



Rapport Projet intégrateur: Systèmes RKS et SRP-ECDHE

Lucas MARACINE, Loic TESTA
Institut Mines Télécom-Télécom Paris Tech

26 juin 2024

Résumé

Ce rapport est une présentation générale et non exhaustive des systèmes RKS, ainsi qu'une ébauche d'implémentation d'un protocole SRP basé sur ECDHE et non sur un simple Diffie Hellman. Ce rapport n'a pas la prétention d'être un guide complet sur les systèmes RKS mais d'en présenter des points clés.

L'entièreté du code et des implémentations sont disponibles à l'adresse suivante :
<https://github.com/Foxanivia/Projet-integrateur-RKS.git>

Table des matières

1 Etat de l'art	3
1.1 Fonctionnement général	3
1.1.1 Code fixe et code tournant	3
1.1.2 RKE mono et bi directionnel	3
1.1.3 Méthodes de modulation	4
1.2 Implémentation de test	6
1.2.1 Implémentation RollingCode	6
1.2.2 Implémentation Challenge	9
1.3 Attaques possibles	11
1.3.1 Bruteforce	11
1.3.2 Replay attaque	11
1.3.3 Rolljam attaque	11
1.3.4 Rollback attaque	11
2 Implémentation d'un algorithme d'authentification basé sur SRP	13
2.1 Théorie, idées	13
2.1.1 Secure Remote Password (SRP)	13
2.1.2 Choix des courbes elliptiques comme moyen d'authentification	18
2.2 Implémentation sous python d'une solution possible	21

3 Démonstration	28
3.1 Idée de base	28
3.2 Démonstration PC/RPI via câble Ethernet direct	30
3.3 Ouverture : Utilisation de la clé partagée ECDHE pour chiffrer toutes les communications SRP	34
4 References	35

1 Etat de l'art

1.1 Fonctionnement général

1.1.1 Code fixe et code tournant

Les systèmes de télécommande sans clé (RKE - Remote Keyless Entry) utilisent généralement deux types de codes pour l'authentification et l'autorisation : le code fixe et le code tournant.

Code fixe : Le code fixe est un code unique et statique qui est envoyé à chaque fois que la télécommande est utilisée. Ce type de code est vulnérable aux attaques par enregistrement et relecture (replay attack), où un attaquant peut enregistrer le signal et le rejouer pour accéder au véhicule.

Code tournant (Rolling Code) : Le code tournant utilise un algorithme pour générer un nouveau code pour chaque transmission. Les deux dispositifs (la télécommande et le récepteur dans le véhicule) doivent être synchronisés pour utiliser le même algorithme. Cette technique améliore la sécurité en empêchant les attaques par enregistrement et relecture.

1.1.2 RKE mono et bi directionnel

Les systèmes RKE peuvent être séparés entre les systèmes monodirectionnels et les systèmes bidirectionnels. Ces deux catégories peuvent présenter de nombreuses différences, en termes de coût technologique, énergétique ou en termes de sécurité.

Caractéristique	Monodirectionnel	Bidirectionnel
Communication	Unidirectionnelle (télécommande → véhicule)	Bidirectionnelle (télécommande ↔ véhicule)
Sécurité	Vulnérable aux attaques par interception	Plus résistant aux attaques grâce à l'authentification mutuelle
Complexité	Simple, facile à mettre en œuvre	Plus complexe, nécessite une synchronisation des échanges
Retour d'information	Aucun retour d'information	Retour d'information visuel ou auditif pour l'utilisateur
Coût	Moins coûteux	Plus coûteux en raison de la technologie avancée

TABLE 1 – Comparaison entre les systèmes RKE monodirectionnels et bidirectionnels

Mono directionnel : Les systèmes RKE monodirectionnels sont les plus simples et les plus couramment utilisés dans les véhicules. Dans ce type de système, la communication se fait uniquement dans une direction, de l'émetteur dans la télécommande vers le récepteur dans le véhicule. L'émetteur ne peut donc pas être challenger, il prouve son identité directement avec un code.

Bi directionnel : Même si les codes roulants permettent de limiter ce risque, les systèmes RKE sont souvent sensibles aux failles de rejeux. Systèmes RKE bidirectionnels Les systèmes RKE bidirectionnels offrent une communication à double sens entre l'émetteur et le récepteur du véhicule. Ce type de système offre d'avantages de flexibilité en termes de fonctionnalité et permet d'améliorer la sécurité du système. L'émetteur après être entré en communication avec le récepteur va se voir soumettre un challenge de ce dernier qu'il lui renverra.

1.1.3 Méthodes de modulation

Les systèmes de télécommande sans clé (RKE) utilisent diverses méthodes de modulation pour transmettre des données entre l'émetteur (télécommande) et le récepteur (véhicule). Les trois types principaux de modulation utilisés dans ces systèmes sont la modulation d'amplitude (ASK), la modulation de fréquence (FSK), et la modulation par sauts de fréquence (FHSS).

Modulation d'Amplitude (ASK) : La modulation d'amplitude (ASK) est une technique où l'amplitude du signal porteur est modifiée en fonction des données numériques à transmettre. En général on utilise 2 symboles l'un avec une amplitude haute représentant le bit 1 l'autre avec une amplitude faible voir nul représentant le bit 0.

— **Avantages :**

- Simplicité de mise en œuvre et de conception.
- Coût réduit des composants nécessaires.

— **Inconvénients :**

- Sensibilité élevée aux interférences et aux bruits, ce qui peut entraîner des erreurs de transmission.
- Moins sécurisé par rapport à d'autres techniques de modulation.

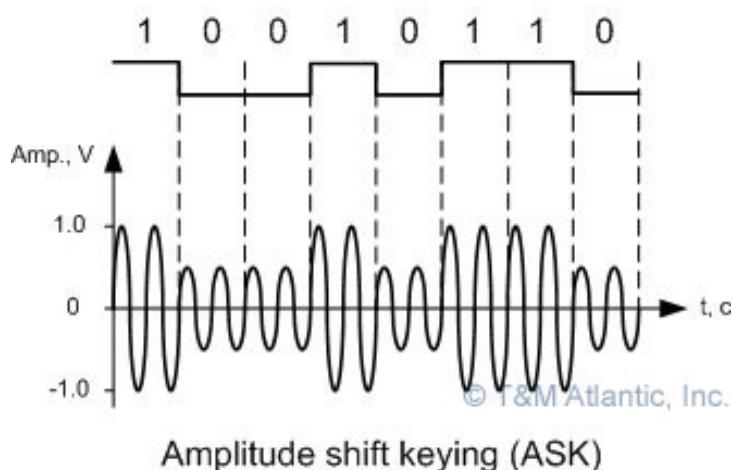


FIGURE 1 – Illustration d'une transmission de donnée en ASK

Modulation de Fréquence (FSK) : La modulation de fréquence (FSK) utilise des variations de fréquence du signal porteur pour représenter les données numériques.

— **Avantages :**

- Meilleure résistance aux interférences et aux bruits comparée à l'ASK.
- Fiabilité accrue des communications.

— **Inconvénients :**

- Conception et mise en œuvre plus complexes.
- Coût plus élevé des composants nécessaires.

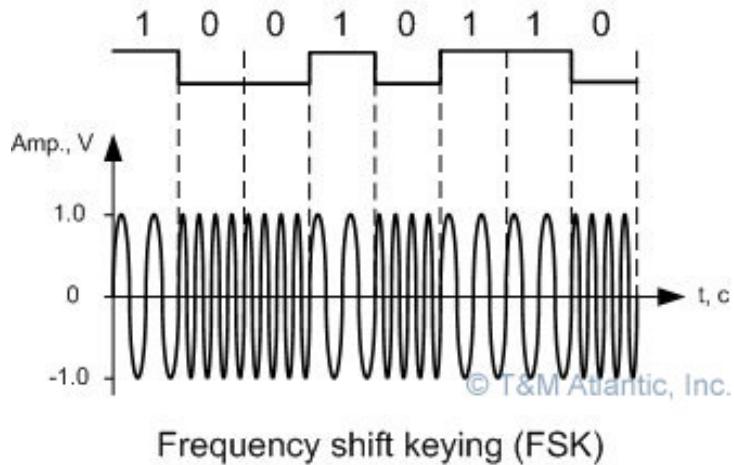


FIGURE 2 – Illustration d'une transmission de donnée en FSK

Modulation par Sauts de Fréquence (FHSS) : La modulation par sauts de fréquence (FHSS) implique de changer rapidement et de manière pseudo-aléatoire la fréquence porteuse selon une séquence connue à la fois de l'émetteur et du récepteur.

— **Avantages :**

- Excellente résistance aux interférences et aux tentatives de brouillage.
- Sécurité accrue grâce à la complexité du schéma de saut de fréquence.

— **Inconvénients :**

- Complexité élevée de la conception et de la mise en œuvre.
- Consommation d'énergie plus importante due aux changements fréquents de fréquence.

Chacune de ces méthodes de modulation présente des avantages et des inconvénients en termes de coût, de complexité, de sécurité et de performance dans les systèmes RKE. Le choix de la méthode de modulation dépend des exigences spécifiques de l'application et des contraintes de conception.

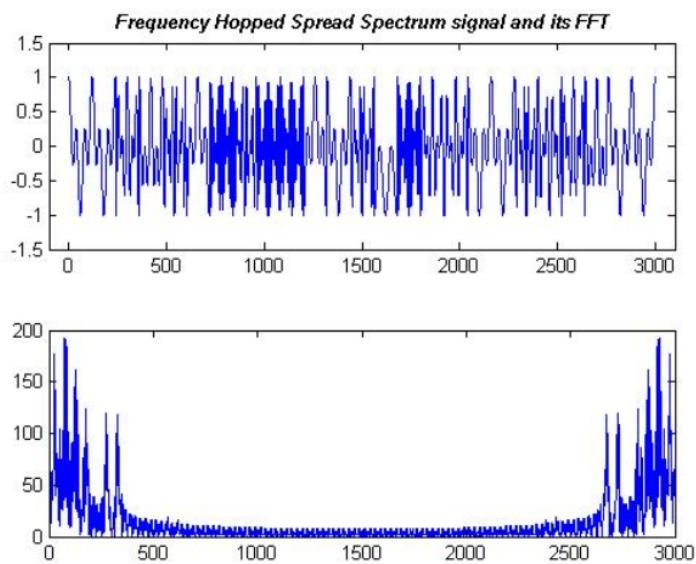


FIGURE 3 – Illustration d'une transmission de donnée en FSK

1.2 Implémentation de test

1.2.1 Implémentation RollingCode

KeeLoq : KeeLoq est un algorithme de chiffrement utilisé dans les systèmes d'entrée sans clé. Il est largement déployé dans les systèmes de sécurité automobile.

