Constrained Pseudorandom Functions, Revisited

Julian Mouthon, Mario Razafinony
Master 1 Cryptologie, Calcul haute-performance et Algorithmique
Sorbonne Université

February 20, 2026

Introduction

Ce projet s'intéresse aux fonctions pseudo-aléatoires contraintes (*Constrained Pseudorandom Functions*, CPRF), une extension des fonctions pseudo-aléatoires classiques permettant de restreindre les évaluations à un sous-ensemble d'entrées défini par une contrainte.

Dans un premier temps, nous présentons les notions fondamentales liées aux fonctions pseudo-aléatoires et aux CPRFs, ainsi que la construction proposée par Bost, Minaud et Ohrimenko en 2017 (BMO17).

Dans un second temps, nous nous intéressons aux travaux récents de Cheng et Jaeger en 2025 (CJ25), qui mettent en évidence une attaque contre cette CPRF et proposent une variante hachée assurant la sécurité face à un tel attaquant.

Le code source est disponible sur GitHub : https://github.com/julian-mtn/bmo17-cprf.git

Contents

1 Définitions

Fonction pseudo-aléatoire (PRF)

Une fonction pseudo-aléatoire

$$F : \mathcal{K} \times D \to R$$

est une fonction telle que, pour une clé secrète K, la fonction $F(K, \cdot)$ est indiscernable d'une fonction réellement aléatoire $G : D \to R$ pour tout adversaire efficace.

Contrainte

Une contrainte est un prédicat booléen

$$C : D \to \{0, 1\},$$

qui partitionne le domaine D en deux ensembles:

- les entrées *autorisées*, telles que $C(x) = 1$
- les entrées *interdites* (ou contraintes), telles que $C(x) = 0$

Intuitivement, la contrainte spécifie l'ensemble des entrées sur lesquelles une clé contrainte est autorisée à évaluer la fonction pseudo-aléatoire.

Ici, nous considérons principalement des contraintes de la forme

$$C_n(x) = [x < n],$$

qui autorisent toutes les entrées strictement inférieures à un seuil $n \in \mathbb{N}$.

Fonction pseudo-aléatoire contrainte (CPRF)

Une fonction pseudo-aléatoire contrainte (CPRF) est donnée par quatre algorithmes efficaces:

- $\mathsf{Setup}(1^\lambda) \to K$ (clé maître): génère la clé secrète principale K.
- $\mathsf{Constrain}(K, C) \to K_C$ (clé contrainte): produit une clé spéciale K_C permettant d'évaluer la fonction uniquement sur les entrées autorisées par C.
- $\mathsf{Eval}(K, x) \to y$: évalue la fonction pseudo-aléatoire sur une entrée x avec la clé maître.
- $\mathsf{EvalC}(K_C, x) \to y$: évalue la fonction sur l'entrée x avec la clé contrainte K_C.

Une CPRF est correcte si, pour toute clé K, toute contrainte C et toute entrée x telle que $C(x) = 1$, on a

$$\mathsf{EvalC}(K_C, x) = \mathsf{Eval}(K, x).$$

Autrement dit, la clé contrainte permet d'évaluer la PRF exactement sur les entrées autorisées.

2 Construction de la CPRF de BMO17

2.1 Propriétés

La CPRF de BMO17 est basée sur l'itération successive d'une permutation à trappe, c'est-à-dire une fonction bijective facile à calculer mais difficile à inverser sans information secrète. On note cette permutation π.

Génération de la clé maîtresse

La clé maîtresse est composée des éléments suivants :

- $ST_0 \in \mathbb{Z}_N$, un état initial secret choisi aléatoirement
- SK, une clé secrète RSA définissant la permutation à trappe π_{SK}

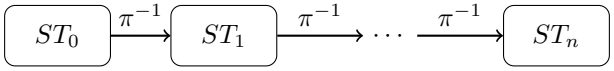
La clé publique associée PK permet uniquement de calculer la permutation directe π.

Evaluation de la CPRF avec la clé maîtresse

Avec la clé maîtresse (ST_0, SK) et une entrée c, on peut évaluer la CPRF sur c par :

$$Eval((SK, ST_0), c) = \pi_{SK}^{-c}(ST_0)$$

L'évaluation consiste à appliquer c fois l'inverse de la permutation à partir de l'état initial ST_0.



