# 深入淺出 Google/Dropbox/GitHub 都採用的 FIDO U2F

js [at] jong.sh

# 大綱

- 密碼
- 密碼管理員
- OTP 兩步驟驗證
- FIDO 聯盟與 U2F 兩步驟驗證標準
- U2F 虛擬硬體實作
- U2F 伺服器端函式庫實作

# 我是誰

- 快速密碼學實驗室
- 密碼學
  - 後量子密碼學
  - 各種被廣泛使用的通訊協定如 TLS
- 魔術方塊
- js@jong.sh

# 今天來聊聊線上使用者身份認證

- 假設已經有單向認證的安全加密通道
  - 良好的 HTTPS 伺服器、良好的 HTTPS 客戶端、良好的 PKI
  - 使用者已經知道遠端伺服器的身份
  - 接下來的通訊明文只有自己和遠端伺服器看得見

- 此時, 使用者如何向遠端證明自己是誰？

# 芝麻開門之，密碼身份認證

- 註冊:將密碼送出去
- 認證:將密碼送出去

- 問題？

# 密碼的問題

- 不能抵禦 replay attack (廢話)
  - 你不斷地傳送一模一樣的訊息, 使該訊息暴露於許多通訊節點, 但是希望這個訊息不會落入惡意攻擊者手中

# 密碼的問題

- 唯一的秘密資訊，密碼本身，以明文的形式出現在太多地方，太多洩漏的可能性
  - 本地端：鍵盤（可見光、指紋、聲音）、Bluetooth、USB Hub、作業系統（用戶空間各種程式、內核空間）
  - 遠端伺服器：你無法掌控
- 你的密碼會以某種形式長時間保存在 server
- 你每次登入，密碼明文都會直接出現在 server
- 即使你的本地端是安全的，你也無法得知 server 是否確實「妥善處理」你的密碼

# 密碼的問題

- 很容易以釣魚攻擊方式釣走
- 一不小心就把常用的密碼敲出去
- 特別是一些神速的打字員，常用密碼已經打得非常熟練

# 密碼的問題

- 使用者體驗違反資訊安全實踐，它先天「鼓勵」人們重複使用類似、甚至相同密碼於多個地方，並且久不更換。
  - 在手機上輸入具有 64-bit entropy 的長密碼？
  - 在 10 個最常用的線上服務都使用完全不同、沒有關聯的密碼？
- 一個服務被攻陷的連鎖反應：使用者在其他地方的帳號莫名地被盜取。
- 對於一般大眾，線上帳號被盜的事件層出不窮。

# 密碼的問題

- 「我是線上服務 X 的忠實使用者，我本來就完全信賴他們能妥善處理我的資料。就算有災情，對我的影響也僅僅限於 X 上罷了！」

- 密碼資料庫洩露，被得知密碼明文了，攻擊者（特別是針對性的）還可能會得知更多關於你的資訊
- 密碼是否完全沒有重複使用過？沒有特定模式？
- 密碼是否 entropy 夠高？（針對離線甚至線上的暴搜）

# 密碼管理員

- 你只要記住一組 master password 就好，由管理員幫你生成（甚至自動填入）各個服務不同的密碼

- 問題？

# 密碼管理員

- 自動生成密碼？自動填入密碼？自動更換新密碼？
- 沒有統一的密碼規則政策，也沒有一個統一的通知介面
- 真的可以安全地自動填入密碼嗎？
- 你的 clipboard 安全嗎？
- 如果在 N 個網站更換新密碼，無法穩定、安全地自動化，你願意做嗎？
- 你的 master password 真的安全嗎？
  - 你在多少地方輸入過你的 master password，那些設備跑著各式各樣的應用程式、直接連接上網路？它這樣真的「安全」嗎？

# 密碼管理員

- 各種設備、各種作業系統、各種原生應用程式、數個瀏覽器、各種線上服務，大部分沒有「密碼管理員」專用的、統一的標準介面。
- 實作往往是不斷跟著平台跑的 hacks
- 使用加密形式儲存你的密碼資料庫？你將這個檔案保存在哪些地方？密碼資料庫是否直接存上雲端？
  - 你的 master password「永遠」不會洩漏嗎？
  - 每一次察覺洩漏可能，所有線上服務都要砍掉重練重置密碼
  - 攻擊者策略：看到密碼資料庫被加密？總之先記錄起來再說。它就像一個上鎖的寶箱，雖然今天打不開，但是也許三年後就開了。也許那個使用者不懂得要常常更新密碼。
- 當然在安全性上，利用密碼管理員 >= 不依賴密碼管理員

# 看來, 只依賴密碼不是個好主意

- 雙重因素身份驗證
  - 比起只用密碼, 多提供一層安全性, 除非兩個驗證機制同時被攻陷...
- 最常見的 2nd factor 形式是 OTP
  - TOTP 軟體(假設有同步 clock, 且雙方都妥善保存對稱式密鑰)
  - 手機簡訊(假設傳遞簡訊的通道是安全的... ?)
  - 硬體 token
- OTP 特色
  - 輕易地緩解密碼不安全的問題
  - 比較沒有 replay attack 的困擾
  - 有標準規範, 使用者體驗較統一

# OTP 的問題

- 使用者體驗不佳
  - 手動切換 app、自行讀取 6-digit PIN、手動輸入之, 操作步驟明顯增加
  - 訊號不佳處, 等待簡訊
  - 隨身多攜帶數個硬體 token, 為了多個服務
- 客戶端安全性
  - 幫你生成 OTP 的 app 如何保存你的金鑰？
- 伺服器端安全性
  - 他們的 OTP 檢查是如何實現的？金鑰保存在哪裡？
- 簡訊的安全性
  - NIST (SP800-63B draft) 宣告簡訊 OTP 的末日
- 仍然對釣魚沒輒
  - 避免釣魚的責任還是落在使用者身上

# 對於大量用戶的線上服務來說

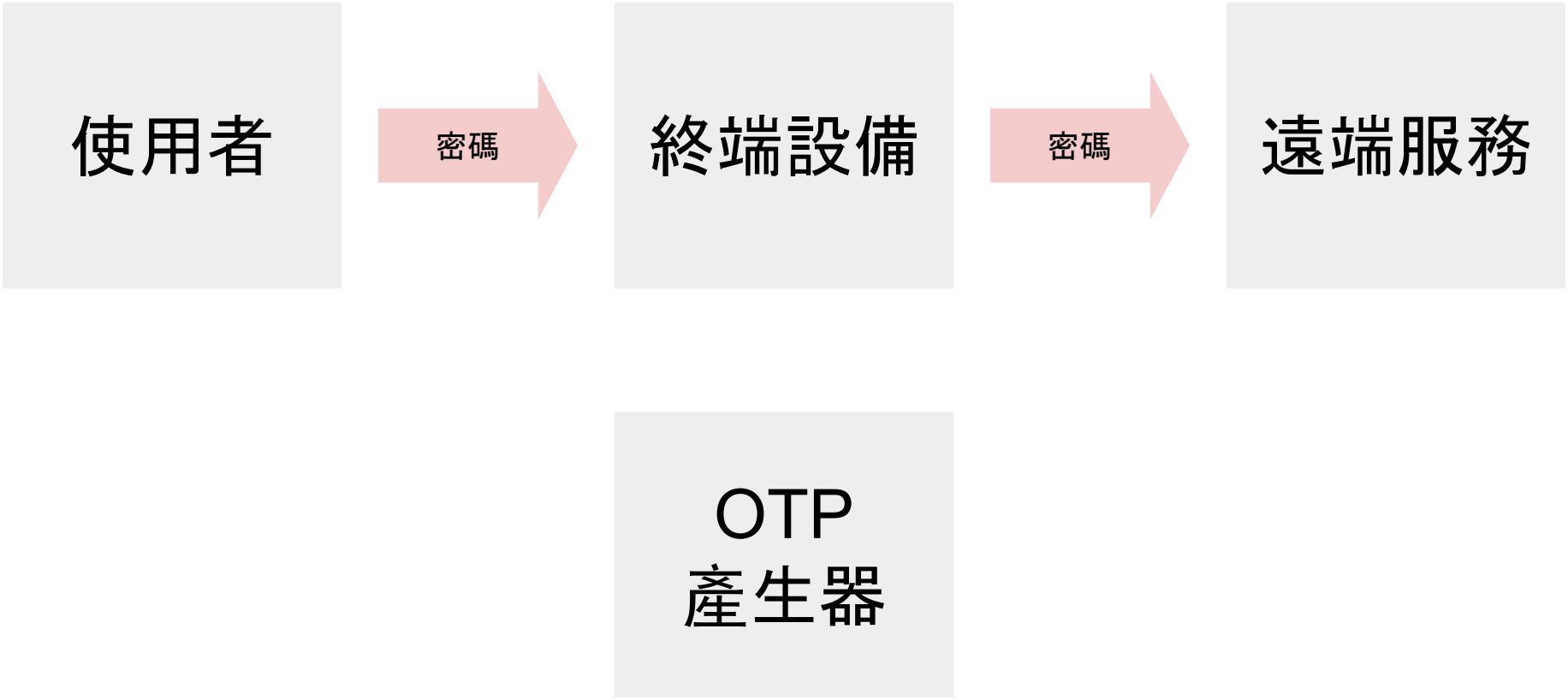
- 主要的資安事件大概有兩類:
  a. 針對使用者
     - 弱身份認證實踐, 搭配社交工程、各種釣魚, 目標在於使用者
  b. 針對服務方
     - 直接進行攻擊伺服器架構、網路, 目標在於服務本身