- Taille du Bloc : 32 bits
- Taille de la clé : 64 bits
- Nombre de tours : 528

L'algorithme KeeLoq fonctionne en utilisant une technique de chiffrement par décalage de bit et une clé de 64 bits pour générer un code tournant unique pour chaque transmission.

Voyons une implémentation de ce dernier, nous avons ci dessous une classe qui gère les fonctions de chiffrement commune à l'émetteur et au récepteur. L'émetteur et le récepteur sont disponibles sur le GitHub du projet, nous utilisons une version avec la clé pré-partagée. Cette implémentation utilise une manière simple de gérer la désynchronisation.

Bien que largement utilisé, KeeLoq a été compromis à de nombreuses reprises par des attaques cryptographiques.

Fautes :

- **DPA** : Les systèmes utilisant KeeLoq sont particulièrement vulnérables aux attaques DPA en raison de leur faible résistance aux analyses de puissance. KeeLoq effectue des opérations bit-à-bit qui montrent des variations de puissance distinctes selon les données manipulées. En enregistrant et en analysant les traces de puissance pendant l'exécution de KeeLoq, les attaquants peuvent déduire les bits de la clé.
<https://iacr.org/archive/crypto2008/51570204/51570204.pdf>
- **Bruteforce** : En raison d'une taille de clé bien trop faible (64 bits seulement), les systèmes KeeLoq sont particulièrement sensibles aux attaques par force brute.
<https://barenghi.faculty.polimi.it/lib/exe/fetch.php?media=parma2012.pdf>

```
1 class KeeLoq:  
2     def __init__(self, key: int):  
3         """  
4             Initialise une instance de la classe KeeLoq avec une clé donnée.  
5             :param key: Clé de chiffrement utilisée pour les opérations KeeLoq.  
6         """  
7         self.key = key  
8  
9     @staticmethod  
10    def __nlfsr(state):  
11        """  
12            Effectue une opération de registre de décalage linéaire rétrograde non  
13            linéaire (NLFSR) sur un état donné.  
14            :param state: L'état initial du NLFSR.  
15            :return: Le bit résultant de l'opération NLFSR.  
16        """  
17        return ((state >> 0) ^ (state >> 2) ^ (state >> 3) ^ (state >> 5) ^ (state >>  
18        7) ^ (state >> 10) ^  
19                (state >> 11) ^ (state >> 13) ^ (state >> 14) ^ (state >> 15) ^ (  
20        state >> 17) ^ (state >> 19) ^  
21                (state >> 22) ^ (state >> 24) ^ (state >> 26) ^ (state >> 28)) & 1  
22  
23    def __feistel(self, data, round_key):  
24        """  
25            Effectue une opération de 32 tour de Feistel (modifiée) sur les données.  
26            :param data: Les données à transformer.  
27            :param round_key: La clé de ronde utilisée pour l'opération Feistel.  
28        """
```

```

25     :return: Les données transformées après l'opération Feistel.
26     """
27     result = 0
28     for i in range(32):
29         lfsr_output = self._lfsr(round_key)
30         bit = ((data >> i) & 1) ^ lfsr_output
31         result |= (bit << i)
32         round_key >>= 1
33         round_key |= (bit << 63)
34     return result
35
36 def encrypt(self, data: int, counter: int):
37     """
38         Chiffre les données en utilisant un compteur et la clé de l'
39         instance KeeLoq.
40         :param data: Les données à chiffrer.
41         :param counter: Le compteur à inclure dans le chiffrement.
42         :return: Les données chiffrées.
43     """
44     data = ((counter & 0xF) << 28) | (data & 0xFFFFFFFF) # Inclure le compteur
45     # de 4 bits dans les 4 bits de poids forts
46     for round in range(528):
47         round_key = (self.key >> (round % 64)) & 0xFFFFFFFF
48         data = self._feistel(data, round_key)
49     return data
50
51 def decrypt(self, data: int):
52     """
53         Déchiffre les données en utilisant la clé de l'instance KeeLoq.
54         :param data: Les données à déchiffrer.
55         :return: Les données déchiffrées et le compteur extrait.
56     """
57     for round in range(528):
58         round_key = (self.key >> ((527 - round) % 64)) & 0xFFFFFFFF
59         data = self._feistel(data, round_key)
60         counter = (data >> 28) & 0xF # Extraire les 4 bits de poids forts comme le
compteur
61         decrypted_data = data & 0xFFFFFFFF # Extraire les 28 bits de poids faibles
comme les données
62     return decrypted_data, counter

```

Listing 1 – Implémentation de la classe KeeLoq

AES based : Les systèmes basés sur AES (Advanced Encryption Standard) utilisent une méthode de chiffrement plus robuste et moderne comparée à KeeLoq. AES est un standard de cryptographie adopté par le gouvernement des États-Unis et largement utilisé à travers le monde pour sécuriser les données.

- Taille du Bloc : 128 bits (mais on peut aussi être à 192 ou 256 bits selon la version)
- Taille de la clé : 128, 192, ou 256 bits
- Nombre de tours : 10, 12, ou 14 tours (en fonction de la taille de la clé)

Les systèmes basés sur AES utilisent une combinaison de substitutions et de permutations pour générer des codes tournants. En raison de la robustesse de l'algorithme AES, ces systèmes sont considérés comme plus sécurisés contre les attaques cryptographiques contemporaines.

L'adoption de codes tournants basés sur AES est en augmentation, car ils offrent une meilleure sécurité et résistance aux attaques par rapport aux anciens systèmes comme KeeLoq. En outre, les systèmes AES peuvent être intégrés avec d'autres techniques de sécurité, comme l'authentification mutuelle et la gestion des clés dynamique, pour renforcer encore la sécurité des systèmes de clé électronique.

Failles : Dans le cas des systèmes AES-Based la majorité des risques de sécurité actuels réside dans des défauts d'implémentation de la part des constructeurs dans leurs automobiles.

```
1  from Crypto.Cipher import AES
2
3
4  class AESRollingCode:
5      def __init__(self, key: bytes, iterations: int = 5):
6          """
7              Initialisation d'un metteur /r cepteur de code roulant
8              :param key: Cl secr te partag
9              :param iterations: Tol rance la desynchronisation
10             """
11            self.key = key
12            self.iterations = iterations
13            self.counter = 0
14            self.current_code = self.__generate_code()
15            self.future_codes = self.__generate_future_codes()
16
17    def __generate_code(self, counter: int = None):
18        """
19            G n re un code roulant partir du compteur et de la cl secr te
20            :param counter: Compteur du code g n rer
21            :return: Renvoie le nouveau code g n r
22        """
23            if counter is None:
24                counter = self.counter
25            cipher = AES.new(self.key, AES.MODE_ECB)
26            code = cipher.encrypt(counter.to_bytes(16, byteorder='big'))
27            return int.from_bytes(code, byteorder='big')
28
29    def __generate_future_codes(self):
30        """
31            G n re les futures codes roulant pour la fen tre de tol rance
32            :return: Renvoie la liste des futurs codes roulant
33        """
34            future_codes = []
35            for i in range(1, self.iterations + 1):
36                future_codes.append(self.__generate_code(self.counter + i))
37            return future_codes
38
39    def __resynchronize(self, received_code: int):
40        """
41            Resynchronise les codes roulants avec le code roulant re u
42            :param received_code: Code sur le quel se resynchroniser
43        """
44            self.current_code = received_code
45            self.counter += self.future_codes.index(received_code) + 1
46            self.future_codes = self.__generate_future_codes()
47
48    def increment_code(self):
49        """
50            Incr mente le compteur et met l'ensemble des code roulants de la classe
51            jour
52        """
53            self.counter += 1
54            self.current_code = self.__generate_code()
55            self.future_codes = self.__generate_future_codes()
56
57    def get_current_code(self):
58        """
```

```

58     :return: Renvoie le code roulant courant
59     """
60     return self.current_code
61
62 def compare_code(self, received_code: int):
63     """
64     Compare et gère la désynchronisation
65     :param received_code: Code reçu
66     :return: Renvoie si les codes sont synchronisés (True) ou non (False)
67     """
68     if received_code == self.current_code:
69         self.increment_code()
70         return True
71     elif received_code in self.future_codes:
72         self._resynchronize(received_code)
73         return True
74     return False

```

Listing 2 – Implémentation de la classe Keeloq

1.2.2 Implémentation Challenge

Hitag : Hitag est une famille de systèmes de transpondeurs utilisés principalement dans les applications d'identification par radiofréquence (RFID) et dans les systèmes de télécommande sans clé (RKE). Ces systèmes sont largement utilisés dans les systèmes d'entrée sans clé des véhicules et les systèmes d'alarme. Hitag est un système basé sur les challenges contrairement à Keeloq ce dernier est bi-directionnel.

- Initialisation :** Lorsque l'utilisateur appuie sur un bouton de la télécommande, celle-ci envoie un message initial au récepteur.
- Challenge :** Le récepteur génère un challenge, qui est une valeur aléatoire ou pseudo-aléatoire, et l'envoie à la télécommande.
- Réponse :** La télécommande utilise un algorithme de chiffrement (souvent basé sur une clé pré-partagée entre la télécommande et le récepteur) pour chiffrer le challenge et génère une réponse.
- Vérification :** Le récepteur utilise le même algorithme de chiffrement pour vérifier si la réponse de la télécommande est correcte. Si la réponse est correcte, l'authentification est réussie et le récepteur exécute la commande (par exemple, déverrouiller les portes du véhicule).

```

1 import socket
2 from Crypto.Cipher import ARC4
3 from Crypto.Random import get_random_bytes
4 import hashlib
5
6 class Hitag3Receiver:
7     def __init__(self, key):
8         """
9             Initialise le récepteur Hitag3 avec une clé de chiffrement.
10            :param key: Clé utilisée pour le chiffrement (doit être de longueur
11            arbitraire, mais sera condensée)
12            """
13            self.key = hashlib.sha256(key).digest()[:16] # Utiliser une clé de 128 bits
14
15    def generate_challenge(self):
16        """
17            Génère un challenge aléatoire.
18            :return: Challenge aléatoire

```

```

18     """
19     return get_random_bytes(8) # Utiliser un challenge de 8 octets pour imiter
Hitag3
20
21 def check_answer(self, challenge, answer):
22     """
23         V rifie si la r ponse chiffr e correspond au challenge initial en
utilisant la cl du r cepteur.
24         :param challenge: Challenge initial
25         :param answer: R ponse chiffr e v rifier
26         :return: True si la r ponse est correcte, False sinon
27     """
28     cipher = ARC4.new(self.key)
29     expected_response = cipher.encrypt(challenge)
30     return answer == expected_response
31
32 def main():
33     # G n ration d'une cl al atoire partag e
34     key = b'secret_keys'
35     receiver = Hitag3Receiver(key)
36
37     # Initialisation du serveur
38     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
39         s.bind(('localhost', 65432))
40         s.listen()
41
42     print("Serveur en attente de connexion...")
43
44     # Acceptation de la connexion d'un client
45     conn, addr = s.accept()
46     with conn:
47         print('Connect par', addr)
48
49         # G n ration du challenge
50         challenge = receiver.generate_challenge()
51         print("G n ration du challenge: ", challenge.hex())
52
53         # Envoi du challenge au client
54         conn.sendall(challenge)
55
56         # R ception de la r ponse du client
57         response = conn.recv(1024)
58         print("R ponse re ue: ", response.hex())
59
60         # V rification de la r ponse
61         if receiver.check_answer(challenge, response):
62             print("Authentication successful")
63         else:
64             print("Authentication failed")
65
66 if __name__ == "__main__":
67     main()

```

Listing 3 – Implémentation d'un RKE par challenge côté voiture

1.3 Attaques possibles

1.3.1 Bruteforce

L'attaque par force brute consiste à essayer toutes les combinaisons possibles d'un code ou d'une clé jusqu'à trouver la bonne. Dans le contexte des systèmes RKE, un attaquant pourrait tenter de deviner le code de déverrouillage en envoyant de multiples combinaisons au récepteur du véhicule. Les systèmes avec des clés de faible longueur et surtout à **code fixe** sont très sensibles à ce genre d'attaque.