Génération de la clé contrainte

À partir de la clé maîtresse (ST_0, SK) et d'un entier n, correspondant à la contrainte $C(c) = [c < n]$, la clé contrainte est composée des éléments suivants :

- PK, la clé publique associée à la permutation π
- $ST_n = \pi_{SK}^{-n}(ST_0)$
- n

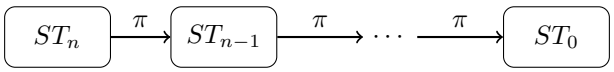
La clé secrète SK n'est pas incluse dans la clé contrainte.

Évaluation de la CPRF avec la clé contrainte

Avec la clé contrainte (PK, ST_n, n) et une entrée c, l'évaluation est possible uniquement si $c < n$. Dans ce cas, la valeur de la CPRF est calculée comme suit :

$$\mathsf{EvalC}((PK, ST_n, n), c) = \pi_{PK}^{n-c}(ST_n)$$

Cette valeur est égale à $Eval((SK, ST_0), c)$ par construction, ce qui assure la correction du schéma.



2.2 Choix d'implémentation

Nous utilisons la bibliothèque OpenSSL pour la gestion des grands entiers (`BIGNUM`), les opérations modulaires et le hachage SHA-256.

L'aléa est sécurisé et est obtenu à partir de `/dev/urandom`, ce qui permet de générer l'état initial ST_0 ainsi que les paramètres RSA de manière plus sûre.

2.2.1 Implémentation de la permutation à trappe RSA

La permutation à trappe utilisée dans BMO17 est défini à l'aide de RSA. Nous implémentons la génération de clés RSA de taille 4096 bits, ainsi que l'évaluation de la permutation dans les deux sens.

Plus précisément, la fonction publique correspond à:

$$x \mapsto x^e \bmod N,$$

tandis que l'inverse de la permutation est calculé via:

$$x \mapsto x^d \bmod N.$$

avec

$$x^{e \cdot d} = x \bmod N, \forall x \in \mathbb{Z}_N.$$

Afin de limiter les fuites par canaux auxiliaires, l'évaluation privée est réalisée à l'aide de la fonction `BN_mod_exp_mont_consttime` d'OpenSSL, qui garantit un temps d'exécution indépendant des données.

2.2.2 Implémentation de la clé maîtresse et génération de l'état initial

La clé maîtresse de la CPRF est composée d'un état initial secret ST_0 et d'une clé privée RSA. L'état initial est généré comme un entier aléatoire de 256 bits à partir de `/dev/urandom`..

L'évaluation avec la clé maîtresse consiste à appliquer c fois l'inverse de la permutation RSA à partir de ST_0.

2.2.3 Implémentation de la dérivation des clés contraintes

À partir de la clé maîtresse et d'un paramètre de contrainte n, nous dérivons une clé contrainte composée de la clé publique RSA (e, N) et de l'état

$$ST_n = \pi^{-n}(ST_0).$$

Cette clé permet ensuite d'évaluer la CPRF uniquement pour les entrées $c < n$ en appliquant $(n - c)$ fois la permutation publique à partir de ST_n. La clé privée n'est jamais incluse dans la clé contrainte, ce qui garantit que seules les évaluations autorisées sont possibles.

2.3 Étude de permutations alternatives à RSA

2.3.1 Permutation à trappe de Rabin

La permutation de Rabin est une fonction cryptographique à trappe fondée sur la difficulté de la factorisation des entiers. Elle est conceptuellement plus simple que RSA et repose sur l'opération de mise au carré modulo un entier composé.

- **Génération de la clé :**

  On choisit deux nombres premiers distincts p et q tels que :

  $$p \equiv q \equiv 3 \pmod 4$$

  La clé publique est définie par :

  $$n = p \cdot q$$

La clé privée est le couple (p, q).

Cette condition sur p et q permet un calcul efficace des racines carrées modulo ces nombres premiers.

- **Permutation de Rabin :**
  La permutation de Rabin est la fonction :

$$f : \mathbb{Z}_n \to \mathbb{Z}_n$$

définie par :

$$f(x) = x^2 \bmod n$$

Cette fonction est facile à calculer, mais difficile à inverser sans connaître la factorisation de n.