- 前者佔據絕大部分, 造成損失之大愈來愈不可忽視

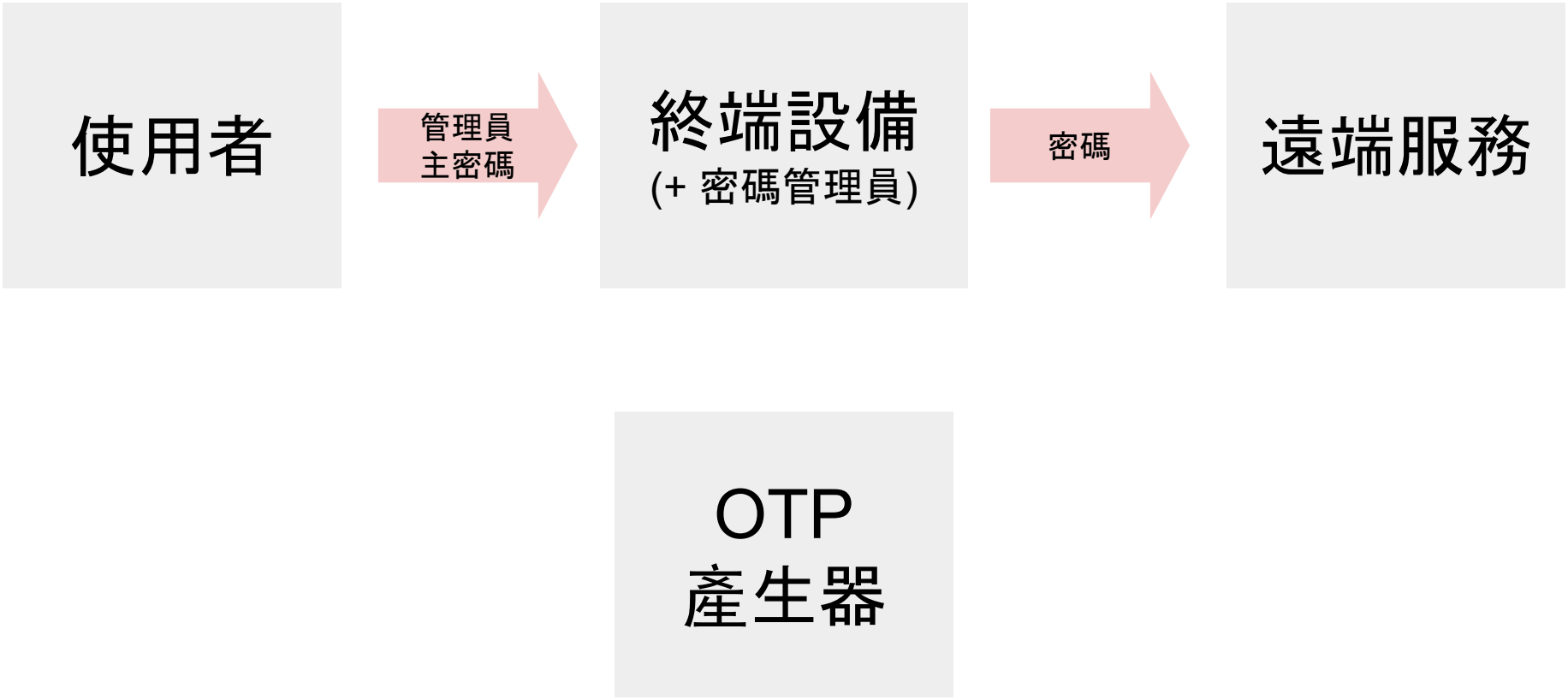
# 於是 FIDO 聯盟成立了 (2012-)

- FIDO = Fast IDentity Online
- 跨平台的公開標準, 處理線上身份認證
- U2F、UAF 兩套標準文件
- W3C working drafts
- 大量成員 https://fidoalliance.org/membership/members/