En général ce genre d'attaque n'incrémente pas juste la clé jusqu'à trouver la bonne mais vont chercher à utiliser des patterns d'attaque en excluant un maximum de possibilité pour réduire au minimum le nombre de test à effectuer.

On peut aussi utiliser les **séquences de Bruijn**

"https://en.wikipedia.org/wiki/De_Bruijn_sequence?ref=secjuice.com"

1.3.2 Replay attaque

L'attaque par rejet, ou "replay attack", consiste à intercepter une transmission de données légitime entre une télécommande et un récepteur pour ensuite la rejouer ultérieurement. Dans le contexte des systèmes de clé électronique à distance (RKE - Remote Keyless Entry), un attaquant pourrait capturer le signal de la clé lorsqu'un utilisateur déverrouille son véhicule et le réutiliser pour déverrouiller le véhicule à nouveau sans autorisation.

- Exemples de CVE :
 - **CVE-2019-20626** : Honda HR-V 2017 permet une attaque par rejet.
 - **CVE-2023-33281** : Nissan Sylphy classic 2021 envoie le même signal pour chaque demande d'ouverture de porte.

1.3.3 Rolljam attaque

L'attaque RollJam exploite une faille dans le mécanisme de synchronisation des codes roulants utilisés par les systèmes de clé électronique à distance. L'attaquant utilise un dispositif qui bloque le signal de la clé lorsque l'utilisateur tente de verrouiller ou déverrouiller le véhicule, tout en enregistrant ce signal. Ensuite, il envoie un signal précédemment capturé au véhicule, déverrouillant ainsi le véhicule sans que l'utilisateur ne le sache.

- Exemples de CVE :
 - **CVE-2021-46145** : Honda CIVIC 2012 utilise des codes roulants non expirants et une resynchronisation des compteurs, facilitant les attaques RollJam.

1.3.4 Rollback attaque

L'attaque RollBack exploite des faiblesses dans la gestion des codes roulants par les systèmes RKE. L'attaquant intercepte plusieurs signaux de la clé pour ensuite rejouer un signal précédent, profitant d'une mauvaise implémentation de la gestion des codes roulants qui permet d'accepter un ancien code comme valide.

- Exemples de CVE :
 - **CVE-2022-38766** : Renault Zoe 2021 utilise le même ensemble de codes roulants pour chaque demande d'ouverture de porte.
 - **CVE-2022-37418** : Nissan, Kia, Hyundai jusqu'en 2017 permettent à des attaquants distants de réaliser des opérations de déverrouillage et de forcer une resynchronisation après avoir capturé deux signaux valides consécutifs de la clé.
 - **CVE-2022-37305** : certains véhicules Honda jusqu'en 2018 permettent à des attaquants distants de réaliser des opérations de déverrouillage et de forcer une resynchronisation après avoir capturé cinq signaux valides consécutifs de la clé RKE.

- **CVE-2022-36945** : certains véhicules Mazda jusqu'en 2020 permettent à des attaquants distants de réaliser des opérations de déverrouillage et de forcer une resynchronisation après avoir capturé trois signaux valides consécutifs de la clé.

2 Implémentation d'un algorithme d'authentification basé sur SRP

2.1 Théorie, idées

2.1.1 Secure Remote Password (SRP)

Le Secure Remote Password (SRP) est un protocole d'authentification sécurisé basé sur le principe de Diffie-Hellman, qui permet l'échange sécurisé de clés sur un canal non sécurisé. SRP est principalement utilisé dans des applications nécessitant une authentification robuste, comme les systèmes de verrouillage électronique dans les voitures.

Fonctionnement du SRP Le protocole SRP commence par un échange de clés entre le client et le serveur. Voici les étapes mathématiques impliquées :

1. Initialisation :

- Le client et le serveur choisissent un nombre premier N et une base g . Ces paramètres sont publics.
- Le client choisit un mot de passe P et calcule une valeur de vérification x à partir d'un hachage du sel s et du mot de passe : $x = H(s, P)$.
- Le serveur stocke $v = g^x N$ comme vérificateur du mot de passe.

2. Échange de valeurs publiques :

- Le client génère une valeur aléatoire secrète a et calcule $A = g^a N$. Il envoie A au serveur.
- Le serveur génère une valeur aléatoire secrète b et calcule $B = kv + g^b N$. Il envoie B au client.

3. Calcul de la clé de session :

- Les deux parties calculent un paramètre $u = H(A, B)$.
- Le client calcule la clé de session S_c comme suit :

$$S_c = (B - kg^x)^{a+ux} N$$

- Le serveur calcule la clé de session S_s comme suit :

$$S_s = (Av^u)^b N$$

- Les deux parties hachent la clé de session pour obtenir la clé finale K :

$$K = H(S)$$

Les formules mathématiques détaillées sont les suivantes :

$$\begin{aligned} x &= H(s, P) \\ v &= g^x N \\ A &= g^a N \\ B &= kv + g^b N \\ u &= H(A, B) \\ S_c &= (B - kg^x)^{a+ux} N \\ S_s &= (Av^u)^b N \\ K &= H(S) \end{aligned}$$

Ce protocole garantit que même si un attaquant intercepte les valeurs A et B , il ne pourra pas calculer la clé de session sans connaître les valeurs secrètes a et b .

Cas d'utilisation Le SRP est utilisé dans des systèmes nécessitant une authentification sécurisée, comme les systèmes de verrouillage électronique des voitures. Cela permet de garantir que seul un utilisateur authentifié peut déverrouiller le véhicule.

Implémentation L'implémentation de SRP passe par le réseau, avec un échange entre un client et un serveur. Dans les codes ci-dessous, des commentaires explicatifs. Pour des raisons de clarté, nous avons utilisé la librairie python existante ‘srp’. Néanmoins sur le github du projet, vous pourrez trouver une implémentation "fait maison" du protocole SRP :

```
1 import srp
2 import socket
3 import json
4 import base64
5
6
7 class AuthenticationFailed(Exception):
8     """Exception levée en cas d'échec de l'authentification."""
9     pass
10
11 def start_user_authentication(username, password):
12     """Démarrer le processus d'authentification pour l'utilisateur."""
13     print("Starting user authentication...")
14     usr = srp.User(username, password)
15     uname, A = usr.start_authentication()
16     A_encoded = base64.b64encode(A).decode('utf-8')
17     print(f"User authentication started. Username: {uname}, A: {A_encoded}\n")
18     return usr, uname, A_encoded
19
20 def process_server_challenge(usr, s_encoded, B_encoded):
21     """Traite le défi envoyé par le serveur."""
22     s = base64.b64decode(s_encoded)
23     B = base64.b64decode(B_encoded)
24     print("Processing server challenge...")
25     M = usr.process_challenge(s, B)
26     M_encoded = base64.b64encode(M).decode('utf-8')
27     print(f"Challenge processed. M: {M_encoded}\n")
28     return M_encoded
29
30 def verify_session_on_client(usr, hamk_encoded):
31     """Vérifie la session sur le client."""
32     hamk = base64.b64decode(hamk_encoded)
33     print("Verifying session on client...")
34     usr.verify_session(hamk)
35     print("Session verified on client.\n")
36
37 def send_to_server(conn, data):
38     """Envoie les données au serveur."""
39     conn.sendall(json.dumps(data).encode('utf-8'))
40
41 def receive_from_server(conn):
42     """Reçoit les données du serveur."""
43     data = conn.recv(1024)
44     return json.loads(data.decode('utf-8'))
45
46 def main():
47     # Informations utilisateur pour l'exemple
48     username = 'testuser'
49     password = 'testpassword'
50
51     # But de l'authentification utilisateur
```

```

52     usr, uname, A_encoded = start_user_authentication(username, password)
53
54     # Initialisation de la connexion au serveur
55     conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
56     conn.connect(('localhost', 8080))
57
58     try:
59         # Envoi du nom d'utilisateur et de A au serveur
60         send_to_server(conn, {'username': uname, 'A': A_encoded})
61
62         # Reception du dfi du serveur
63         server_data = receive_from_server(conn)
64         s_encoded = server_data['s']
65         B_encoded = server_data['B']
66
67         # Si le serveur choue cr er le challenge, l'authentification choue
68         if s_encoded is None or B_encoded is None:
69             raise AuthenticationFailed()
70
71         # Le client traite le challenge du serveur
72         M_encoded = process_server_challenge(usr, s_encoded, B_encoded)
73
74         # Si le client choue traiter le challenge, l'authentification choue
75         if M_encoded is None:
76             raise AuthenticationFailed()
77
78         # Envoi de M au serveur pour v rification de la session
79         send_to_server(conn, {'M': M_encoded})
80
81         # Reception de la v rification finale du serveur
82         server_data = receive_from_server(conn)
83         HAMK_encoded = server_data['HAMK']
84
85         # Si le serveur choue v rifier la session, l'authentification choue
86         if HAMK_encoded is None:
87             raise AuthenticationFailed()
88
89         # V rification finale de la session sur le client
90         verify_session_on_client(usr, HAMK_encoded)
91
92         # V rification que le client est authentifi
93         print("Authentication process completed.")
94         if usr.authenticated():
95             print("Client is authenticated.")
96         else:
97             raise AuthenticationFailed()
98     finally:
99         conn.close()
100
101 if __name__ == '__main__':
102     main()