- **Inverser la permutation de Rabin :**
  Inverser la permutation de Rabin revient à résoudre l'équation :

$$x^2 \equiv y \pmod{n}$$

Lorsque $n = p \cdot q$, cette équation admet exactement **quatre solutions distinctes** modulo n.

On commence par réduire le problème modulo p et q :

$$\begin{cases} x^2 \equiv y \pmod{p} \\ x^2 \equiv y \pmod{q} \end{cases}$$

Comme $p \equiv 3 \pmod 4$, une racine carrée modulo p est donnée par :

$$r_p = y^{\frac{p+1}{4}} \bmod p$$

De même, modulo q :

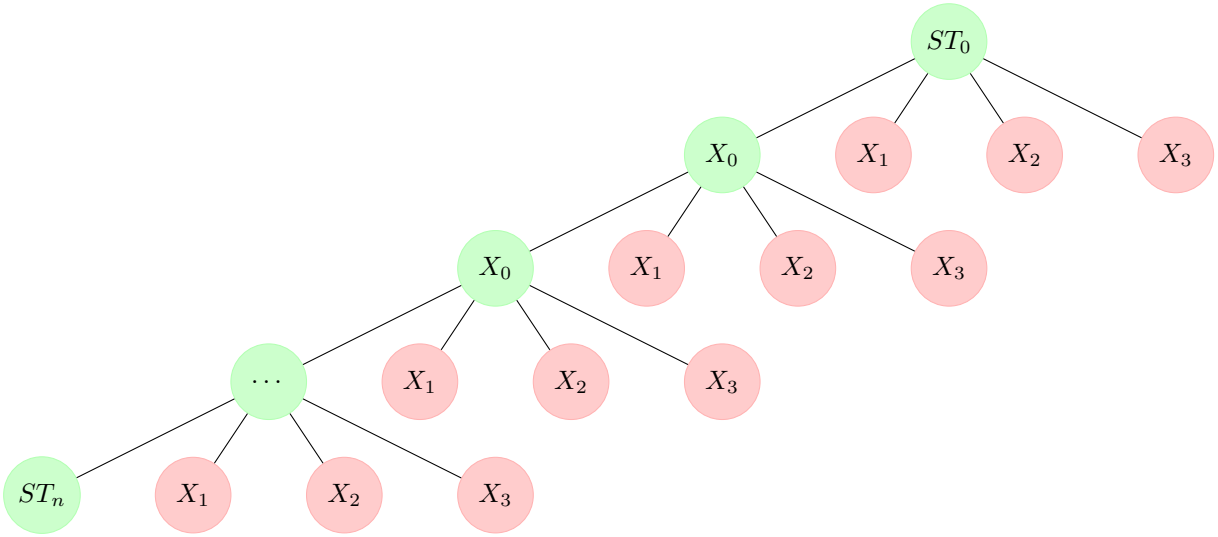
$$r_q = y^{\frac{q+1}{4}} \bmod q$$

Les solutions sont donc:

- $X_0 = (r_p * q * q^{-1}_{\bmod p} + r_q * p * p^{-1}_{\bmod q}) \bmod n$
- $X_1 = (r_p * q * q^{-1}_{\bmod p} - r_q * p * p^{-1}_{\bmod q}) \bmod n$
- $X_2 = (-r_p * q * q^{-1}_{\bmod p} + r_q * p * p^{-1}_{\bmod q}) \bmod n$
- $X_3 = (-r_p * q * q^{-1}_{\bmod p} - r_q * p * p^{-1}_{\bmod q}) \bmod n$

Problème: Laquelle de ces racines correspond au message original ?

La fonction $f(x) = x^2 \bmod n$ n'est pas injective: chaque chiffré c a 4 pré-images. Même avec la clé secrète (p, q), on peut calculer toutes les racines mais **pas déterminer directement celle qui correspond au message initial**. Appliquer l'inverse $n - c$ fois n'assure pas de retrouver le message original.