# 傳統密碼 + OTP 使用者體驗



# 傳統密碼 + OTP 使用者體驗

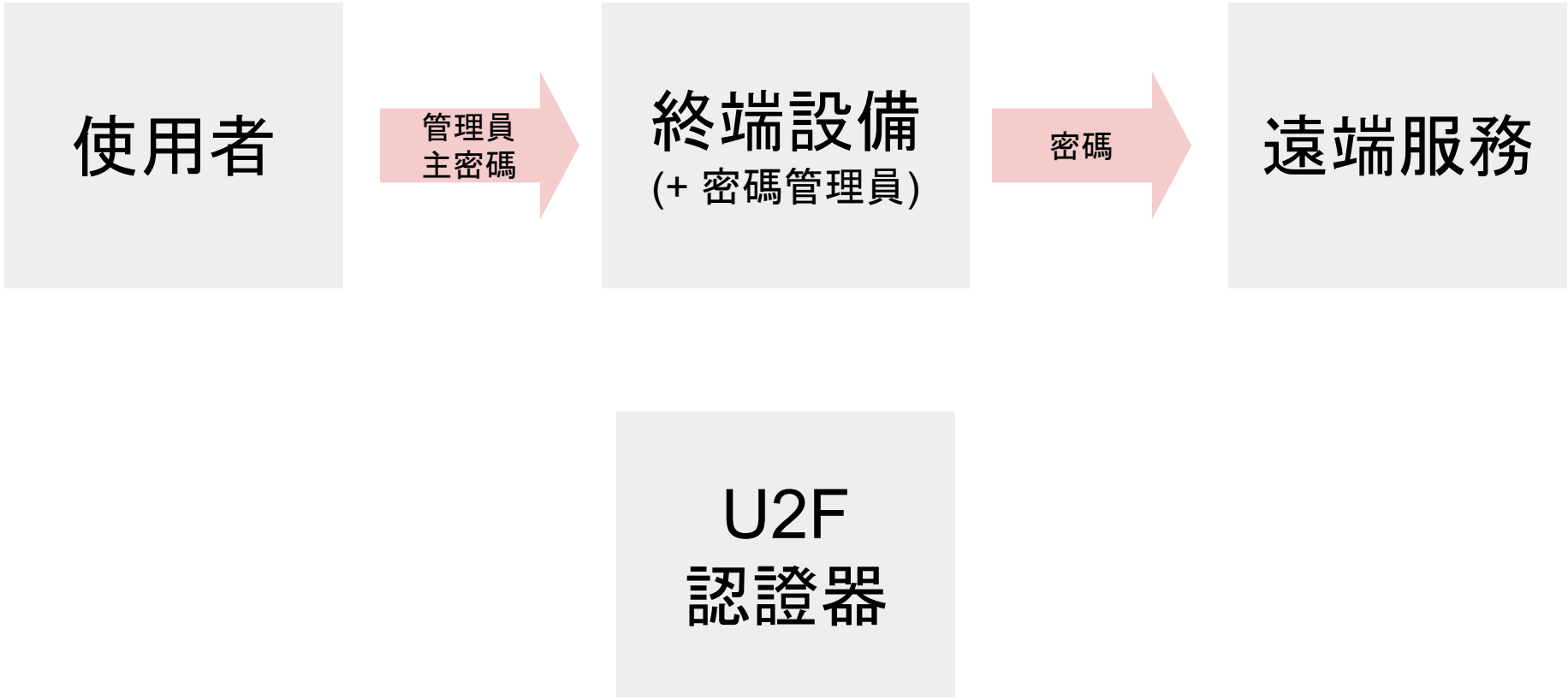


# 傳統密碼 + OTP 使用者體驗

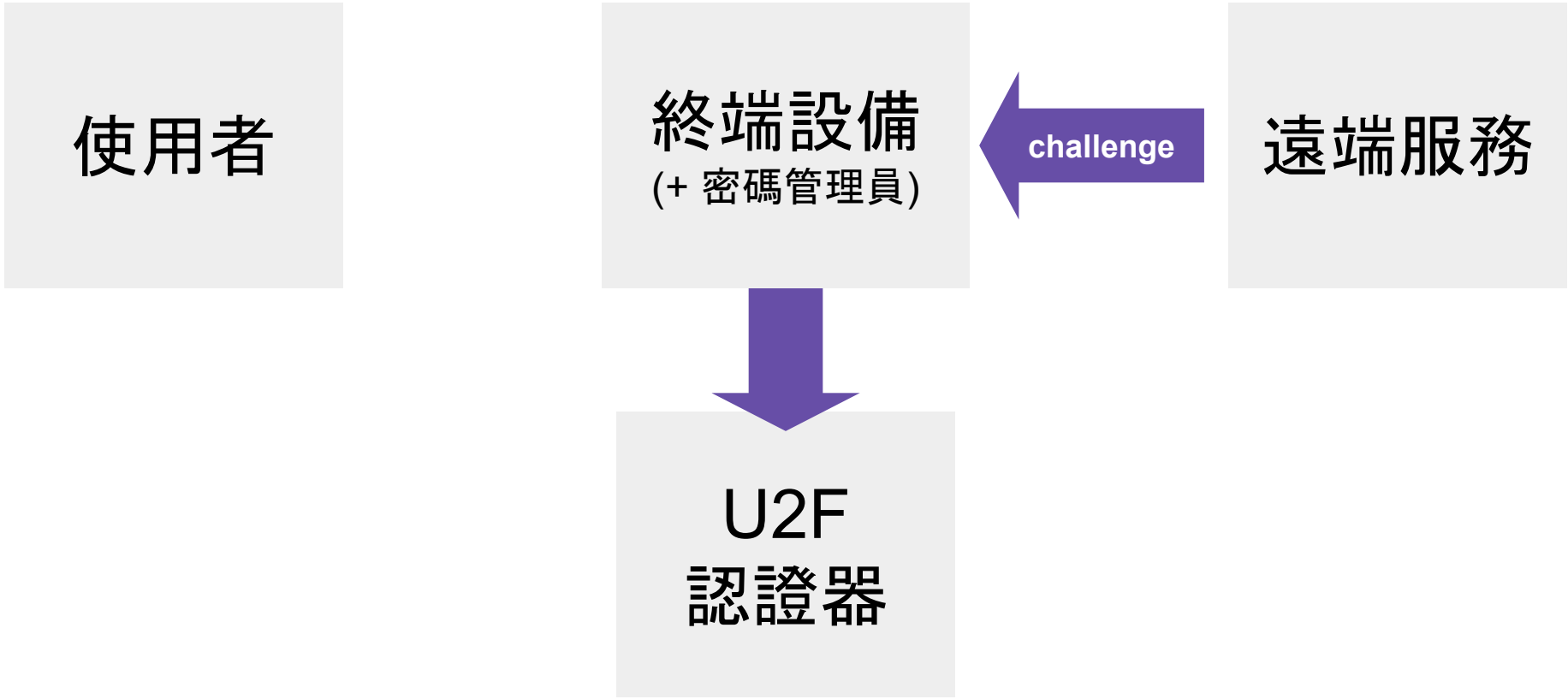
# 傳統密碼 + OTP 使用者體驗

# 傳統密碼 + OTP 使用者體驗

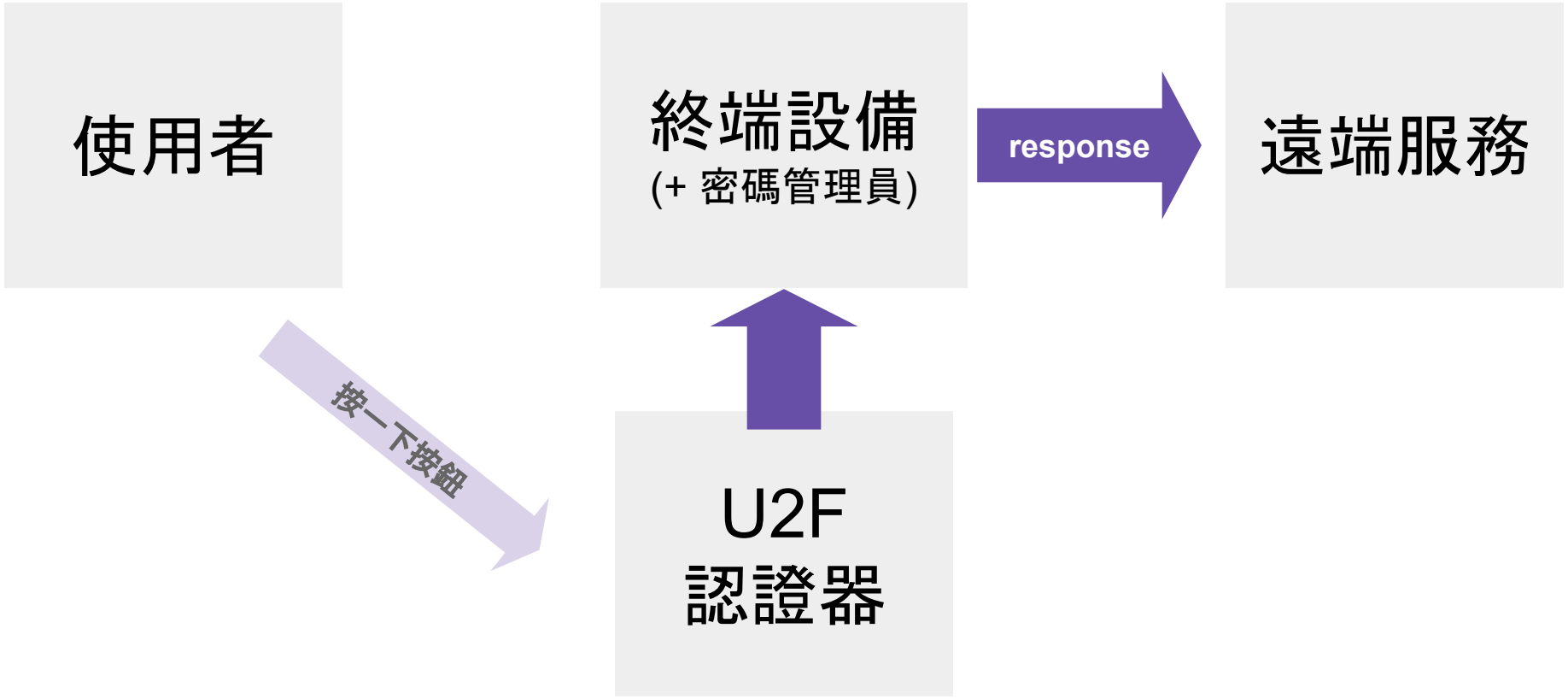
# 傳統密碼 + U2F 使用者體驗



# 傳統密碼 + U2F 使用者體驗

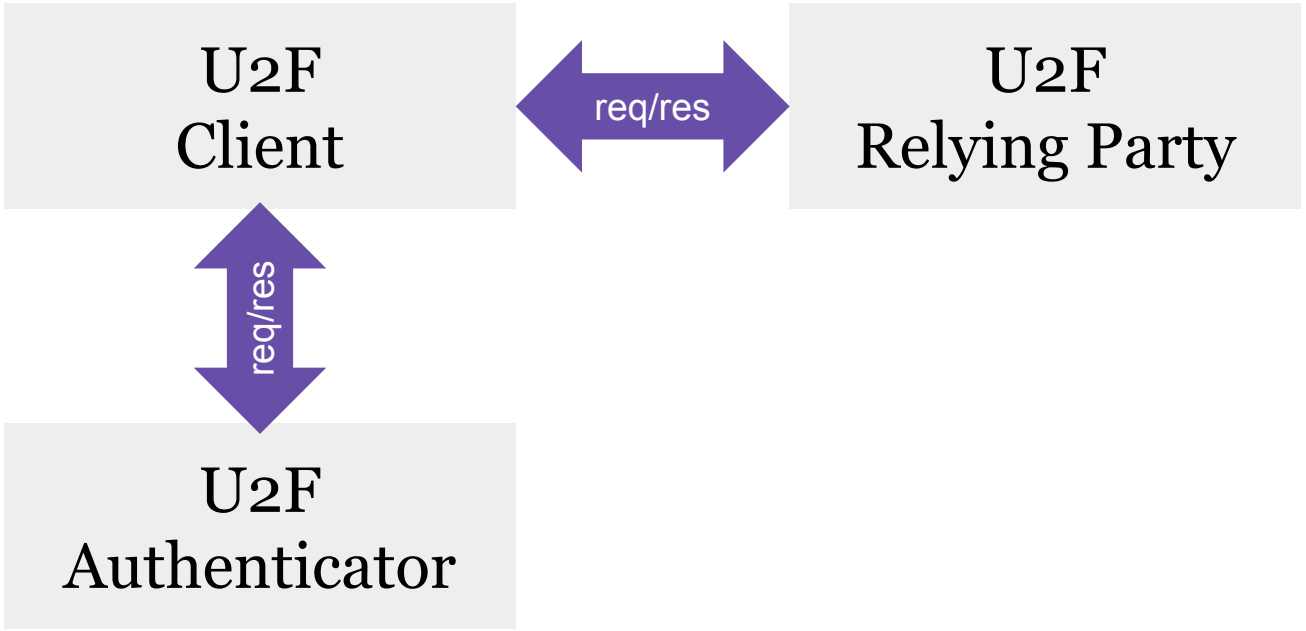


# 傳統密碼 + U2F 使用者體驗



# U2F 框架下的三種角色

- U2F Relying Party (網站)
- U2F Client (瀏覽器)
- U2F Authenticator (使用者端, 外部或內建的安全裝置)
  - 又稱 U2F Token / U2F Device / U2F Security Key



# 天使(魔鬼？)藏在細節裡

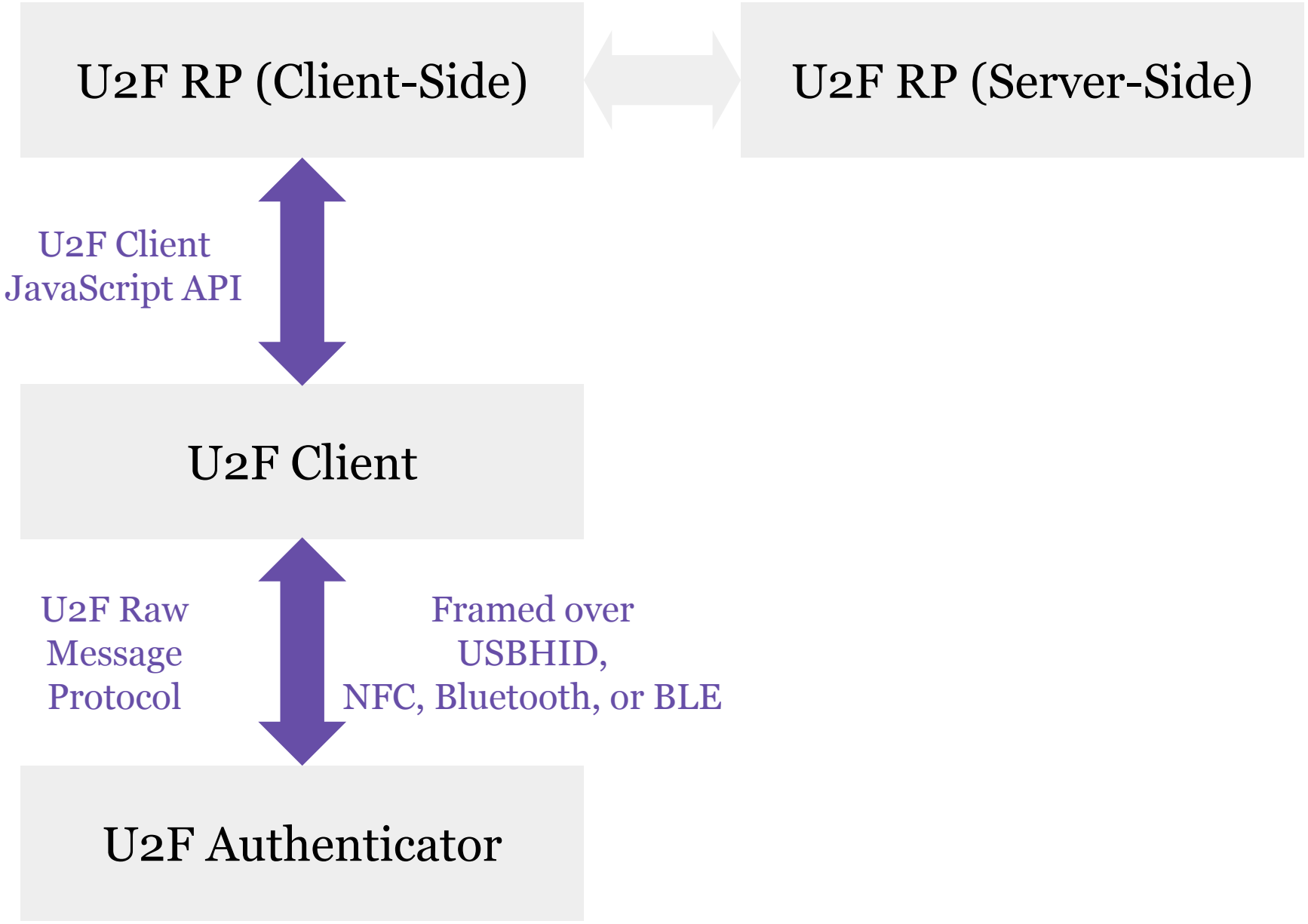
- 不想洩漏秘密資訊 => 使用橢圓曲線公鑰數位簽章
- 不要繁瑣使用者體驗 => 規範元件間標準介面, 不需要安裝額外的 driver, 所有訊息都可以自動傳遞
- 提升匿名性 => 每次註冊都產生完全不同的公私鑰配對, 每一對公私鑰都有一個 key handle 唯一地標誌

