Schéma Cryptographique Probablement peu sûr

Evan Voyles

Schéma 7

On considère la famille de fonctions à sens unique définie par

$$f_p : \mathbb{Z}_p^\times \times \mathbb{Z}_p^\times \to \mathbb{Z}_{p^3}$$
$$(x, y) \mapsto x/y \bmod p^3$$

Nous considérons le cas où p premier.

Implantation

Avant de plonger dans notre IDE préféré et écrire du code, il faudrait reculer pour bien examiner la définition de cette famille f_p.

Espace de départ

Dans un premier temps, nous étudions l'espace de départ, $\mathbb{Z}_p^\times \times \mathbb{Z}_p^\times$. Pourvu que p est premier, nous avons l'espace de départ $\{1, 2, ..., p-1\} \times \{1, 2, ..., p-1\}$ avec cardinalité $(p-1)(p-1)$. Ensuite, nous étudions la division de x par y.

x / y

Il y a plusiers manières d'interpréter cette symbole trompeuse de la division. Si on ne fait pas attention, nous pouvons le parser automatiquement comme la division usuelle des entiers ce qui donne une fonction pas du tout intéressant. Autrement, on lit la symbole / comme la multiplication par une inverse multiplicative c'est-à-dire $x * y^{-1}$. Naturellement, on se pose la question suivante: dans quelle espace est y^{-1} une inverse ?

Nous avons imaginé deux cas: le premier cas utilise l'espace $\mathbb{Z}_p^\times$ déjà explicité, et l'autre cas s'en sert d'une sorte de plongement implicite dans l'espace latent $\mathbb{Z}_{p^3}^\times$. Dans tous les deux cas, pour chaque $y \in \mathbb{Z}_p^\times$, il existe une inverse multiplicative y^{-1}.

inv_p

Pour la fonction $\text{inv}_p : \mathbb{Z}_p^\times \to \mathbb{Z}_p^\times$, nous dérivons facilement une définition qui est triviale à implémenter en Python en faisant appel au théorème d'Euler. Comme y et p sont premier entre eux (par définition !), la congruence

$$y^{\varphi(p)} \equiv 1 \bmod p,$$

où φ est la fonction totient d'Euler, est vérifiée. Nous dérivons facilement une expression pour y^{-1} dans $\mathbb{Z}_p^\times$:

$$\begin{aligned} y^{\varphi(p)} &\equiv 1 \bmod p \\ y * y^{\varphi(p)-1} &\equiv 1 \bmod p \\ \Rightarrow y^{-1} &= y^{\varphi(p)-1} \bmod p \end{aligned}$$

Pour p premier nous avons l'implémentation efficace en python:

```python
def inv_p(y: int, p: int) -> int:
    """Compute the multiplicative inverse y of p in Z_p^*"""
    return pow(y, p - 1, p)
```

Ce dernier utilise l'algorithme 'rapide' d'exponentiation. Voici quelques examples de l'image de inv_p pour p petit:

$$\begin{aligned}\text{inv}_5(\mathbb{Z}_5^\times) &= \{1,3,2,4\}\\ \text{inv}_7(\mathbb{Z}_7^\times) &= \{1,4,5,2,3,6\}\\ \text{inv}_{13}(\mathbb{Z}_{13}^\times) &= \{1,7,9,10,8,11,2,5,3,4,6,12\}\end{aligned}$$

inv_{p^3}

La fonction d'inverse dans $\mathbb{Z}_{p^3}^\times$ et aussi facile à implémenter en python mais il faut un peu de calcul avant de s'y lancer. Dans un premier temps, nous justifions l'existence de cette $y^{-1} \in \mathbb{Z}_{p^3}^\times$ en remarquant que les facteurs premiers de p^3 sont $p * p * p$. Par conséquent, pour tout $y \in \mathbb{Z}_p^\times$, $\gcd(y, p^3) = 1$ et il existe une inverse $y^{-1} \in \mathbb{Z}_{p^3}^\times$. L'expression pour y^{-1} se dérive aussi facilement qu'avant en appliquant le théorème d'Euler:

$$\begin{aligned}y^{\varphi(p^3)} &\equiv 1 \bmod p^3\\ y * y^{\varphi(p^3)-1} &\equiv 1 \bmod p^3\\ \Rightarrow y^{-1} &= y^{\varphi(p^3)-1} \bmod p^3\end{aligned}$$

Par les propriétés de la fonction totient d'Euler, nous avons $\varphi(p^3) = p^2\varphi(p) = p^2(p-1)$, pour p premier. Nous finnissons avec l'implémentation suivante:

```python
def inv_p3(y: int, p: int) -> int:
    """Compute the multiplicative inverse of y in Z_p^3^*"""
    tot = p * p * (p - 1)
    p3 = pow(p, 3)
    return pow(y, tot - 1, p3)
```

Cette deuxième interprétation de l'inverse produit une image de $\mathbb{Z}_p^\times$ qui est beaucoup plus varié ainsi que saupoudré dans l'espace $\mathbb{Z}_p^3$. En comparaison avec inv_p, nous avons:

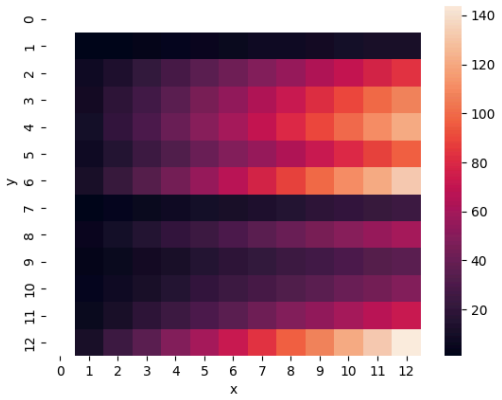
$$\begin{aligned}\text{inv}_{5^3}(\mathbb{Z}_5^\times) &= \{1,63,42,94\}\\ \text{inv}_{7^3}(\mathbb{Z}_7^\times) &= \{1,172,229,86,206,286\}\\ \text{inv}_{13^3}(\mathbb{Z}_{13}^\times) &= \{1,1099,1465,1648,879,1831,314,824,1953,1538,799,2014\}\end{aligned}$$

Une fois que nous avons défini l'inversion nous pouvons finalement implanté la famille de fonctions f_p.