Conclusion : La permutation de Rabin n'est pas adaptée pour notre CPRF car chaque valeur a quatre pré-images possibles. Même avec la clé privée, il est impossible de savoir laquelle correspond à l'entrée originale, ce qui empêche l'application répétée de l'inverse $(n-c)$ fois pour retrouver correctement le résultat de la CPRF.

2.3.2 Permutation à trappe AES

Le chiffrement AES applique une fonction bijective $E_k(x)$ qui dépend de la clé k. Contrairement à $x \mapsto x^2 \bmod n$, chaque bloc chiffré a une **unique pré-image** pour une clé donnée. Cependant, pour déchiffrer, il faut absolument connaître la clé k :

- Sans la clé, l'attaquant ne peut pas inverser la permutation.
- AES reste sécurisé grâce à la clé secrète.

Donc l'utilisation d'AES comme permutation à trappe n'est pas adaptée pour notre CPRF.

3 L'attaque CJ25

3.1 Description générale et intuition de l'attaque

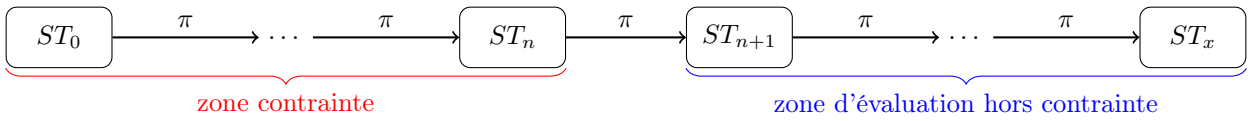
L'attaque CJ25 exploite le fait qu'un attaquant peut obtenir une clé contrainte permettant d'évaluer la fonction pseudo-aléatoire sur un sous-ensemble d'entrées, tout en conservant l'accès à un oracle d'évaluation global.

Dans un premier temps, l'adversaire choisit un indice n et demande à l'oracle une clé contrainte (PK, ST_n, n). Cette clé lui permet d'évaluer la fonction sur toute entrée x telle que $x > n$ à l'aide des permutations publiques associées.

Dans un second temps, l'adversaire interroge l'oracle d'évaluation sur une entrée $x > n$ et obtient une valeur ST_x. À ce stade, l'attaquant ne sait pas si ST_x provient d'une fonction pseudo-aléatoire contrainte ou d'une fonction aléatoire.

À l'aide de la clé contrainte, l'attaquant applique successivement les permutations publiques afin de calculer la valeur $\pi_{PK}^{x-n}(ST_x)$. Si l'attaquant retrouve $ST_a = ST_x$, il peut alors conclure que la valeur ST_x est le résultat d'une fonction pseudo-aléatoire contrainte.

Il est donc possible pour un attaquant d'avoir des informations sur la structure de la CPRF à partir de la clé contrainte et des évaluations obtenues, ce qui lui permet de distinguer une CPRF d'une fonction aléatoire avec une probabilité non négligeable.



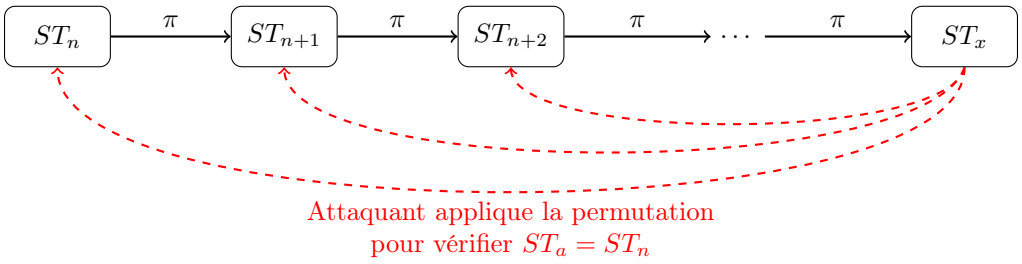
3.2 Modélisation du jeu de sécurité et avantage de l'attaquant

Jeu de sécurité pour les CPRF

L'attaquant $\mathcal{A}$ interagit avec un oracle O qui est soit une $CPRF$ soit une fonction aléatoire. L'objectif de l'attaquant est de distinguer les deux cas.