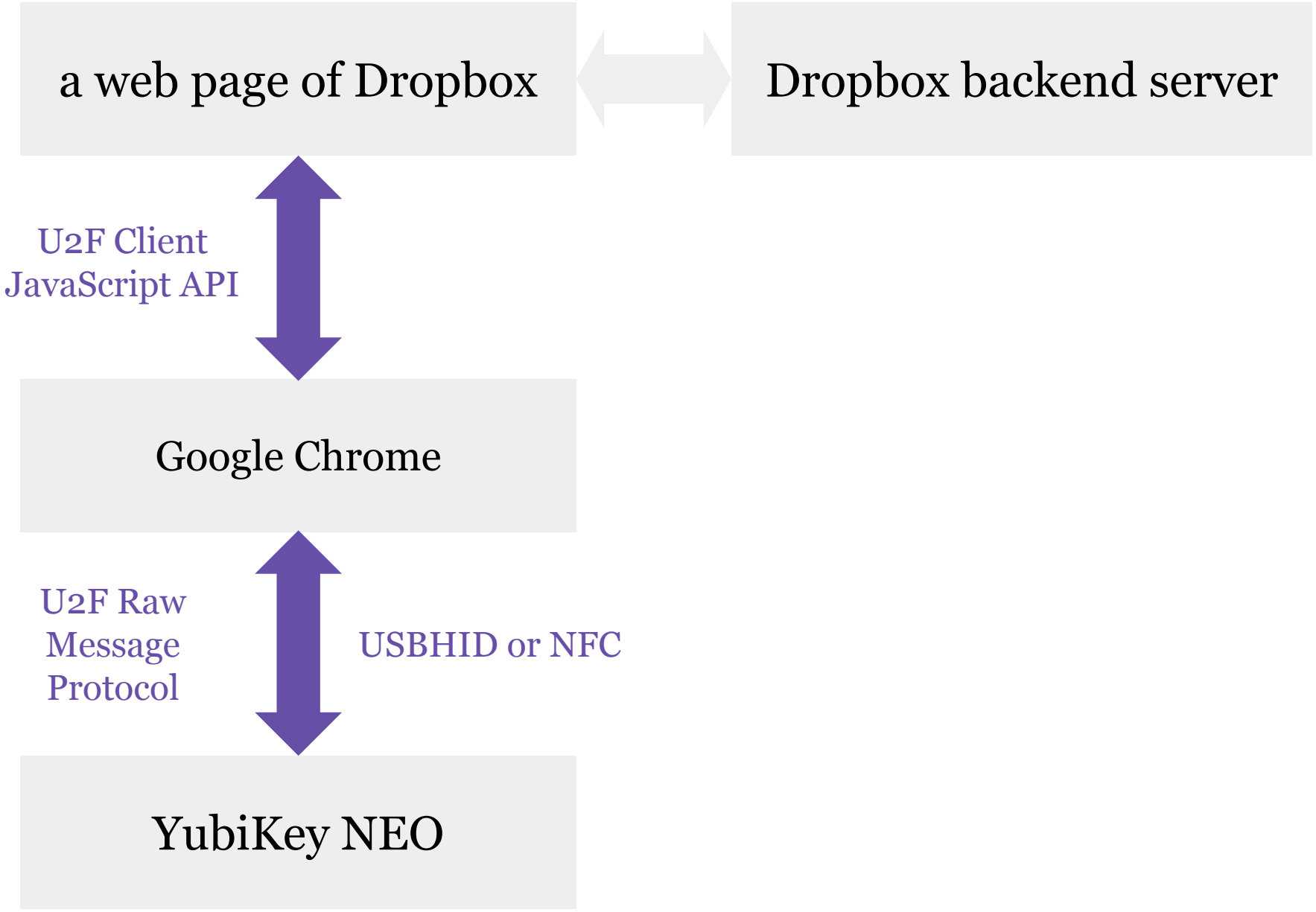
# 天使(魔鬼？)藏在細節裡

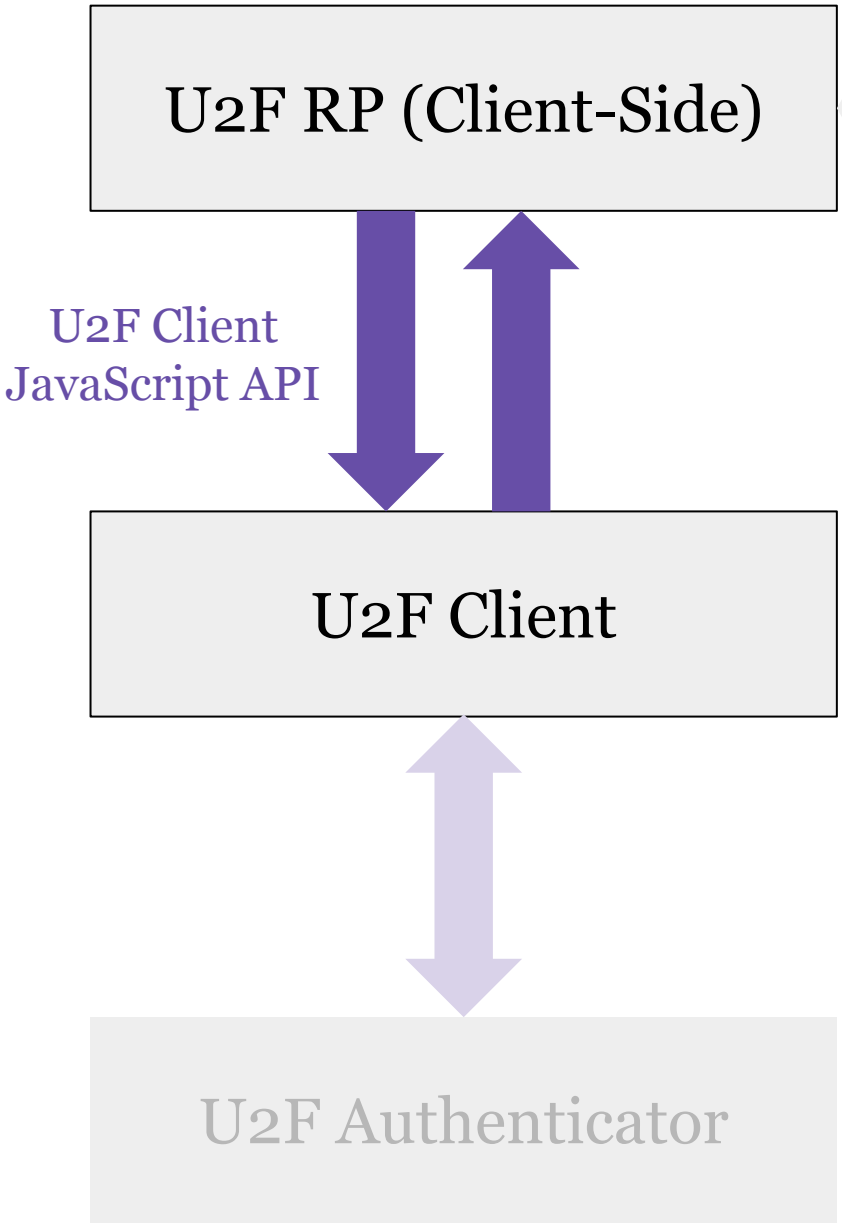
- 自動化釣魚防護 => 將 AppID 編碼進 challenge-response 訊息, 客戶端需依照規範的演算法, 計算發出信息的網頁的合法性。每把公私鑰都和一個 AppID 唯一地綁定在一起
- 防止 replay attack => 所有 challenge-response 訊息都有 nonce 編碼在內, 無法重用

# 天使(魔鬼？)藏在細節裡

- 對抗中間人攻擊 => 將 ChannelID (Token Binding) 資訊編碼進 challenge-response 訊息
- 偵測裝置複製、金鑰竊取的情況 => 將一個嚴格地增的 authentication counter 編碼進 challenge-response 訊息
- 整個流程完全不需要 trusted third party


U2F RP (Client-Side)
U2F RP (Server-Side)
U2F Client
JavaScript API
U2F Client
U2F Raw
Message
Protocol
Framed over
USBHID,
NFC, Bluetooth, or BLE
U2F Authenticator

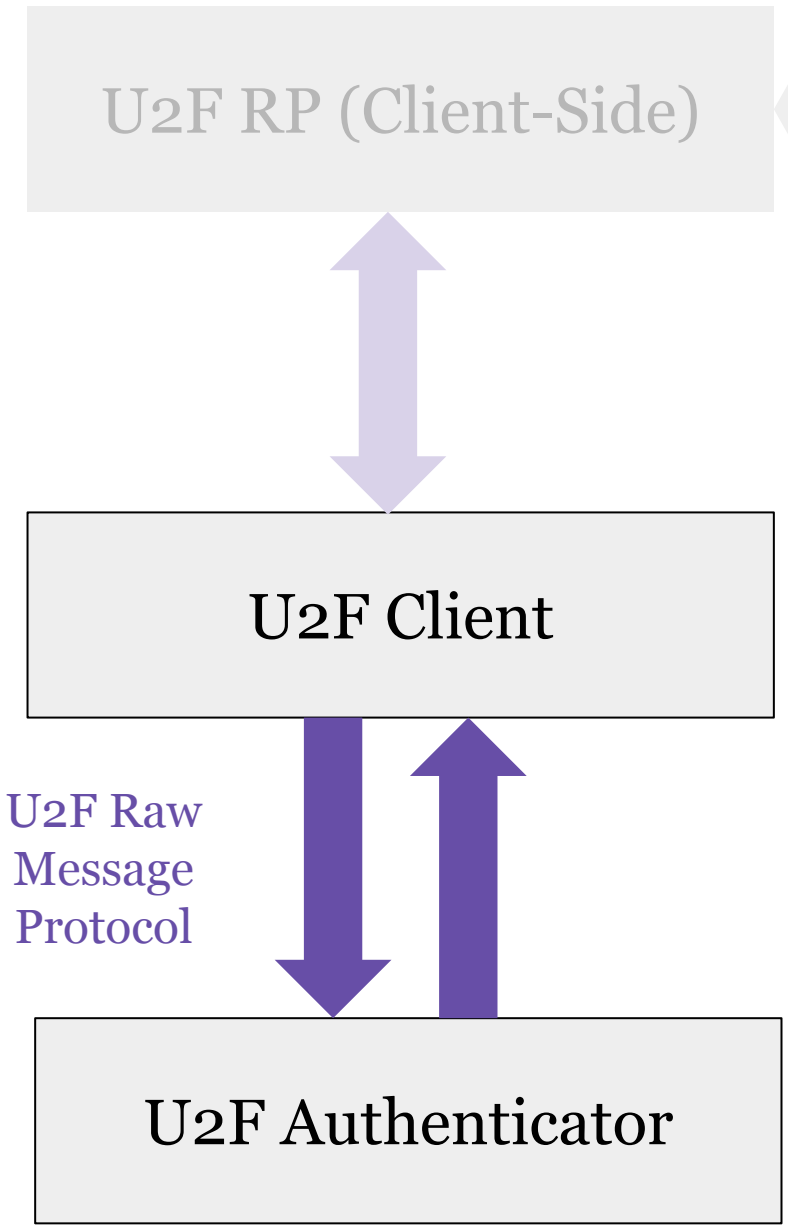

a web page of Dropbox
Dropbox backend server
U2F Client
JavaScript API
Google Chrome
U2F Raw
Message
Protocol
USBHID or NFC
YubiKey NEO




Two types of requests & responses:

