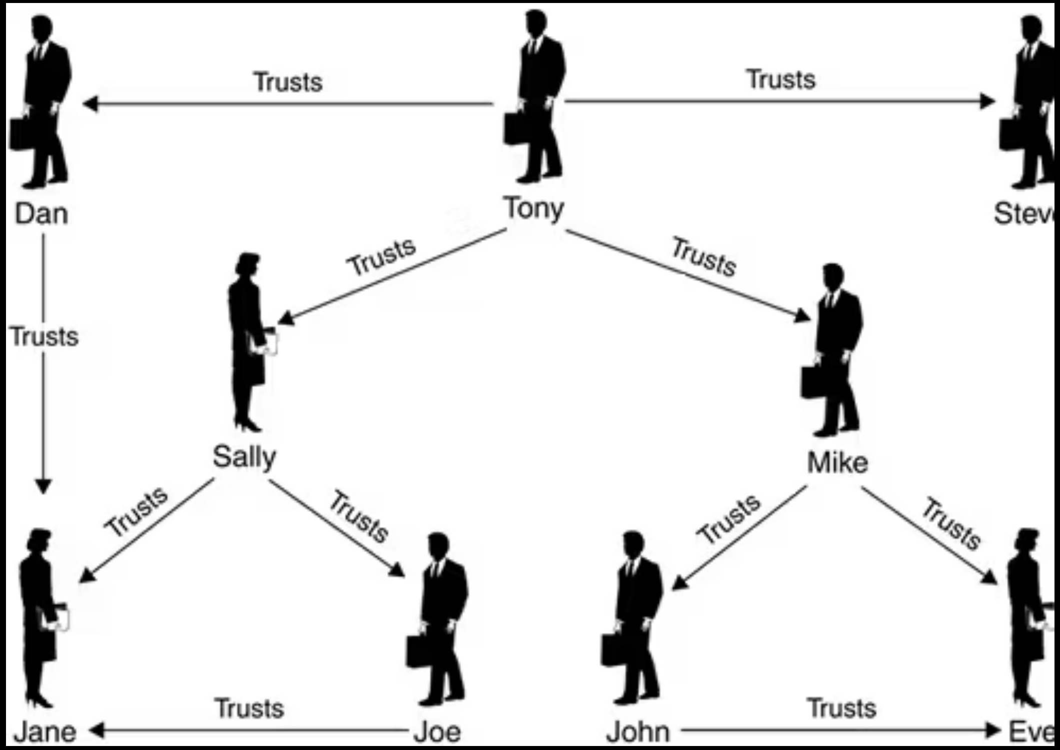
- Example:

```
Command> check
uid  Blake (Executioner) <blake@cyb.org>
sig!       9E98BC16 1999-06-04   [self-signature]
sig!       BB7576AC 1999-06-04   Alice (Judge) <alice@cyb.org>
```

- Note: signing someone's else public key is like: *"I, with my private key, vouch that this key really belongs to this person/email"*
- So, you're creating a digital signature on their key that says, "I trust that this key belongs to Blake blake@cyb.org"
- This process is part of the **Web of Trust** model
  - https://en.wikipedia.org/wiki/Web_of_trust

GnuPG

- WoT is used to establish authenticity of the public key to its owner
- It’s a **decentralized** alternative to the **Public key infrastructure**
    - PKI relies on certificate authorities



GnuPG

- A single key may have several signatures and its own self-signature
- For example, to see the signatures on john@example.com's key

```
gpg --check-sigs john@example.com
pub       rsa3072 2020-01-28 [SC]
          1234 5678 90AB CDEF 1234 5678 90AB CDEF 1234 5678
uid       [unknown] John Doe <john@example.com>
sig!3     90ABCDEF12345678 2020-01-28 John Doe <john@example.com>
sig       1122334455667788 2020-01-29 Alice Smith <alice@another.org>
sig       9988776655443322 2020-03-01 Bob Brown <bob@example.net>
```

  - sig!3 by John Doe = self-signature (done when the key is created)
  - sig by Alice and Bob = Other people have signed this key because they've verified it.
  - [unknown] = this user hasn't been trusted yet. If it is trusted, it will show [ultimate]

GnuPG

- Encrypting documents

```
gpg --output <output filename> --encrypt --recipient <recipientID> <filename>
```

- Example:

```
echo 'This is a document to be encrypted' > mydoc
```

```
gpg --output enc_mydoc.gpg --encrypt --recipient myemail@email.com mydoc
```

- Decrypting documents (you will be requested to enter the passphrase)

```
gpg --output <output filename> --decrypt <encrypted filename>
```

```
gpg --output dec_mydoc.gpg --decrypt enc_mydoc.gpg
```

GnuPG

- Symmetric key encryption

```
gpg --output <output filename> --symmetric <input filename>
```

- Example: you will be request to enter a passphrase

```
gpg --output sym_enc_doc.gpg --symmetric mydoc
```

- The key used to drive the symmetric cipher is derived from the passphrase.
  - For good security, it should not be the same passphrase that you use to protect your private key.
- To decrypt

```
gpg --output sym_dec_mydoc --decrypt sym_enc_doc.gpg
```

GnuPG

- Sign a document
  - you will be requested to enter your passphrase to unlock your private key

```
# gpg --output <out filename> --sign <in filename>
gpg --output doc.sig --sign mydoc
```

  - The document is compressed before signed, and the output is in binary format.
- To verify the signature

```
# gpg --verify <filename>
gpg --verify doc.sig
```

```
└─$ gpg --verify doc.sig
gpg: Signature made Mon Apr 28 03:17:59 2025 EDT
gpg:                using RSA key 699A43AD47F41A01EB1A630783687899CC59B3C9
gpg: Good signature from "ThiisOmar <myemail@email.com>" [ultimate]
```

- To verify and recover the message

```
gpg --output recv_sig --decrypt doc.sig; cat recv_sig
```

GnuPG

- In many cases, it is recommended to generate a clear signature
  - Signing the document without compressing it

```
# gpg --clearsign <in filename>
gpg --clearsign mydoc
```

  - If you didn't specify an output file, the output is written to $< filename >.asc$

```
cat mydoc.asc
```

GnuPG

- Using previous methods, the signature is attached to the original document
- To generate a detached signature (a separate signature from the file)

```
gpg --output doc.sig --detach-sig mydoc
```

- To verify it, you need both the document file and the signature

```
gpg --verify doc.sig mydoc
```

GnuPG

- To cleanup (delete) your keys, you must remove secret keys and then the public keys.

```
# gpg --delete-secret-keys <user ID>
gpg --delete-secret-keys myemail@email.com
```

```
# gpg --delete-keys <user ID>
gpg --delete-keys myemail@email.com
```