- **Phase 1 :** $\mathcal{A}$ demande une clé contrainte $K_C = (PK, ST_n, n)$ pour une contrainte n de son choix, lui permettant d'évaluer la fonction sur un sous-ensemble d'entrées.
- **Phase 2 :** $\mathcal{A}$ fait un nombre de requêtes d'évaluation à l'oracle O sur des entrées $x > n$ pour obtenir des valeurs ST_x non contraintes.
- **Phase 3 :** $\mathcal{A}$ utilise les valeurs obtenues pour évaluer $ST_a = \pi_{PK}^{n-x}(ST_x)$ et compare avec ST_n.
- **Décision :** $\mathcal{A}$ décide que O est la $CPRF$ si $ST_a = ST_n$, sinon il décide que O est une fonction aléatoire.

Dans le cas où O est une $CPRF$, l'attaquant $\mathcal{A}$ retrouve toujours $ST_a = ST_n$, et il ne se trompe jamais. En revanche, si O est une fonction aléatoire, la probabilité que $ST_a = ST_n$ est de $1/N$, où N est la taille de l'espace de sortie de la fonction. Dans ce cas, l'attaquant se trompe avec une probabilité de $1/N$. L'attaquant $\mathcal{A}$ gagne donc avec une probabilité de $1 - 1/N$ de distinguer correctement les deux cas, ce qui est non négligeable pour des tailles de N raisonnables.



3.3 Mise en œuvre expérimentale de l'attaque

Pour simuler le jeu de sécurité, nous avons mis en place une architecture client–serveur locale reposant sur TCP. Le serveur implémente l'oracle du jeu (évaluation et génération de clés contraintes), le client joue le rôle de l'attaquant (requêtes d'évaluation et de contrainte, tests, ...).

3.3.1 Implémentation de l'oracle de sécurité

L'oracle met à disposition deux interfaces accessibles à l'attaquant :

- `EVAL(x)` : retourne une valeur associée à l'entrée x.
  En monde PRF, l'oracle retourne l'évaluation de la CPRF à l'aide de la clé maîtresse.
  En monde aléatoire, il retourne une valeur uniforme de même taille.
- `CONSTRAIN(n)` : retourne une clé contrainte associée à l'indice n, permettant l'évaluation de la fonction sur un sous-ensemble d'entrées.

3.3.2 Implémentation de l'attaquant et stratégie

L'attaquant choisit un indice n et demande à l'oracle la clé contrainte correspondante. Cette clé lui permet d'évaluer la fonction sur des entrées strictement supérieures à n.

Ensuite il effectue des requêtes `EVAL` sur des entrées $x > n$. À partir des valeurs retournées par l'oracle et de la clé contrainte, il applique les permutations publiques afin de tester la cohérence de la structure de la CPRF.

Si l'attaquant trouve une correspondance, il peut conclure que l'oracle implémente une CPRF plutôt qu'une fonction aléatoire.

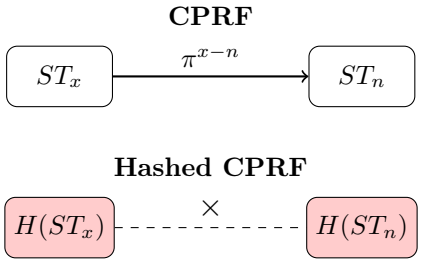
4 Renforcement de la CPRF par hachage

Afin d'améliorer la sécurité de la construction de BMO17 face à l'attaque CJ25, il est possible d'implémenter une version hachée de la CPRF.

4.1 Principe général de la CPRF hachée

L'idée est d'introduire une fonction de hachage cryptographique dans l'évaluation de la CPRF. L'attaquant pourra toujours évaluer la CPRF sur les entrées non contraintes, mais il ne pourra pas exploiter les résultats pour distinguer la CPRF d'une fonction aléatoire car ceux-ci seront hachés de manière irréversible.

→ Le rôle de la fonction de hachage est de rendre impossible toute utilisation ou comparaison via les permutations publiques.



4.1.1 Implémentation

Pour implémenter cette version hachée, nous avons utilisé la fonction de hachage **SHA-256** d'OpenSSL. L'évaluation de la CPRF est effectuée normalement, puis le résultat est converti en bytes et est haché avant d'être retourné à l'attaquant. De même pour l'évaluation avec la clé contrainte.