1. Device Enrollment/Registration
   a. 檢查裝置是否已註冊過
   b. 請裝置產生新的 key pair
   c. 將 public key 和其他訊息回傳

2. Identity Assertion/Authentication
   a. 請裝置對一個挑戰訊息簽章
   b. 將數位簽章和 auth counter 等其他訊息回傳



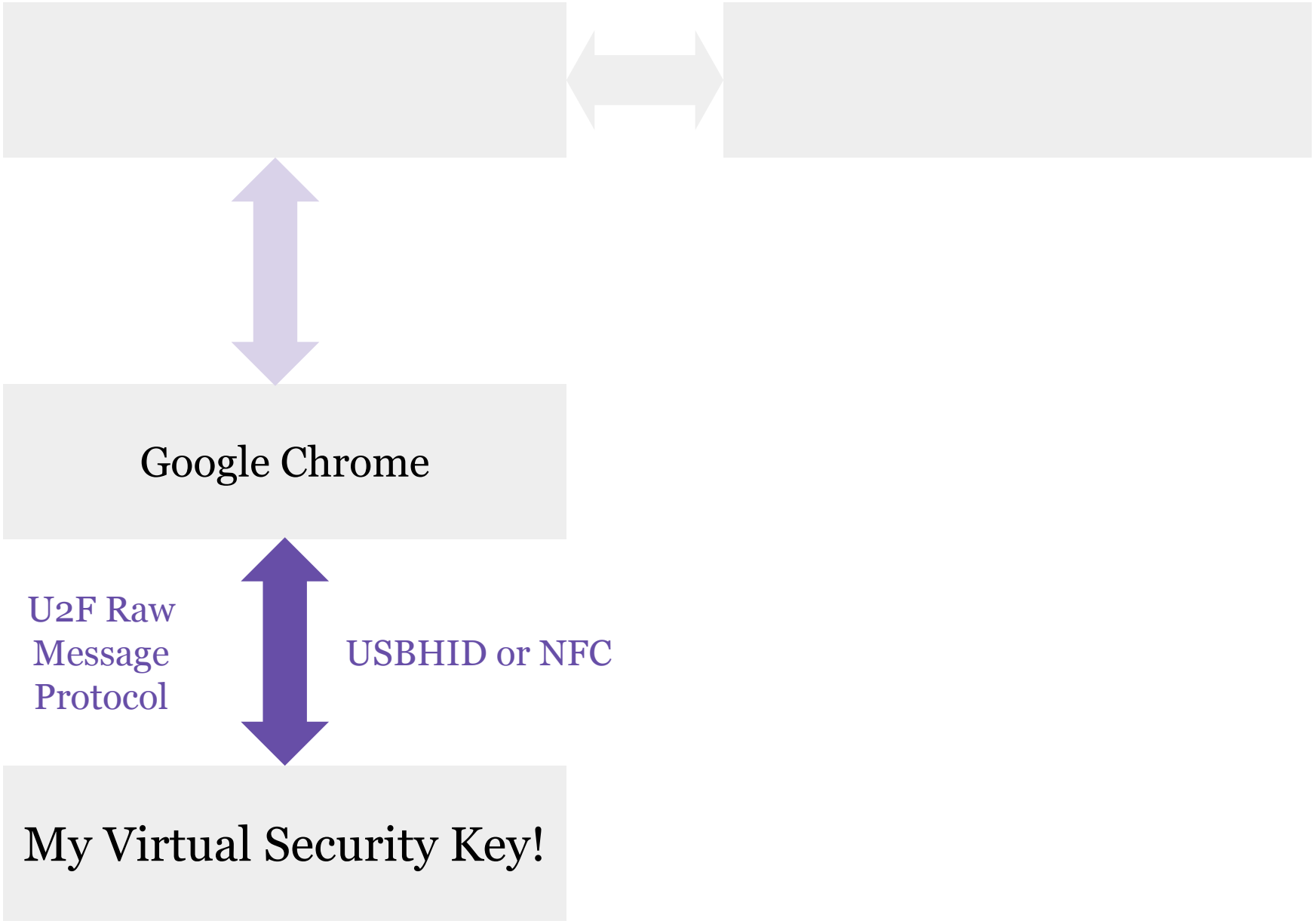

Four APDU commands & responses:

1. GetVersion
2. Register
3. CheckKeyHandle
4. Authenticate

# 把標準文件看了一遍...

- 感覺起來好像很簡單, 但是又有很多模糊不清的地方
- 徹底弄懂 protocol 最直接的辦法, 就是把它實作一遍

- 碰巧實驗室成員最近正在研究各種公鑰數位簽章電路的旁通道攻擊與防禦。大家都挺熟悉這些演算法實作。
- 先從自幹一個 virtual U2F Security Key 開始吧！？


Google Chrome
U2F Raw Message Protocol
USBHID or NFC
My Virtual Security Key!

# 一個 U2F USB 裝置是如何和瀏覽器互動的？

- 其實是先有蛋才有雞的
- 先有實作才有標準 Orz

- Google 搞了個 “Gnubby” security token 和他們自家瀏覽器的支援, 後來加入了 FIDO 聯盟, 發現野人們竟然還不知如何用火。於是決定將這項工作改名為 U2F 並撰寫文件, 成為 FIDO 的第二重因素認證標準。

→ 文件沒說清楚的部分, Google 大神實作的就是規範（！？

# 看看 Google Chrome 的實作

- 從 chromium 原始碼一窺究竟
- https://chromium.googlesource.com/chromium/src
- repo 很大, 不要輕易嘗試 git clone

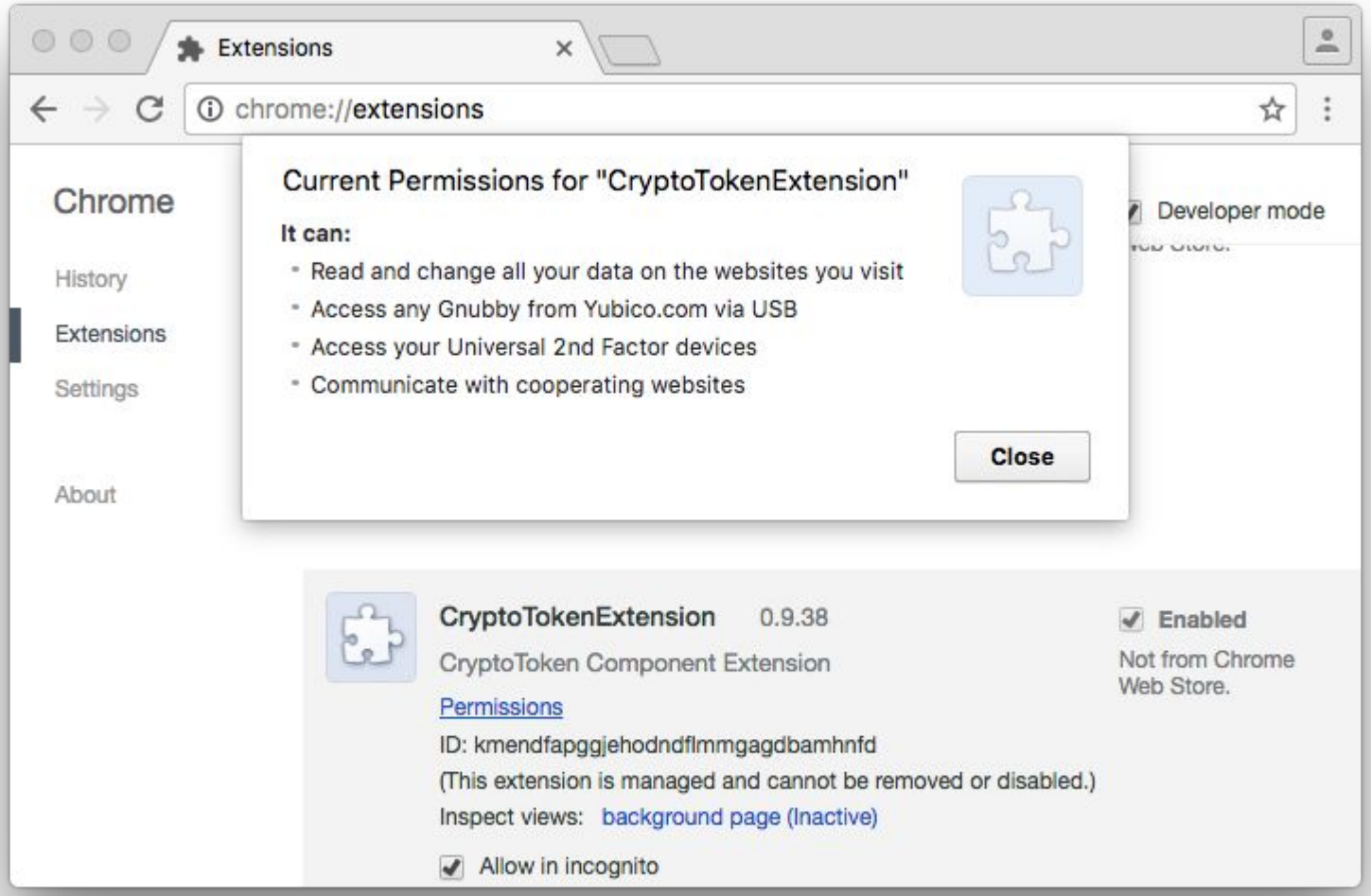
# Google Chrome 的 U2F Client

- Chrome 裡內建的 U2F Client 邏輯, 完全只用 JavaScript 實現, 以一個 extension 形式內建於 Google Chrome
- 這個 extension 名為 CryptoTokenExtension, 而 ID 為 `kmendfapggjehodndflmmgagdbamhnfd` 對上提供這樣的 API:

```
var p = chrome.runtime.connect( ID );
p.postMessage( <request> );
p.onMessage.addListener( <response_callback> );
```