```

Listing 4 – Client SRP (Par exemple la clé de la voiture)

```

1
2 import srp
3 import socket
4 import json
5 import base64
6
7 class AuthenticationFailed(Exception):

```

```

8     """Exception lev e en cas d' chec de l'authentification."""
9     pass
10
11 def create_salted_verification_key(username, password):
12     """Cr e une cl de v rification sal e pour un utilisateur donn ."""
13     print("Creating salted verification key...")
14     salt, vkey = srp.create_salted_verification_key(username, password)
15     salt_encoded = base64.b64encode(salt).decode('utf-8')
16     print("Salt and verification key created.\n")
17     return salt_encoded, vkey
18
19 def create_server_verifier(username, salt_encoded, vkey, A_encoded):
20     """Cr e un v rificateur de serveur pour l'utilisateur."""
21     salt = base64.b64decode(salt_encoded)
22     A = base64.b64decode(A_encoded)
23     print("Creating server verifier...")
24     svr = srp.Verifier(username, salt, vkey, A)
25     s, B = svr.get_challenge()
26     s_encoded = base64.b64encode(s).decode('utf-8')
27     B_encoded = base64.b64encode(B).decode('utf-8')
28     print(f"Server verifier created. Salt: {s_encoded}, B: {B_encoded}\n")
29     return svr, s_encoded, B_encoded
30
31 def verify_session_on_server(svr, M_encoded):
32     """V rifie la session sur le serveur."""
33     M = base64.b64decode(M_encoded)
34     print("Verifying session on server...")
35     HAMK = svr.verify_session(M)
36     HAMK_encoded = base64.b64encode(HAMK).decode('utf-8')
37     print(f"Session verified on server. HAMK: {HAMK_encoded}\n")
38     return HAMK_encoded
39
40 def send_to_client(data, conn):
41     """Envoie les donn es au client."""
42     conn.sendall(json.dumps(data).encode('utf-8'))
43
44 def receive_from_client(conn):
45     """Re oit les donn es du client."""
46     data = conn.recv(1024)
47     return json.loads(data.decode('utf-8'))
48
49 def main():
50     # Informations utilisateur pour l'exemple (c t le serveur, ces infos devraient
51     # tre dans une base de donn es)
52     username = 'testuser'
53     password = 'testpassword'
54
55     # Cr ation de la cl de v rification avec salage (simul e ici)
56     salt_encoded, vkey = create_salted_verification_key(username, password)
57
58     # Cr ation du socket serveur
59     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
60         s.bind(('0.0.0.0', 8080))
61         s.listen()
62         print("Server is listening on port 8080...")
63         conn, addr = s.accept()
64         with conn:
65             print(f"Connected by {addr}")
66
67             # R ception des donn es de l'utilisateur

```

```

67     client_data = receive_from_client(conn)
68     uname = client_data['username']
69     A_encoded = client_data['A']
70
71     # Cr ation du v rificateur du serveur
72     svr, s_encoded, B_encoded = create_server_verifier(uname, salt_encoded,
73     vkey, A_encoded)
74
75     # Si le serveur choue      cr er le challenge, l'authentification
76     choue
77     if s_encoded is None or B_encoded is None:
78         raise AuthenticationFailed()
79
80     print("ENVOI DE S ET B")
81     # Envoi du d fi au client
82     send_to_client({'s': s_encoded, 'B': B_encoded}, conn)
83
84     # R ception de M du client pour v rification de la session
85     client_data = receive_from_client(conn)
86     M_encoded = client_data['M']
87
88     # Si le serveur choue      v rifier la session, l'authentification
89     choue
90     if M_encoded is None:
91         raise AuthenticationFailed()
92
93     # V rification de la session sur le serveur
94     HAMK_encoded = verify_session_on_server(svr, M_encoded)
95
96     # V rification que le serveur est authentifi
97     print("Authentication process completed.")
98     if svr.authenticated():
99         print("Server is authenticated.")
100    else:
101        raise AuthenticationFailed()
102
103 if __name__ == '__main__':
104     main()

```

Listing 5 – Serveur SRP (Par exemple la voiture/système CAN gérant la centralisation)

Illustration de l'échange SRP Voici une illustration du déroulement de l'échange SRP :

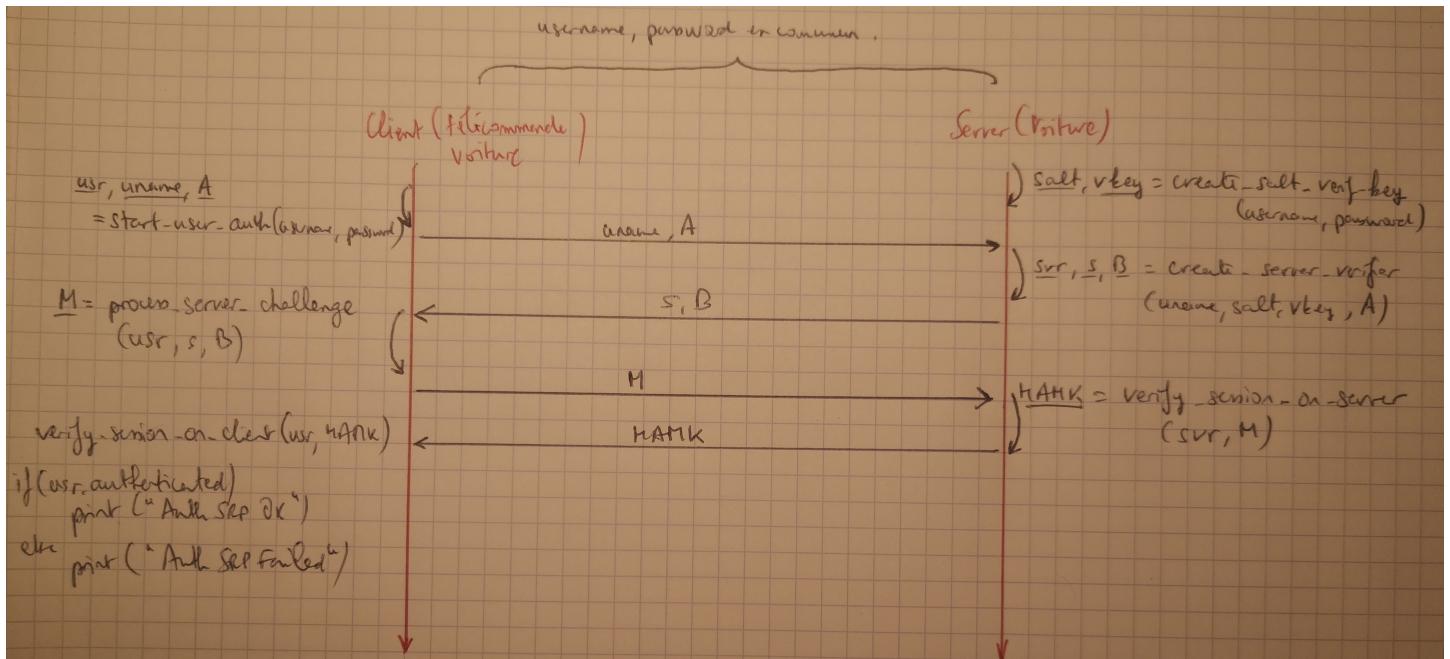


FIGURE 4 – Déroulement de l'échange SRP

Problématique La problématique à laquelle nous devons répondre est le développement d'une méthode d'authentification pour SRP afin d'éviter les attaques de type Man-in-the-Middle (MITM). Une attaque MITM se produit lorsque l'attaquant intercepte et éventuellement altère la communication entre le client et le serveur à leur insu. Dans le cas du SRP, le protocole assure bien la confidentialité des données entre deux parties, mais le protocole n'assure pas l'authentification. C'est cette partie qui va nous intéresser dans la suite.

2.1.2 Choix des courbes elliptiques comme moyen d'authentification

Définition des courbes elliptiques Les courbes elliptiques sont définies par l'équation de Weierstrass sous la forme :

$$y^2 = x^3 + ax + b$$

où a et b sont des constantes qui déterminent la forme de la courbe. Les courbes elliptiques sur un corps fini offrent des propriétés mathématiques utiles pour la cryptographie, notamment la difficulté de résoudre le problème du logarithme discret. Nous verrons dans la suite du rapport (partie code/implémentation) quelle est la courbe que nous choisirons ainsi que ses paramètres.

Protocole ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) L'ECDHE est un protocole de clé d'échange basé sur les courbes elliptiques. Il permet deux parties de générer une clé de session partagée de manière sécurisée sur un canal non sécurisé. Les étapes du processus ECDHE sont les suivantes :

1. Initialisation :

- Lors de l'inscription, l'utilisateur et le serveur effectuent les étapes suivantes :

2. Utilisateur :

- Choisit un mot de passe P .
- Calcule un sel aléatoire s .
- Calcule un vérificateur v :

$$v = g^{H(s,P)} N$$

où g est une base (générateur), N est un grand nombre premier, et H est une fonction de hachage cryptographique.

- **Serveur** : - Stocke le sel s et le vérificateur v dans la base de données.

2. Phase d'authentification :

- **Initialisation de l'authentification** :

- **Utilisateur** :

- Génère une clé privée éphémère a et une clé publique

$$A = g^a N$$

- **Serveur** :

- Génère une clé privée éphémère b et une clé publique

$$B = kv + g^b N$$

- Échange :

- L'utilisateur envoie A au serveur.
- Le serveur envoie B et s à l'utilisateur.

3. Utilisation de l'ECDHE pour la clé de session :

- Chaque partie génère une paire de clés supplémentaire pour l'échange éphémère :

- **Utilisateur** :

- Génère une paire de clés ECDHE (d_U, Q_U) , où d_U est la clé privée et $Q_U = d_U \cdot G$ est la clé publique, avec G étant le générateur de la courbe elliptique.

- Envoie Q_U au serveur.

- **Serveur** :

- Génère une paire de clés ECDHE (d_S, Q_S) , où d_S est la clé privée et $Q_S = d_S \cdot G$ est la clé publique.
- Envoie Q_S à l'utilisateur.

4. Calcul des clés de session avec ECDHE :

- **Utilisateur** :

- Calcule la clé partagée

$$Z_U = d_U \cdot Q_S$$

- Calcule

$$u = H(A, B)$$

- Calcule la clé de session

$$S_U = (B - kv)^{(a+ux)} \cdot Z_U N$$

où

$$x = H(s, P)$$

- **Serveur** :

- Calcule la clé partagée

$$Z_S = d_S \cdot Q_U$$

- Calcule

$$u = H(A, B)$$

- Calcule la clé de session

$$S_S = (A \cdot v^u)^b \cdot Z_S N$$

5. Génération de la clé de session :

- **Utilisateur et serveur :**

- Dérivent la clé de session K en appliquant une fonction de hachage à la clé de session calculée S .