- $Eval_H(x)$: retourne le hachage de l'évaluation de la CPRF sur l'entrée x avec la clé maîtresse, c'est-à-dire $H(Eval((SK, ST_0), x))$.
- $EvalC_H(n)$: retourne le hachage de l'évaluation de la CPRF sur l'entrée n avec la clé contrainte, c'est-à-dire $H(EvalC((PK, ST_n, n), c))$.

Avec $Eval$ l'évaluation de la CPRF normale, $EvalC$ l'évaluation avec clé contrainte et H une fonction de hachage cryptographique.

4.2 Construction explicite de la CPRF hachée de Cheng et Jaeger

Nous présentons ici la version hachée de la CPRF telle que proposée dans l'article CJ25.

Notations utilisées :

- P : fonction de hachage idéale modélisée comme un oracle de hachage aléatoire.
- σ_P : état interne de P (initialement vide) qui mémorise les paires (x, y).
- λ : paramètre de sécurité (taille de la sortie de P).
- k : clé secrète de la CPRF.
- k_R : clé contrainte associée à un ensemble de restrictions R.
- x : entrée de la fonction.
- $y \in \{0,1\}^\lambda$: valeur de hachage associée à une entrée.

L'évaluation de la CPRF hachée est définie par :

$$\begin{aligned} \mathsf{Hashed[CPRF].Eval}(1^\lambda, k, x) &= P.\mathsf{LazySampling}(1^\lambda, \mathsf{CPRF.Eval}(1^\lambda, k, x) : \sigma_P). \\ \mathsf{Hashed[CPRF].EvalC}(1^\lambda, k_R, R, x) &= P.\mathsf{LazySampling}(1^\lambda, \mathsf{CPRF.EvalC}(1^\lambda, k_R, R, x) : \sigma_P). \end{aligned}$$

Lazy Sampling

Chercher x dans σ_P
Si $x \notin \sigma_P$:
  Choisir $y \leftarrow \{0,1\}^\lambda$ au hasard
  Mémoriser (x, y)
Sinon :
  Récupérer y associé à x
Retourner y

Algorithme de lazy sampling d'une primitive idéale.