- 對下會去尋找可用的 U2F 裝置並且發出相對應的指令
- 原始碼位於 `/chrome/browser/resources/cryptotoken`

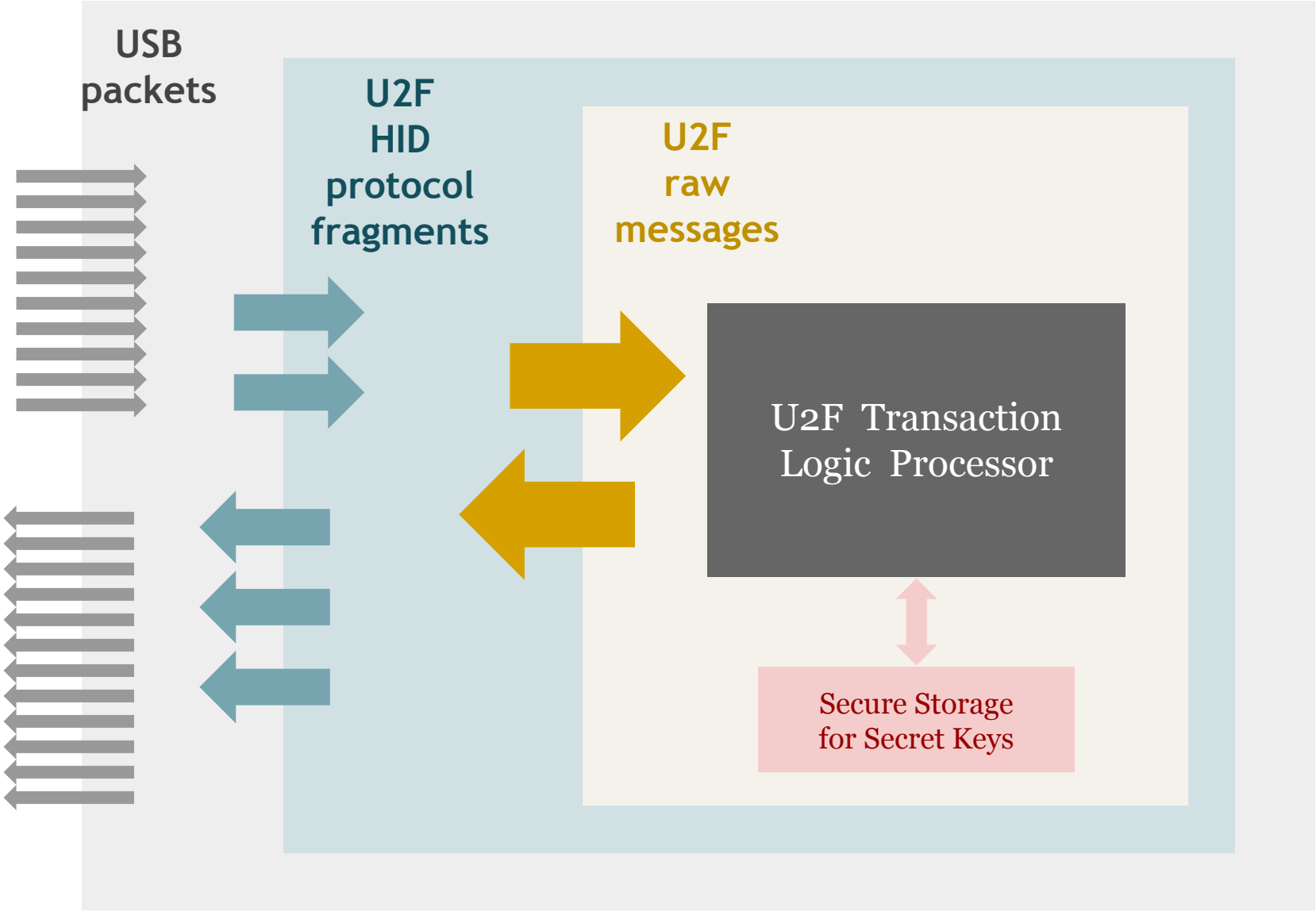
```
google-chrome
--show-component-extension-options
```



# Google Chrome 如何和 U2F USB 裝置互動？

- Chrome 裡內建的 U2F Client 和硬體互動，是透過他自己的 chrome.hid 與 chrome.usb 這兩個 JavaScript APIs
- 不同作業系統有不同的原生 C++ 實作，例如看見 `#include <linux/hidraw.h>` 就明白是在 Linux 上想找 HID 裝置

# 一個 U2F USB 裝置如何接收來自瀏覽器的請求？



# 如何使我的程式被辨識為一個 U2F 裝置？

- 我不想從 USB controller 開始寫起
- 慢著, 從 chromium 原始碼看來, Google Chrome 也並非直接和 USB 裝置通訊
- 以 Linux 為例, 由於 kernel 有實作 HID bus 的介面, 上層應用當然不需要自己處理底層 USB 封包的編解碼, 而是可以直接用 HID reports 和裝置互動。

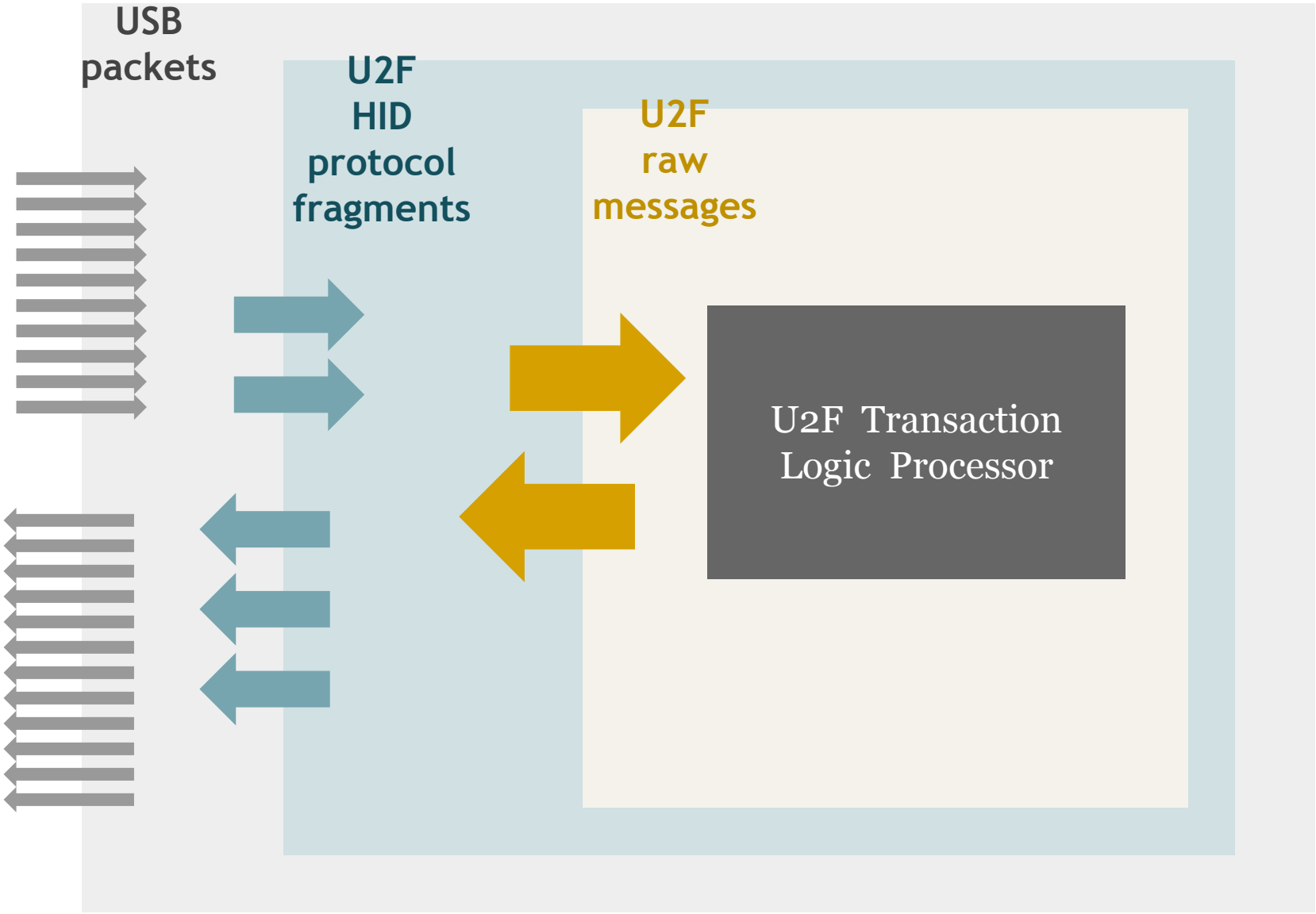
# Linux User-Space HID I/O Driver

- 為 Linux 撰寫驅動程式, 也不用從最底層 USB 開始做
- Kernel 提供的 HID subsystem 讓我們能通過 `/dev/uhid` ABI 掛上自己撰寫的 user-space I/O driver for HID device！

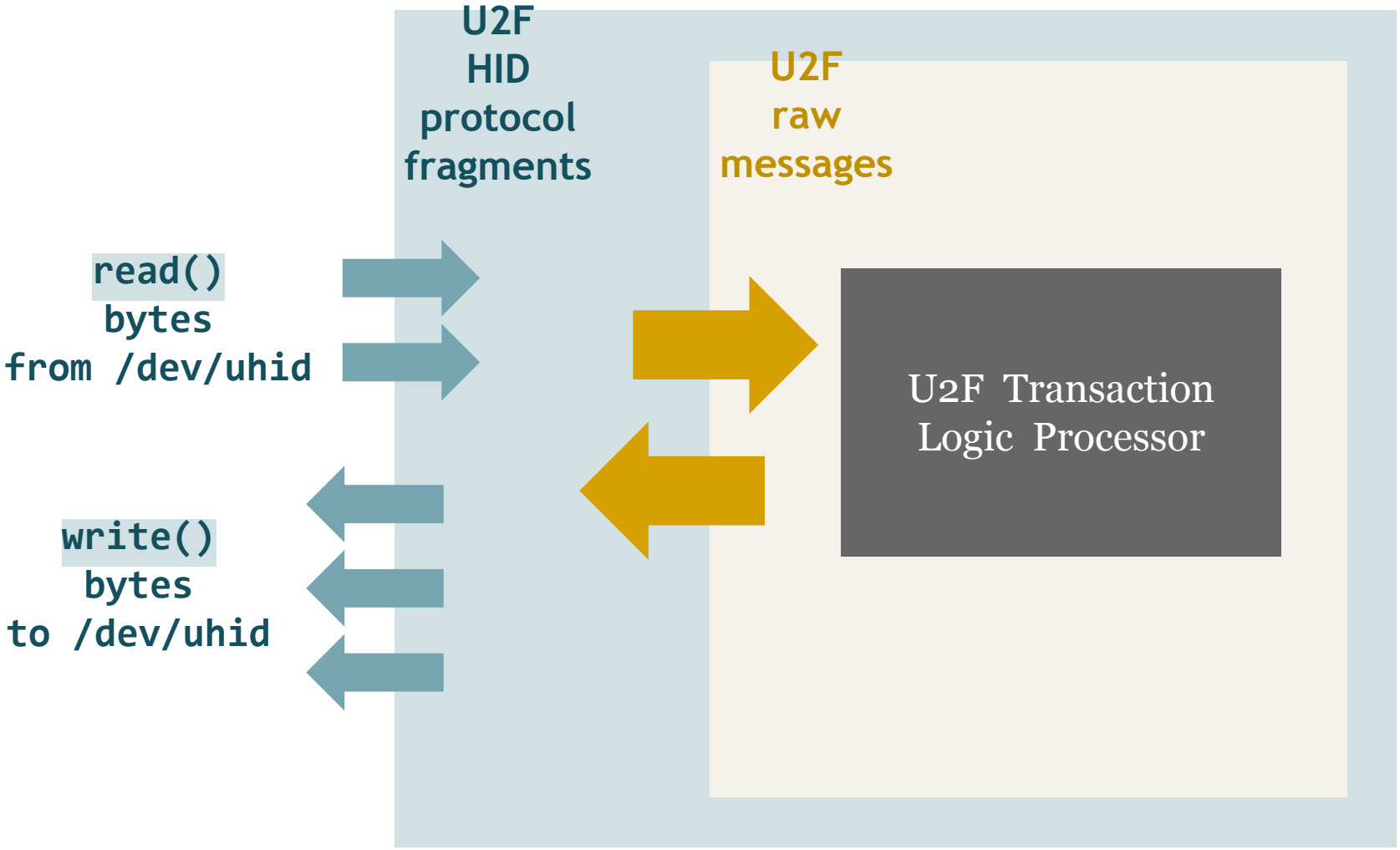
# `open("/dev/uhid", O_RDWR)` for the win!