6. Vérification mutuelle :

- Pour s'assurer que les deux parties possèdent la même clé de session K , elles échangent des preuves de connaissance.

- **Utilisateur :**

- Calcule

$$M_U = H(H(N) \oplus H(g) \parallel H(I) \parallel s \parallel A \parallel B \parallel K)$$

où I est l'identité de l'utilisateur. - Envoie M_U au serveur.

- **Serveur :**

- Calcule M_U de manière indépendante et le compare avec celui reçu de l'utilisateur.
- Si les valeurs correspondent, le serveur envoie

$$M_S = H(A \parallel M_U \parallel K)$$

à l'utilisateur.

- **Utilisateur :**

- Calcule M_S de manière indépendante et le compare avec celui reçu du serveur.

Avantages de l'utilisation de l'ECDHE dans SRP - **Secret parfait (forward secrecy)** : Les clés éphémères utilisées dans ECDHE assurent que même si les clés privées de longue durée sont compromises, les sessions passées restent sécurisées. - **Efficacité cryptographique** : Les courbes elliptiques offrent une sécurité équivalente avec des tailles de clé plus petites, ce qui réduit le temps de calcul et l'utilisation de la bande passante. - **Sécurité accrue** : L'utilisation d'ECDHE ajoute une couche supplémentaire de sécurité, compliquant davantage les tentatives d'attaque.

Ainsi, le protocole SRP avec ECDHE combine les avantages de l'authentification par mot de passe sécurisée de SRP avec les bénéfices de la cryptographie moderne des courbes elliptiques, offrant une solution robuste et efficace pour les connexions sécurisées.

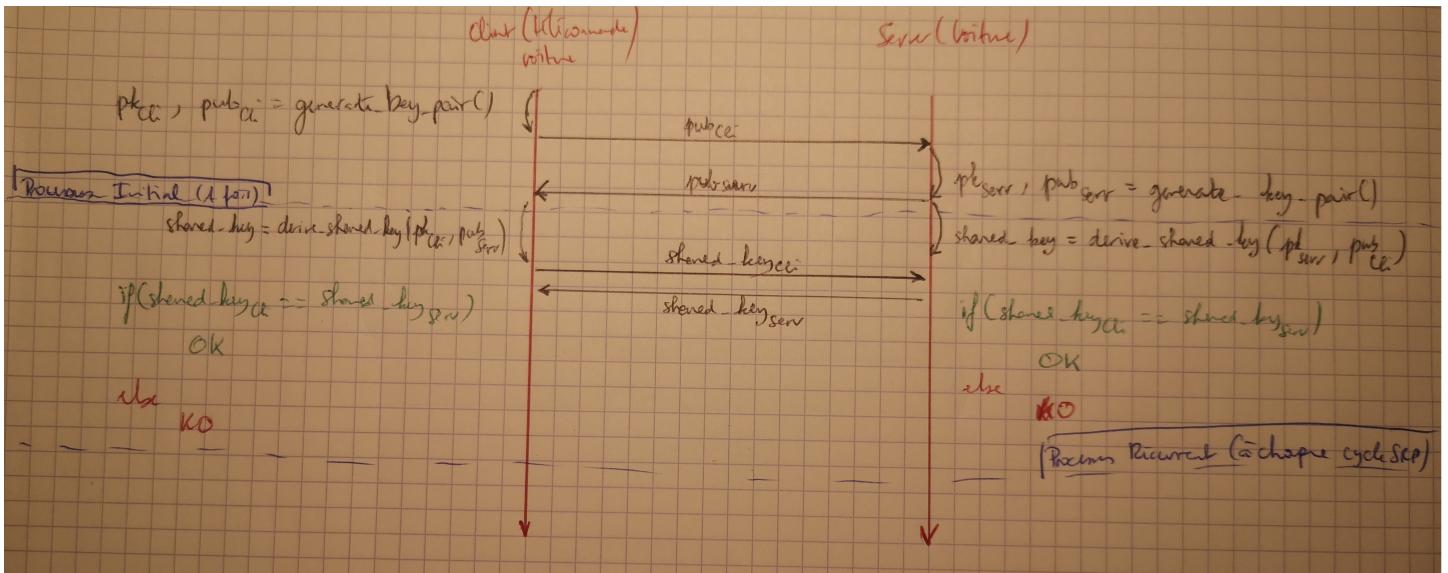


FIGURE 5 – Échanges préliminaires ECDHE

Schéma de l'échange La partie délimitée haute du schéma correspond à la génération des clés, qui doit être effectuée une fois au préalable des communications SRP. La partie délimitée en bas est à exécuter pour chaque cycle SRP.

2.2 Implémentation sous python d'une solution possible

Utilisation des courbes elliptiques SECP256R1 Les valeurs spécifiques pour la courbe elliptique SECP256R1 (alias P-256) sont :

- Prime p :

- ### — Coefficient a :

- ### — Coefficient b :

`b = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b`

- Point de base G (le générateur) :

$G_x = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296$

$G_y = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececcb6406837bf51f5$

Donc, G est le point (G_x, G_y) .

- ## — Ordre n :

$n = 0x f f f f f f f f f f 0 0 0 0 0 0 0 0 f f f f f f f f f f f f f f f f b c e 6 f a a d a 7 1 7 9 e 8 4 f 3 b 9 c a c 2 f c 6 3 2 5 5 1$

- Cofacteur h :

$$h = 1$$

Ces valeurs définissent complètement la courbe elliptique SECP256R1. La clé publique est dérivée de la clé privée par la multiplication scalaire du point de base G par la clé privée d (un entier aléatoire entre 1 et $n - 1$).

Clé privée d : Un entier aléatoire choisi dans l'intervalle $(1, n - 1)$.

Clé publique Q : Calculée comme $Q = d \cdot G$, où G est le générateur de la courbe elliptique et d est la clé privée.

Génération de la clé privée et publique La clé privée d est un entier aléatoire, et la clé publique Q est le point obtenu par la multiplication scalaire de G par d :

$$Q = d \cdot G$$

Dérivation de la clé partagée Pour deux parties, disons Alice et Bob :

- Alice a une clé privée d_A et une clé publique $Q_A = d_A \cdot G$.
- Bob a une clé privée d_B et une clé publique $Q_B = d_B \cdot G$.

La clé partagée est dérivée en utilisant la clé privée de l'une des parties et la clé publique de l'autre :

$$Cl_{partage} = d_A \cdot Q_B = d_A \cdot (d_B \cdot G) = d_B \cdot (d_A \cdot G) = d_B \cdot Q_A$$

La clé partagée est le même point sur la courbe elliptique, que ce soit calculé par Alice ou par Bob.

L'entièreté des codes sont disponibles sur le [repo Github](#) :

```

1 import srp
2 import socket
3 import json
4 import base64
5
6 from cryptography.hazmat.primitives import ec
7 from cryptography.hazmat.primitives import serialization, hashes
8 from cryptography.hazmat.primitives.kdf.hkdf import HKDF
9
10 class AuthenticationFailed(Exception):
11     """Exception lev e en cas d' chec de l'authentification."""
12     pass
13
14 # Utilitaires pour ECDHE
15 def generate_key_pair():
16     """
17         G n re une paire de cl s (priv e et publique) ECDHE.
18
19     Courbe elliptique utilise : SECP256R1 (alias P-256)
20     - Cette courbe est d finie par l' quation :  $y^2 = x^3 + ax + b \text{ mod } p$ 
21     - Param tres de la courbe SECP256R1 :
22         - p (le module) : un grand nombre premier
23         - a et b : coefficients de l' quation de la courbe
24         - G (le g n rateur) : un point de base ( $x_G, y_G$ ) sur la courbe
25         - n : l'ordre de G (le nombre de points sur la courbe)
26
27     La cl priv e est un entier al atoire d ( $0 < d < n$ ). La cl publique est un
28     point  $Q = d * G$ ,
29     o la multiplication est la multiplication scalaire sur la courbe elliptique.
30     """
31     private_key = ec.generate_private_key(ec.SECP256R1()) # d (cl priv e)
32     public_key = private_key.public_key() # Q = d * G (cl publique)
33     return private_key, public_key
34
35 def serialize_public_key(public_key):
36     """
37         S rialise une cl publique en format PEM.
38
39         La cl publique est un point sur la courbe elliptique, souvent repr sent en
40         coordonn es (x, y).
41         La s rialisation convertit ce point en un format standardis pour change .
42     """
43     return public_key.public_bytes(
44         encoding=serialization.Encoding.PEM,

```

```

43     format=serialization.PublicFormat.SubjectPublicKeyInfo
44 )
45
46 def deserialize_public_key(pem_data):
47 """
48     Déserialise une clé publique à partir du format PEM.
49
50     La déserialisation convertit les données PEM en un point (x, y) sur la courbe
51     elliptique.
52 """
53     return serialization.load_pem_public_key(pem_data)
54
55 def derive_shared_key(private_key, peer_public_key):
56 """
57     Crée une clé partagée à partir d'une clé privée et d'une clé publique d'un
58     paire.
59
60     Mathématiquement, si le client possède une clé privée  $d_C$  et le serveur une
61     clé privée  $d_S$ , et que les clés publiques
62     correspondantes sont  $Q_C = d_C * G$  et  $Q_S = d_S * G$  ( $G$  est un générateur
63     sur la courbe), alors la clé partagée
64     est  $S = d_C * Q_S = d_S * Q_C$ , qui est un point sur la courbe elliptique. Le KDF
65     (HKDF ici) est utilisé pour
66     créer une clé symétrique à partir de ce point partagé.
67 """
68     shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
69     return HKDF(
70         algorithm=hashes.SHA256(),
71         length=32,
72         salt=None,
73         info=b'handshake data',
74     ).derive(shared_key)
75
76 def start_user_authentication(username, password):
77     usr = srp.User(username, password)
78     uname, A = usr.start_authentication()
79     A_encoded = base64.b64encode(A).decode('utf-8')
80     return usr, uname, A_encoded
81
82 def process_server_challenge(usr, s_encoded, B_encoded):
83     s = base64.b64decode(s_encoded)
84     B = base64.b64decode(B_encoded)
85     M = usr.process_challenge(s, B)
86     M_encoded = base64.b64encode(M).decode('utf-8')
87     return M_encoded
88
89 def verify_session_on_client(usr, hamk_encoded):
90     hamk = base64.b64decode(hamk_encoded)
91     usr.verify_session(hamk)
92
93 def send_to_server(conn, data):
94     conn.sendall(json.dumps(data).encode('utf-8'))
95
96 def receive_from_server(conn):
97     data = conn.recv(4096)
98     return json.loads(data.decode('utf-8'))
99
100 def main():
101     username = 'testuser'
102     password = 'testpassword'