4.3 Évaluation expérimentale et comparaison des attaques

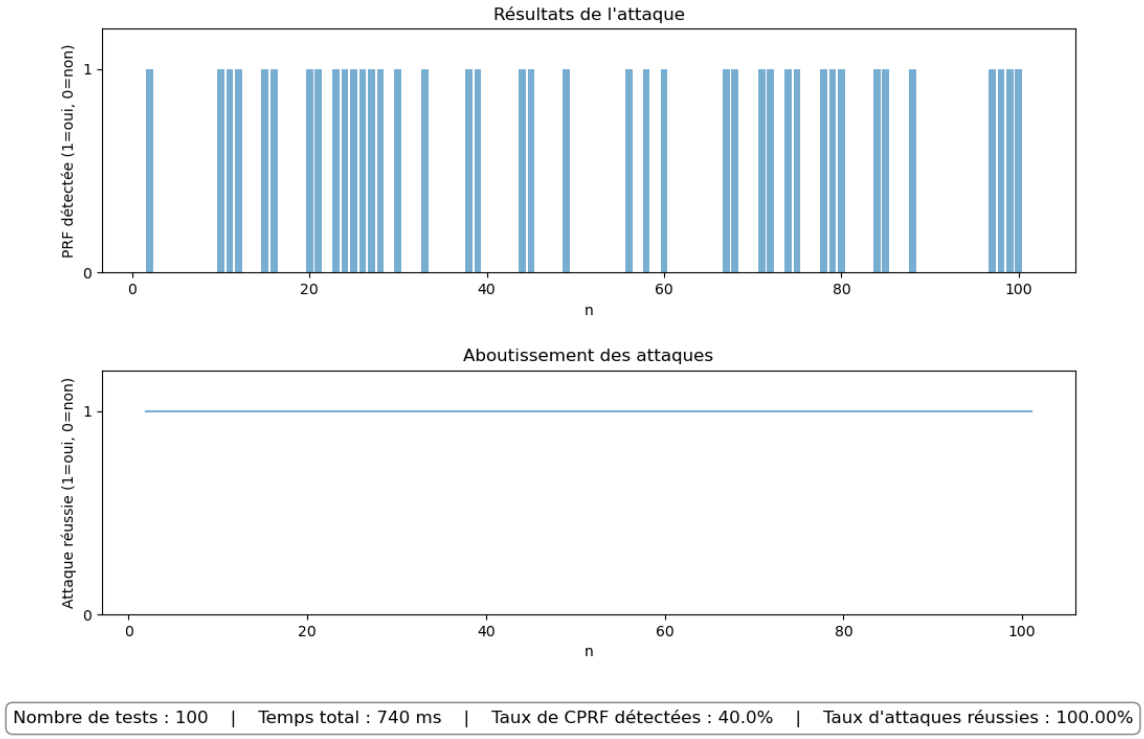


Figure 1: Attaque version non hachée

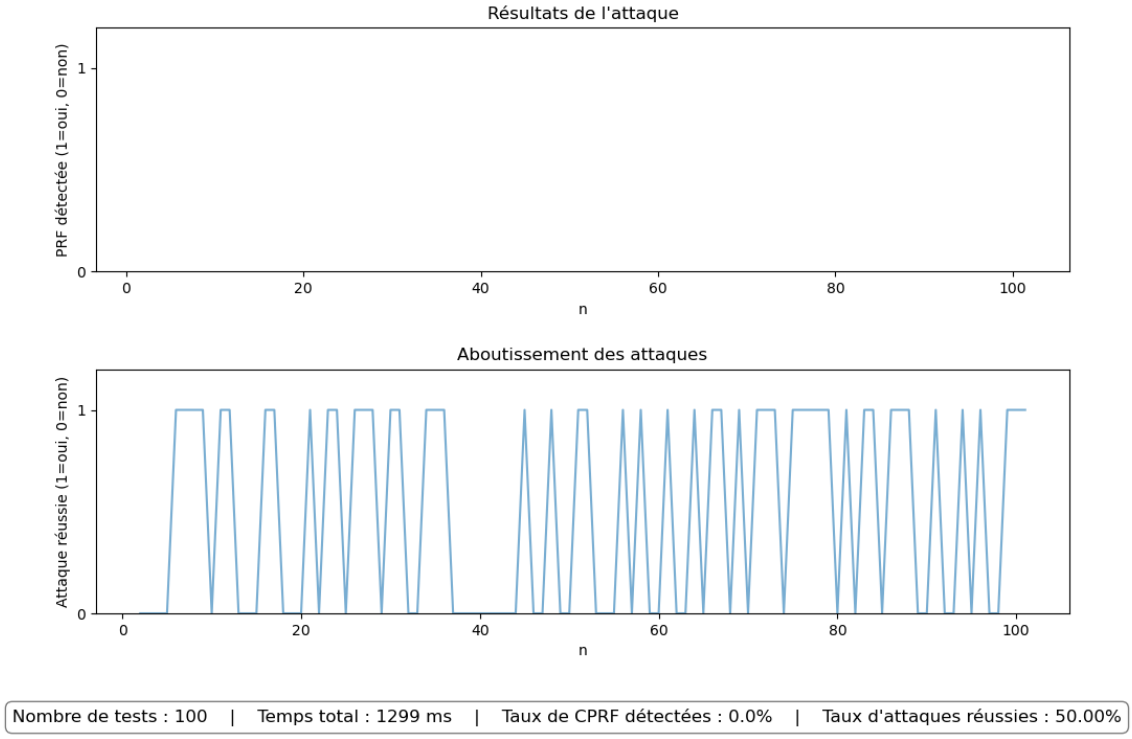


Figure 2: Attaque version hachée

5 Comment lancer le programme

Commande	**Description**
`make`	Compile le projet.
`./start.sh -n <size>`	Lance les tests avec la CPRF normale.
`./start.sh -h <size>`	Lance les tests avec la CPRF hachée.
`./start.sh -l <size>`	Lance les tests avec le lazy sampling.
`<size>`	Nombre de tests à exécuter.
`python3 display.py`	Trace les courbes à partir des résultats enregistrés dans `attack_results.txt`.