- 以 read() 讀取 kernel → user-space 的 UHID 事件
  - 有應用想要打開 USB 連線、和我們通訊
  - 來自 host 的 HID report 請求內容
- 以 write() 來發送 user-space → kernel 的 UHID 事件
  - 回覆來自 host 的 HID report 請求

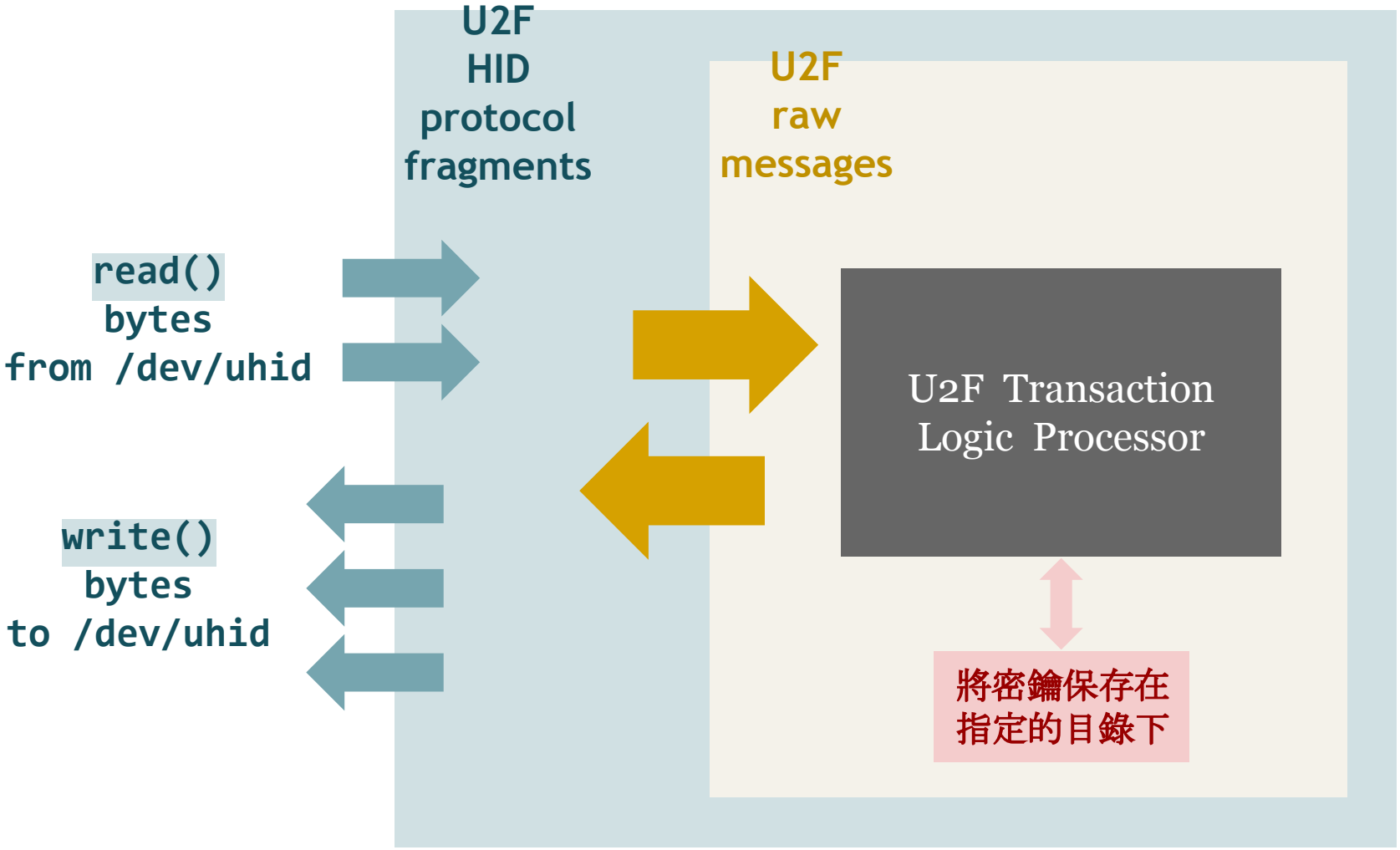
# 在 Linux 上實現 USBHID 的 U2F 裝置



# 在 Linux 上實現 USBHID 的 U2F 裝置

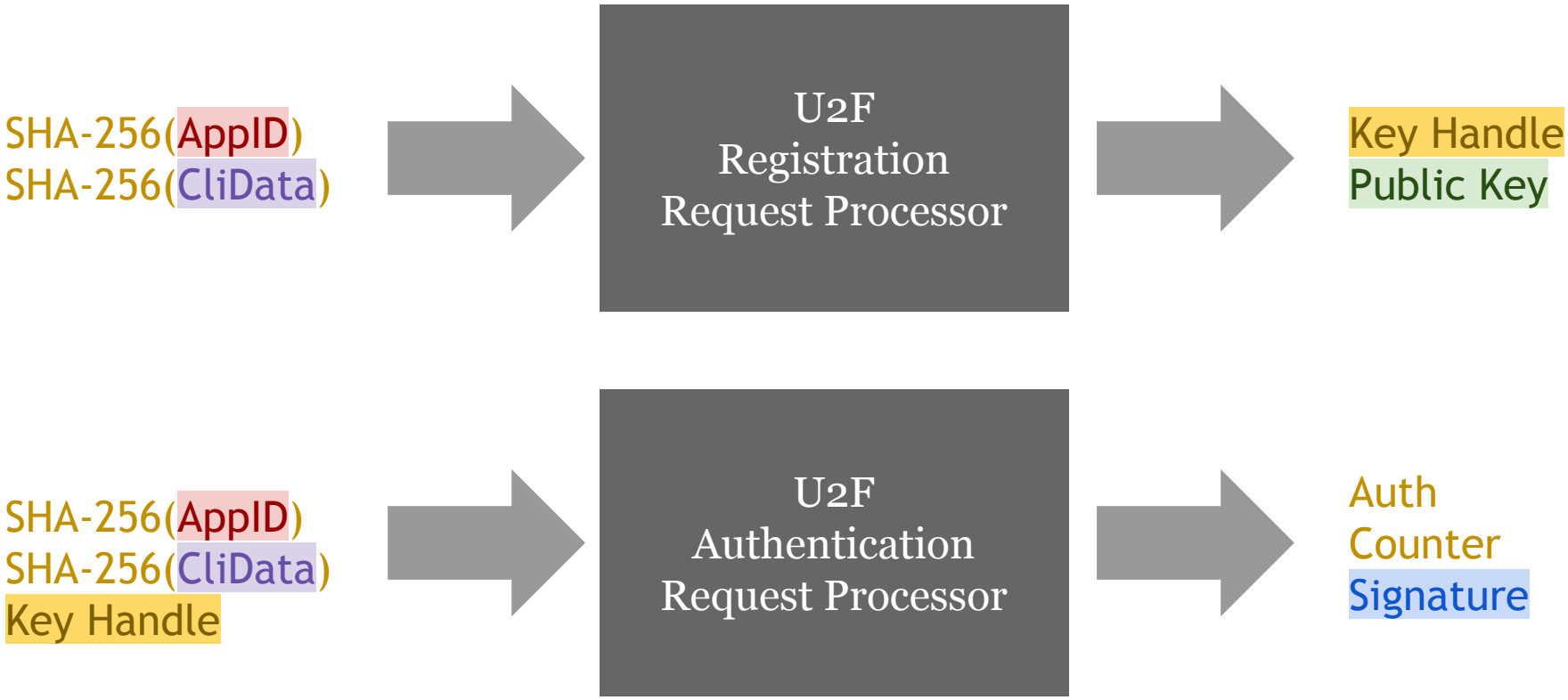


# 在 Linux 上實現 USBHID 的 U2F 裝置



AppID = the identity remote server claims

CliData = (
the operation name
a challenge nonce generated by server
the web origin the client actually sees
channel id info
)



# Key Handle

- 對於 server 而言 key handle 是使用者 public key 的索引

- 對於 authenticator 而言 SHA-256(AppID) 與 key handle 兩份資訊在一起就是個索引, 可以查詢出一對 key pair

```
    (64 bytes)          (32 bytes)       (32 bytes)
DEVICE_MASTER_KEY  =  RNG_SECRET_SEED + SELFCHECK_KEY
```

```
# 每次註冊, 都以 stateless 方式產生新的 key handle 與 public key


hashed_appid     = ...
nonce            = GET_RANDOM_BYTES(32)

self_checksum    = HMAC_SHA_256(SELFCHECK_KEY, hashed_appid + nonce)
key_handle       = nonce + self_checksum

new_private_key = CRYPTO_SECURE_PRNG(RNG_SECRET_SEED, nonce)
new_public_key  = private_to_public(new_private_key)


# 可以生成無限數量的公私鑰配對
```

# `concise/v2f.py` on GitHub

- ~1000 lines of dirty code
- All Python
- Zero external dependency (it only uses Python builtins)
- Working on Linux only, for now
- Good enough to hack with U2F clients