```

```

98     conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
99     conn.connect(('169.254.36.137', 8080))
100
101    try:
102        # Phase ECDHE
103        print("Generating client key pair...")
104        client_private_key, client_public_key = generate_key_pair()
105        serialized_client_public_key = serialize_public_key(client_public_key)
106
107        print("Sending client public key to server...")
108        send_to_server(conn, {'client_public_key': serialized_client_public_key.
109 decode('utf-8')})
110
111        print("Receiving server public key...")
112        server_data = receive_from_server(conn)
113        server_public_key_pem = server_data['server_public_key']
114        peer_server_public_key = deserialize_public_key(server_public_key_pem.encode(
115 'utf-8'))
116
116        print("Deriving shared key on client...")
117        shared_key_client = derive_shared_key(client_private_key,
118        peer_server_public_key)
119        serialized_shared_key_client = base64.b64encode(shared_key_client).decode(
119 'utf-8')
120
120        print("Sending client shared key to server for verification...")
121        send_to_server(conn, {'shared_key_client': serialized_shared_key_client})
122
123        print("Receiving server shared key for verification...")
124        server_data = receive_from_server(conn)
125        serialized_shared_key_server = server_data['shared_key_server']
126
127        assert serialized_shared_key_client == serialized_shared_key_server, "Shared
127 keys do not match!"
128        print("Shared keys match. ECDHE verification completed.")
129
130        # Phase SRP
131        print("Starting user authentication...")
132        usr, uname, A_encoded = start_user_authentication(username, password)
133        print(f"User authentication started. Username: {uname}, A: {A_encoded}")
134
135        print("Sending username and A to server...")
136        send_to_server(conn, {'username': uname, 'A': A_encoded})
137
138        print("Receiving challenge from server...")
139        server_data = receive_from_server(conn)
140        s_encoded = server_data['s']
141        B_encoded = server_data['B']
142
143        if s_encoded is None or B_encoded is None:
144            raise AuthenticationFailed()
145
146        print("Processing server challenge...")
147        M_encoded = process_server_challenge(usr, s_encoded, B_encoded)
148        print(f"Challenge processed. M: {M_encoded}")
149
150        if M_encoded is None:
151            raise AuthenticationFailed()
152

```

```

153     print("Sending M to server...")
154     send_to_server(conn, {'M': M_encoded})
155
156     print("Receiving HAMK from server...")
157     server_data = receive_from_server(conn)
158     HAMK_encoded = server_data['HAMK']
159
160     if HAMK_encoded is None:
161         raise AuthenticationFailed()
162
163     print("Verifying session on client...")
164     verify_session_on_client(usr, HAMK_encoded)
165     print("Session verified on client.")
166
167     print("Authentication process completed.")
168     if usr.authenticated():
169         print("Client is authenticated.")
170     else:
171         raise AuthenticationFailed()
172 finally:
173     conn.close()
174
175 if __name__ == '__main__':
176     main()

```

Listing 6 – Client SRP avec ECDHE

```

1
2
3 import srp
4 import socket
5 import json
6 import base64
7 from cryptography.hazmat.primitives.asymmetric import ec
8 from cryptography.hazmat.primitives import serialization, hashes
9 from cryptography.hazmat.primitives.kdf.hkdf import HKDF
10
11 class AuthenticationFailed(Exception):
12     """Exception lev e en cas d' chec de l'authentification."""
13     pass
14
15 # Utilitaires pour ECDHE
16 def generate_key_pair():
17     """
18     G n re une paire de cl s (priv e et publique) ECDHE.
19
20     Courbe elliptique utilis e : SECP256R1 (alias P-256)
21     - Cette courbe est d finie par l' quation :  $y^2 = x^3 + ax + b \text{ mod } p$ 
22     - Param tres de la courbe SECP256R1 :
23         - p (le module) : un grand nombre premier
24         - a et b : coefficients de l' quation de la courbe
25         - G (le g n rateur) : un point de base ( $x_G, y_G$ ) sur la courbe
26         - n : l'ordre de G (le nombre de points sur la courbe)
27
28     La cl priv e est un entier alatoire d ( $0 < d < n$ ). La cl publique est un
29     point  $Q = d * G$ ,
30     o la multiplication est la multiplication scalaire sur la courbe elliptique.
31     """
32
33     private_key = ec.generate_private_key(ec.SECP256R1()) # d (cl priv e)
34     public_key = private_key.public_key() # Q = d * G (cl publique)
35     return private_key, public_key

```

```

34
35 def serialize_public_key(public_key):
36     """
37         S rialise une cl  publique en format PEM.
38
39         La cl  publique est un point sur la courbe elliptique, souvent repr sent en
40         coordonn es (x, y).
41         La s rialisation convertit ce point en un format standardis pour change .
42         """
43
44     return public_key.public_bytes(
45         encoding=serialization.Encoding.PEM,
46         format=serialization.PublicFormat.SubjectPublicKeyInfo
47     )
48
49 def deserialize_public_key(pem_data):
50     """
51         D s rialise une cl  publique partir du format PEM.
52
53         La d s rialisation convertit les donn es PEM en un point (x, y) sur la courbe
54         elliptique.
55         """
56
57     return serialization.load_pem_public_key(pem_data)
58
59 def derive_shared_key(private_key, peer_public_key):
60     """
61         D rive une cl  partag e partir d'une cl  priv e et d'une cl  publique d'
62         un pair.
63
64         Math matiquement, si le client poss de une cl  priv e  $d_C$  et le serveur une
65         cl  priv e  $d_S$ , et que les cl s publiques
66         correspondantes sont  $Q_C = d_C * G$  et  $Q_S = d_S * G$  ( $G$  est un g n rateur
67         sur la courbe), alors la cl  partag e
68         est  $S = d_C * Q_S = d_S * Q_C$ , qui est un point sur la courbe elliptique. Le KDF
69         (HKDF ici) est utilis pour
70         d river une cl  sym trique partir de ce point partag .
71         """
72
73     shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
74     return HKDF(
75         algorithm=hashes.SHA256(),
76         length=32,
77         salt=None,
78         info=b'handshake data',
79     ).derive(shared_key)
80
81 def create_salted_verification_key(username, password):
82     salt, vkey = srp.create_salted_verification_key(username, password)
83     salt_encoded = base64.b64encode(salt).decode('utf-8')
84     return salt_encoded, vkey
85
86 def create_server_verifier(username, salt_encoded, vkey, A_encoded):
87     salt = base64.b64decode(salt_encoded)
88     A = base64.b64decode(A_encoded)
89     svr = srp.Verifier(username, salt, vkey, A)
90     s, B = svr.get_challenge()
91     s_encoded = base64.b64encode(s).decode('utf-8')
92     B_encoded = base64.b64encode(B).decode('utf-8')
93     return svr, s_encoded, B_encoded
94
95 def verify_session_on_server(svr, M_encoded):
96     M = base64.b64decode(M_encoded)

```

```

88     HAMK = svr.verify_session(M)
89     HAMK_encoded = base64.b64encode(HAMK).decode('utf-8')
90     return HAMK_encoded
91
92 def send_to_client(conn, data):
93     conn.sendall(json.dumps(data).encode('utf-8'))
94
95 def receive_from_client(conn):
96     data = conn.recv(4096)
97     return json.loads(data.decode('utf-8'))
98
99 def main():
100     username = 'testuser'
101     password = 'testpassword'
102
103     #print("Creating salted verification key...")
104     salt_encoded, vkey = create_salted_verification_key(username, password)
105     #print("Salt and verification key created.")
106
107     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
108         s.bind(('0.0.0.0', 8080))
109         s.listen()
110         #print("Server is listening on port 8080...")
111         conn, addr = s.accept()
112         with conn:
113             #print(f"Connected by {addr}")
114
115             # Phase ECDHE
116             #print("Receiving client public key...")
117             client_data = receive_from_client(conn)
118             client_public_key_pem = client_data['client_public_key']
119             peer_client_public_key = deserialize_public_key(client_public_key_pem.
encode('utf-8'))
120
121             #print("Generating server key pair...")
122             server_private_key, server_public_key = generate_key_pair()
123             serialized_server_public_key = serialize_public_key(server_public_key)
124
125             #print("Deriving shared key on server...")
126             shared_key_server = derive_shared_key(server_private_key,
peer_client_public_key)
127             serialized_shared_key_server = base64.b64encode(shared_key_server).decode
('utf-8')
128
129             #print("Sending server public key to client...")
130             send_to_client(conn, {'server_public_key': serialized_server_public_key.
decode('utf-8')})
131
132             #print("Receiving client shared key for verification...")
133             client_data = receive_from_client(conn)
134             serialized_shared_key_client = client_data['shared_key_client']
135
136             #print("Sending server shared key to client for verification...")
137             send_to_client(conn, {'shared_key_server': serialized_shared_key_server})
138
139             assert serialized_shared_key_client == serialized_shared_key_server, "Shared keys do not match!"
140             #print("Shared keys match. ECDHE verification completed.")
141
142             # Phase SRP

```

```

143     #print("Receiving data from client...")
144     client_data = receive_from_client(conn)
145     uname = client_data['username']
146     A_encoded = client_data['A']
147
148     #print("Creating server verifier...")
149     svr, s_encoded, B_encoded = create_server_verifier(uname, salt_encoded,
150     vkey, A_encoded)
151     #print(f"Server verifier created. Salt: {s_encoded}, B: {B_encoded}")
152
153     if s_encoded is None or B_encoded is None:
154         raise AuthenticationFailed()
155
156     #print("Sending challenge to client...")
157     send_to_client(conn, {'s': s_encoded, 'B': B_encoded})
158
159     #print("Receiving M from client...")
160     client_data = receive_from_client(conn)
161     M_encoded = client_data['M']
162
163     if M_encoded is None:
164         raise AuthenticationFailed()
165
166     #print("Verifying session on server...")
167     HAMK_encoded = verify_session_on_server(svr, M_encoded)
168     #print(f"Session verified on server. HAMK: {HAMK_encoded}")
169
170     #print("Sending HAMK to client...")
171     send_to_client(conn, {'HAMK': HAMK_encoded})
172
173     #print("Authentication process completed.")
174     if svr.authenticated():
175         #print("Server is authenticated.")
176         pass
177     else:
178         raise AuthenticationFailed()
179 if __name__ == '__main__':
180     main()

```

Listing 7 – Serveur SRP avec ECDHE

3 Démonstration

3.1 Idée de base

L'idée de base pour la démonstration était d'utiliser des émetteurs et récepteurs RF 433 MHz classiques avec deux Raspberry Pi (RPI) pour établir la communication entre un client et un serveur. Chaque RPI serait équipé de un émetteur et un récepteur en raison de la nature unidirectionnelle de ces modules RF. Malheureusement, cette approche n'a pas abouti en raison de plusieurs limitations.

Composants utilisés

- **Raspberry Pi (RPI)** : Un mini-ordinateur à faible coût et haute performance.
- **Émetteur RF 433 MHz** : Module utilisé pour envoyer des signaux RF.
- **Récepteur RF 433 MHz** : Module utilisé pour recevoir des signaux RF.

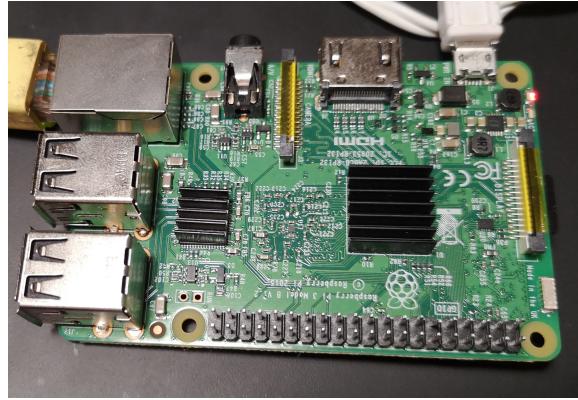


FIGURE 6 – Raspberry Pi (RPI)



FIGURE 7 – Émetteur RF 433 MHz

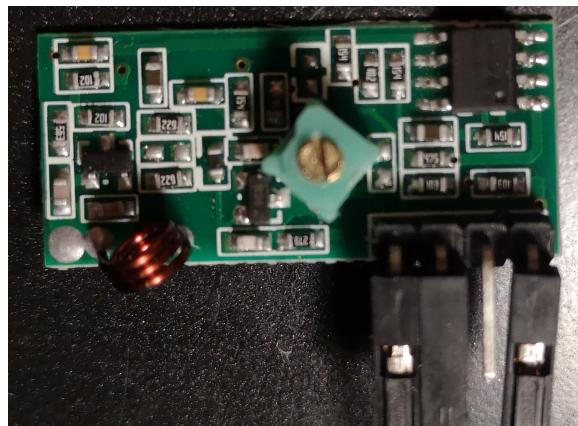


FIGURE 8 – Récepteur RF 433 MHz

Images des composants

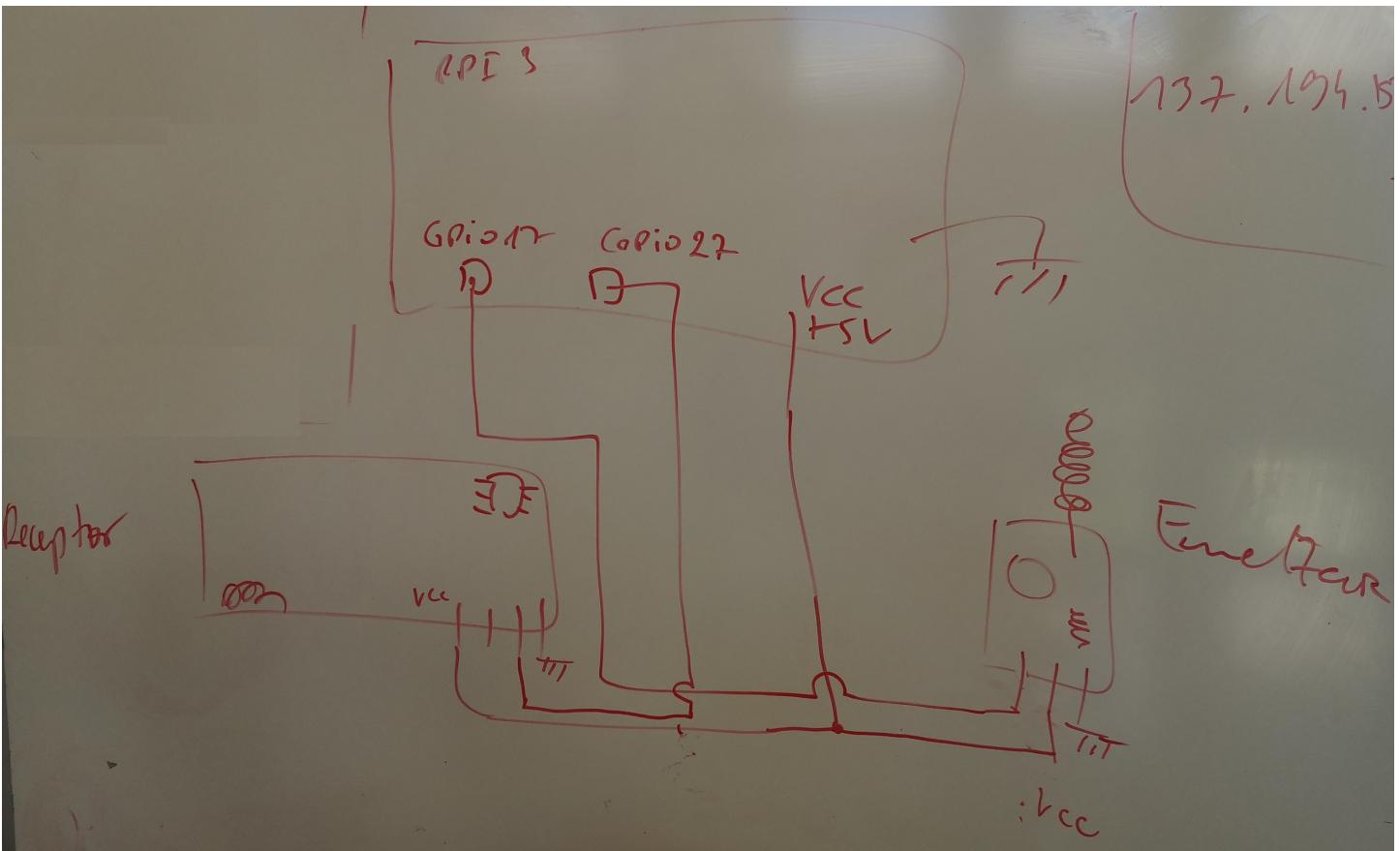


FIGURE 9 – Schéma de branchement des émetteurs et récepteurs RF sur les Raspberry Pi

Schéma de branchement

Description du schéma de branchement Le schéma de branchement illustre la configuration des émetteurs et récepteurs RF 433 MHz sur les Raspberry Pi. Chaque RPI est connecté à un émetteur et un récepteur pour la communication unidirectionnelle. Les connexions des modules RF aux broches GPIO des RPI sont indiquées.

Limitations de cette approche L'utilisation de modules RF 433 MHz pour cette démonstration a présenté plusieurs limitations :

- **Unidirectionnalité** : Chaque module RF est soit émetteur, soit récepteur, ce qui nécessite des modules doubles pour une communication bidirectionnelle.
- **Lenteur** : Les modules RF 433 MHz sont conçus pour transmettre des entiers, ce qui les rend lents pour des applications nécessitant des échanges de données plus complexes.
- **Fiabilité** : La communication RF peut être sujette à des interférences et des pertes de données, compromettant la fiabilité de la démonstration.

3.2 Démonstration PC/RPI via câble Ethernet direct

Idée de base Pour cette démonstration, nous avons utilisé un PC et un Raspberry Pi (RPI) connectés directement via un câble Ethernet. Cette configuration permet une communication bidirectionnelle fiable et rapide entre les deux dispositifs. Le RPI peut jouer alternativement le rôle de client et de serveur.

Captures d'écran des scripts client/serveur Voici les captures d'écran montrant le verbose complet des deux scripts client et serveur.

```

mufle@central-fr-2:~/rks $ python client.py
Generating client key pair...
Sending client public key to server...
Receiving server public key...
Deriving shared key on client...
Sending client shared key to server for verification...
Receiving server shared key for verification...
Shared keys match. ECDHE verification completed.
Starting user authentication...
User authentication started. Username: testuser, A: kLEJHJsktDToi8MFNCEl00foCxTNJugjX/C/juMHh19Jcwlb9RDXpWHEdm7q0W6galSG
Vhl3+977l2f3Efj3qeQvxSc5dmkh1sFhNjVUNTYn90Y2IFo8b5aInDvm0tCN2AnnbDbttFk8csZck3wld+896Mt5VGdjntVsR2rH2tN1N0Qy6B6FUJnZDh0
1k+NRx74G4XtrZUSFUigGdyS1Tw3PuDKbF93F+ibP7NO4bkBV0enu3AGbIbpDkqJgPffFaRwpMc7zbAXiP7Tm6heZbitS5+8EuEl73wNUcOF5wa1hqGQ/xZJ
nvEpEF9Y4aeReevkPXwk+rStoBYG/PaStA==
Sending username and A to server...
Receiving challenge from server...
Processing server challenge...
Challenge processed. M: 7TH2vY0HyjEkYUse+qnzSGf7cog=
Sending M to server...
Receiving HAMK from server...
Verifying session on client...
Session verified on client.
Authentiaction process completed.
Client is authenticated.

```

FIGURE 10 – Verbose du script client sur le RPI

```

PS C:\Users\SetOff244\Desktop\telecomparis\2eme Annee\ProjetIntegrateur\implementation\SRP_ECDHE_bi> python .\server.py
Creating salted verification key...
Salt and verification key created.
Server is listening on port 8080...
Connected by ('169.254.36.137', 57912)
Receiving client public key...
Generating server key pair...
Deriving shared key on server...
Sending server public key to client...
Receiving client shared key for verification...
Sending server shared key to client for verification...
Shared keys match. ECDHE verification completed.
Receiving data from client...
Creating server verifier...
Server verifier created. Salt: Pk3Qxg==, B: j805Lg3zwYborZo0ha/+qe6pTc04qM3Mp1V4eGqve6ChJR9S9t6oC2R4F0mEBFvkawkvnGss/BaX
umvKMl0JOJW7rKdyT/5wmMZhg1BDyPgiPInGyc60y0FY+zmJ9Uzm4smDj17w5M7nn1zb7x9VbhnxLfbixtcNo700H2Hxgi4Voiqf6DLZLNHU04Ble9Pwpe5y2
cUFyaELpsXdwNAjzBSPi4cearAlxVocProiVmI4pFEnamGBvcZNx0Q60zP9ihb4Z09WMYLROQKrzmCU1k9BPhjuJD01gLxE23DMYj4On24QaDHKukLvd7Oxo
oume08DHJ8xXggGIwJAa8YOrA==
Sending challenge to client...
Receiving M from client...
Verifying session on server...
Session verified on server. HAMK: leBqskYV1Dchkhip76hGk2rVbNU=
Sending HAMK to client...
Authentication process completed.
Server is authenticated.