- Still pre-alpha stage
- Serious refactoring & docs are needed

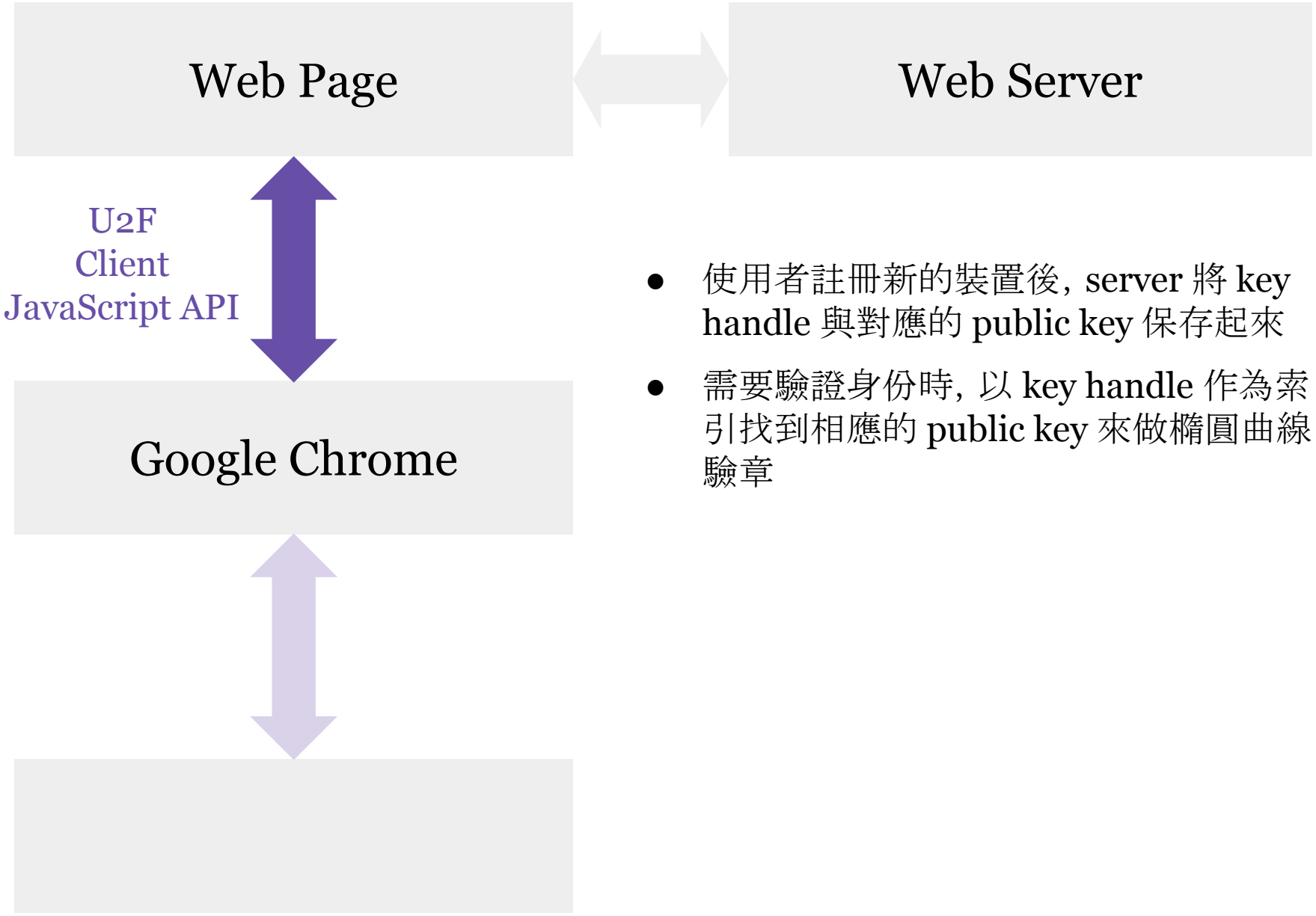
Let's see `v2f.py` in action

# 當然, 硬體解決方案比較安全

- 硬體提供較好的保護, 金鑰從來就不需離開硬體, 讓你可以安全地將他「帶著走」
- 軟體 virtual security key 需要將金鑰存放在檔案系統... 比較可能被攻擊者取走

- 解鎖 security key 時, 要求使用者證明身份？
  輸入 passphrase 以解鎖 keychain 內的密鑰？
  (原本按一下按鈕、或輸入 yes 只是表示有人在而已)

那麼，伺服器端呢？




- 使用者註冊新的裝置後, server 將 key handle 與對應的 public key 保存起來
- 需要驗證身份時, 以 key handle 作為索引找到相應的 public key 來做橢圓曲線驗章

```
# Pseudo-code of U2F server-side registration transaction



APPID     = "https://jong.sh"

uid       = ...
keys_info = fetch_keys_info_from_database(uid)
nonce     = ...

request  = generate_registration_request(APPID, nonce, keys_info)
response = get_response_from_client_side(response)
result   = process_registration_response(APPID, nonce, response)

db_result = sync_new_key_info_to_database(uid, result)



# If both `result` and `db_result` are good
# the registration transaction succeeded
```

```
# Pseudo-code of U2F server-side authentication transaction



APPID     = "https://jong.sh"

uid       = ...
keys_info = fetch_keys_info_from_database(uid)
nonce     = ...

request  = generate_authentication_request(APPID, nonce, keys_info)
response = get_response_from_client_side(response)
result   = process_authentication_response(APPID, nonce, response)

db_result = sync_old_key_info_to_database(uid, result)



# If both `result` and `db_result` are good
# the authentication transaction succeeded
```

# `concise/lightu2f.py` on GitHub

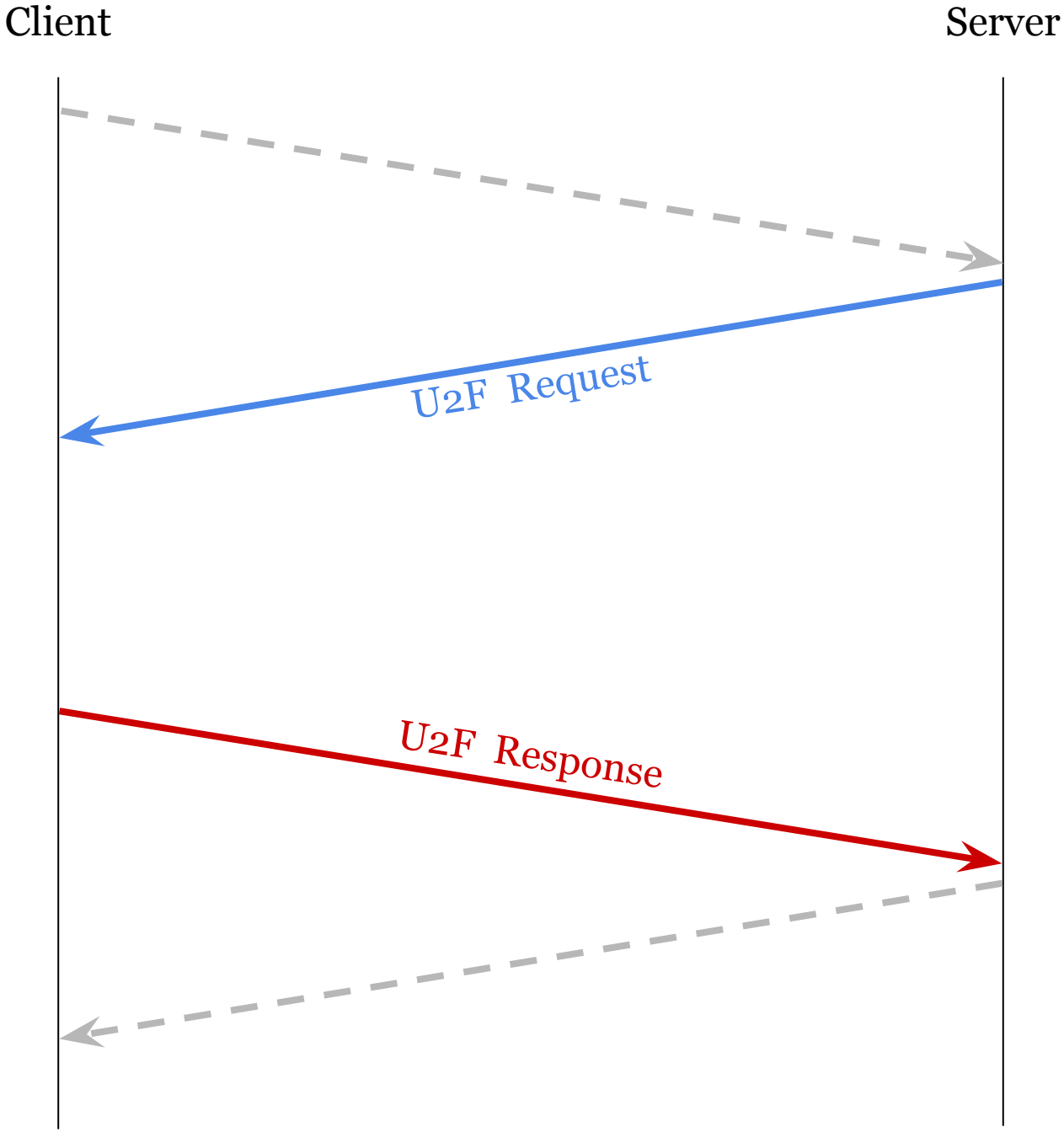
- Contains the four most important algorithms
  - `generate_registration_request()`
  - `process_registration_response()`
  - `generate_authentication_request()`
  - `process_authentication_response()`
- They are pure stateless functions
- ~600 lines of dirty code
- All Python & zero external dependency

# Multi-step Transaction Processing In Practice

```
APPID      = "https://jong.sh"
uid        = ...
keys_info = fetch_keys_info_from_database(uid)
nonce      = ...

request   = generate_registration_request(APPID, nonce, keys_info)
response = get_response_from_client_side(response)
result    = process_registration_response(APPID, nonce, response)

db_result = sync_new_key_info_to_database(uid, result)
```

# Multi-step Transaction Processing In Practice

```
APPID      = "https://jong.sh"
uid        = ...
keys_info = fetch_keys_info_from_database(uid)
nonce      = ...

request  = generate_registration_request(APPID, nonce, keys_info)
response = get_response_from_client_side(response)    # ASYNC !!!
result   = process_registration_response(APPID, nonce, response)

db_result = sync_new_key_info_to_database(uid, result)
```


Client
Server
U2F Request
U2F Response

# Two ways to do this

- A stateful solution:
  - Server-maintained sessions
  - Clients hold session IDs
- A stateless solution: (forget everything)
  - Pass information to clients, and let them return it
  - Public data: HMAC-SHA-256
  - Confidential data: AES-128-GCM
  - Don't forget to add a timestamp or counter

Let's see `lightu2f.py` in action

# U2F 相較於 OTP 的優勢

- 橢圓曲線公鑰數位簽章, 不將秘密資訊交給別人
- 標準規範各個元件需要支援的統一介面, 較容易看見裝置開發商、平台開發者支援它
- 每次註冊, 就算同一個網站, 也會自動生成不同的金鑰
- 釣魚防護、中間人攻擊、簡易的金鑰複製(竊取)偵測

# U2F 的幾點阻力

- 使用者需要購買額外的硬體才行
  - e.g., YubiKey for $18
- 使用者端的應用平台需要跟上標準, 實作 U2F Client
  - Firefox 等其他瀏覽器尚未實作完畢
  - 缺乏作業系統直接的支援, 在 Android / iOS 需要額外的 app
- 要重新教育使用者, 培養新的習慣
- 服務方需要部屬相關的程式碼、再次規劃資料庫
  - 和為了支援 OTP 所要做的更動量差不多

Questions ?