```

FIGURE 11 – Verbose du script serveur sur le PC

Évaluation de la consommation d'énergie Un wattmètre a été utilisé pour mesurer la consommation énergétique du RPI en mode client et en mode serveur. Les mesures ont été effectuées avec le RPI en état de repos (idle), en tant que serveur, et en tant que client.



FIGURE 12 – Raspberry Pi en mode idle



FIGURE 13 – Raspberry Pi en mode serveur



FIGURE 14 – Raspberry Pi en mode client

Évaluation par rapport à une pile CR2032 Pour estimer la durée de vie d'une pile CR2032 en fonction de l'utilisation d'un script consommant 0.6W, nous devons prendre en compte plusieurs facteurs, notamment la capacité de la pile et la consommation énergétique par exécution du script.

Calculs de la durée de vie de la pile CR2032 :

- **Capacité d'une pile CR2032 :**
 - Une pile CR2032 typique a une capacité de l'ordre de 220 mAh (milliampères-heure).
 - La tension nominale d'une pile CR2032 est de 3V.
- **Énergie totale disponible :**

$$Energie = Capacite(Ah) \times Tension(V)$$

$$Energie = 0.22 Ah \times 3 V = 0.66 Wh$$

- **Consommation par exécution du script :**

$$Consommation = 0.6 W$$

- **Energie consommée par exécution (en watt-seconde ou joules) :**

$$Energie par execution = 0.6 W \times t s$$

— Nombre d'exécutions possibles :

$$0.66 \text{ Wh} = 0.66 \text{ Wh} \times 3600 \text{ seconds/hour} = 2376 \text{ Joules}$$

Le script s'exécute pendant 1 seconde à chaque fois :

$$\text{Nombred'executions} = \frac{3960}{1} = 3960$$

Avec une pile CR2032, et en supposant que le script consomme 0.6W pendant toute la durée d'exécution, vous pouvez utiliser la formule $\frac{3960}{t}$ pour déterminer le nombre d'exécutions possibles, où t est le temps en secondes que prend chaque exécution du script (ici mesuré 1 seconde).

Évaluation pour une utilisation quotidienne Pour une utilisation quotidienne à raison de 10 fermetures/ouvertures par jour, la pile tiendrait environ un an.

Comparaison avec Keeloq Keeloq est un autre protocole d'authentification utilisé pour des systèmes similaires. Voici une capture d'écran indiquant une consommation de 0.3W, en comparaison avec les 0.6W de cette implémentation SRP/ECDHE.

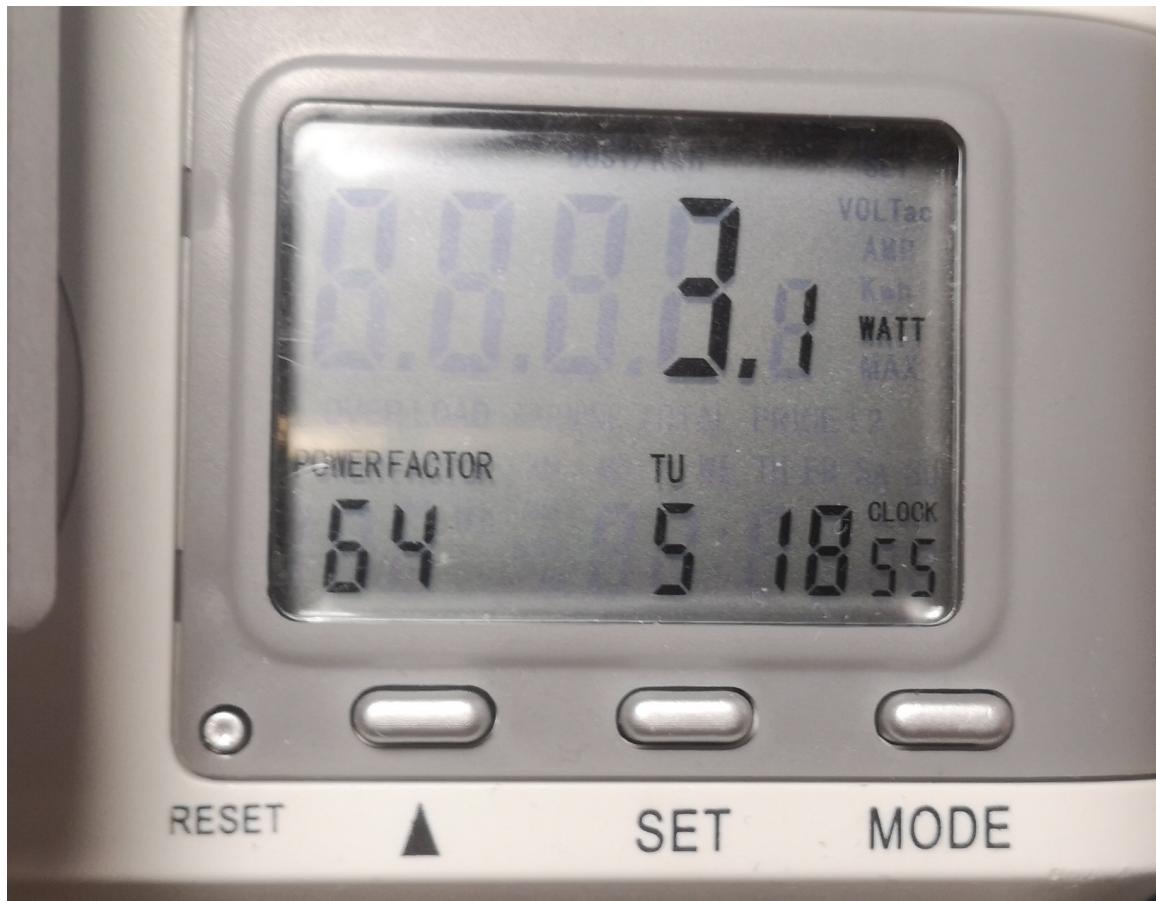


FIGURE 15 – Consommation de Keeloq : 0.3W

3.3 Ouverture : Utilisation de la clé partagée ECDHE pour chiffrer toutes les communications SRP

Jusque-là, nous avons utilisé les courbes elliptiques afin d'authentifier uniquement les échanges SRP. Mais une implémentation plus radicale serait d'utiliser en plus la clé partagée, pour chiffrer les communications SRP :

Avantages de l'utilisation de la clé partagée ECDHE L'utilisation de la clé partagée dérivée de l'ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) pour chiffrer toutes les communications SRP offre plusieurs avantages en termes de sécurité :

- **Confidentialité** : Toutes les données échangées entre le client et le serveur sont chiffrées, assurant que seuls les participants autorisés peuvent lire le contenu des messages.
- **Authenticité** : L'utilisation de clés partagées garantit que les communications proviennent bien des parties authentifiées.
- **Intégrité** : Le chiffrement empêche la modification non autorisée des messages échangés.
- **Protection contre les attaques de type Man-in-the-Middle (MITM)** : Même si un attaquant parvient à intercepter les messages, il ne pourra pas les déchiffrer sans la clé partagée.

Impact sur les performances et la consommation Cependant, chiffrer toutes les communications SRP avec la clé partagée ECDHE a un impact significatif sur les performances et la consommation énergétique :

- **Complexité de calcul** : Le processus de chiffrement et de déchiffrement des messages ajoute une charge de calcul considérable, ce qui peut ralentir les communications et augmenter le temps de réponse.
- **Consommation d'énergie** : Le chiffrement continu des messages nécessite une utilisation intensive du processeur, augmentant la consommation d'énergie. Ceci est particulièrement critique pour des dispositifs à faible consommation d'énergie comme les Raspberry Pi, où l'autonomie peut être un facteur limitant.
- **Temps de traitement** : Chaque message doit être chiffré et déchiffré, augmentant ainsi le temps nécessaire pour traiter les communications par rapport à une transmission non chiffrée.

4 References

1. **GitHub Repository** : L'entièreté du code et des implémentations sont disponibles à l'adresse suivante : <https://github.com/Foxanivia/Projet-intégrateur-RKS.git> Projet intégrateur RKS sur GitHub
2. **RFC 5054** : The Secure Remote Password (SRP) Protocol, T. Wu, 2007. Disponible à <https://tools.ietf.org/html/rfc5054> RFC 5054
3. **Diffie-Hellman Key Exchange** : *New Directions in Cryptography*, Whitfield Diffie and Martin Hellman, IEEE Transactions on Information Theory, 1976.
4. **ECDHE** : *Elliptic Curve Cryptography*, National Institute of Standards and Technology (NIST). Disponible à <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf> NIST ECC
5. **Keeloq Security** : *Analysis of the KeeLoq Secure Remote Keyless Entry System*, Courtois, N.T., Bard, G.V., Bogdanov, A. (2007). Disponible à <https://eprint.iacr.org/2007/055.pdf> IACR KeeLoq Analysis
6. **Brute Force Attacks** : Techniques et défenses contre les attaques par force brute, disponible à <https://github.com/X-Stuff/CudaKeeloqkeeloq> bruteforce
7. **Replay Attacks** : Attaques par rejeu sur les systèmes de clé électronique, exemples de CVE :
 - CVE-2019-20626 : Honda HR-V 2017
 - CVE-2023-33281 : Nissan Sylphy classic 2021
8. **RollJam Attacks** : *Practical Attacks on Remote Keyless Entry Systems*, disponible à <https://www.blackhat.com/us-16/materials/us-16-Garcia-RollJam-Radio-Frequency-Jamming-Attacks-Against-RKE-Systems-wp.pdf> Black Hat RollJam
9. **Elliptic Curve Cryptography** : *An Introduction to the Theory of Elliptic Curves*, Washington, Lawrence C., The Johns Hopkins University Press, 2008.
10. **Python Implementation of SRP** : Utilisation de la librairie Python 'srp', documentation disponible à <https://pypi.org/project/srp/> Python SRP
11. **Secure Remote Password (SRP) Protocole** : Présentation et implémentation théorique disponible à <https://srp.stanford.edu/ndss.html> SRP Overview
12. **Energy Consumption** : Estimation de la consommation d'énergie pour les dispositifs Raspberry Pi, mesurée avec un wattmètre dans différentes configurations de script (idle, client, serveur). <https://psutil.readthedocs.io/en/latest/psutils.html>