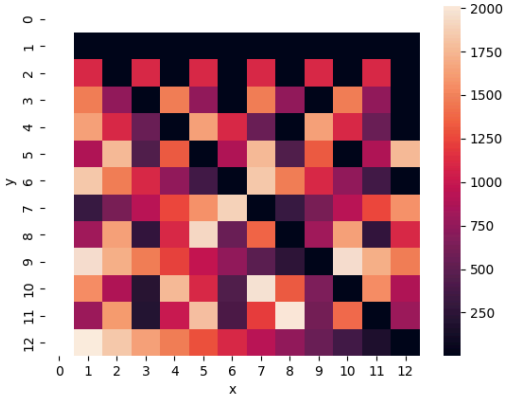
Voici l'implantation: https://github.com/ejovo13/cryptoa_rattrapage/blob/wip/crypto_rat/factory.py.

Visualization Géometrique

Il est tres util de visualizer les fonctions pour développer de l'intution géometrique (et donc, invariablement algebriquement aussi). Nous comparons ici la difference que notre choix de de inv fait, en dessinant les images de f_p comme un heatmap, en commençant avec $p = 13$:



f_{13} with $x/y := x * \text{inv}_p(y)$



f_{13} with $x/y := x * \text{inv}_{p^3}(y)$

Il y a deux différences remarquables. Dans un premier temps, à gauche, comme l'image de notre inversion avec inv_p reste dans $\mathbb{Z}_p^\times$, l'opération x/y est bornée par $(p-1)(p-1)$ donc on ne voit jamais un débordement dans le calcul $\bmod p^3$. La colonne tout à gauche est l'image de $\text{inv}_p(\mathbb{Z}_p^\times)$, et nous voyons clairement la progression des tons de gauche à droite ce qui représente les additions répétées de la multiplication. La ligne tout en haut est l'inverse de 1 (ce qui vaut toujours 1) multiplié par $x \in \{1, 2, ..., 12\}$, donc elle vaut $\{1, 2, ..., 12\}$.

A droite, en revanche, nous avons une image qui se répand partout dans l'espace $\mathbb{Z}_p^3$ et qui *semble* être plus chaotique, bien que il se manifeste une certaine structure algébrique. Pour commencer, il existe la même ligne tout en haut qui est, tout pareil, l'inverse de 1 (ce qui est 1) multiplié par x. Nous voyans aussi une bande diagonale qui provient du fait que $x/x = 1$. La dernière motif que nous soulignons c'est la répétition de certaines chaîne de couleurs, la longueur desquelles dépendante de la ligne. Par exemple, pour la deuxième ligne avec $y = 2$, nous voyons un bloc de 2 couleurs qui se répète. Après pour la ligne 3 il y a un motif de trois couleurs. Ainsi 4 pour la quatrième ligne. En effet, c'est parce que n multiplications de l'inverse de n renvoie toujours 1. De plus, nous observons que les valeurs s'augmente par 1 à chaque tour. Concrètement, prenons la deuxieme ligne comme exemple, où $y = 2$:

$$f_{13}(\mathbb{Z}_p^\times, 2) = \{f_{13}(1,2), f_{13}(2,2), ..., f_{13}(12,2)\}$$
$$f_{13}(\mathbb{Z}_p^\times, 2) = \{1099, 1, 1100, 2, 1101, 3, 1102, 4, 1103, 5, 1104, 6\}$$

Il est important à remarquer que notre visualisation est trompeuse parce que nous pouvons pas distinguer entre les tons numériquement adjacents. Par exemple, $f_p = 40$ et $f_p = 41$ nous semblent identique dans le heatmap.

La Sécurité, les Attaques

Pour p premier de n bits, nous rappelons que la cardinalité de $\mathbb{Z}_p^\times \times \mathbb{Z}_p^\times$ est (p - 1)(p - 1). Nous avons implémenté une premiere attaque exhaustive qui parcours le produit cartésien de $\mathbb{Z}_p^\times$ et $\mathbb{Z}_p^\times$ et qui peut craquer la "pre-image" d'une valeur z, c'est-à-dire trouver toutes les paires (x_i, y_i) tels que $f_p(x_i, y_i) = z$, dans quelques minutes pour des p moins de 15 bits. Cela devient rapidement à l'ordre des *jours* pour en trouver la pré-image pour p au-delà de 20 bits.

En revanche, la valeur de p dans notre challenge est de l'ordre de 128 bits, ce qui fournit assez de sécurité contre une attaque exhaustive.

Challenge
p = 0xd2bf071417608219223ad076131586a9

z =
0x4520670aac4c7f5af9f86bed585d6066dcb73b
9ec8c9b88536b46e252e64a1d28da6f8cf0d8bbf60fa6a4f8ee9854909

Nous avons pas réussi à trouver une attaque efficace pour trouver une pré-image de z. Toutes les stratégies que nous avons imaginés visaient s'en servir des motifs cycliques que l'on observe pour les multiplications des y^{-1}. Par contre, nous avons pas trouvé une manière pour réduire l'espace de recherche; nous avons toujours commencé par calculé les inverses de y dans $\mathbb{Z}_{p^3}^\times$, mais même cette première étape est beaucoup trop coûteuse pour une p avec 128 bits.

Fonction de hachage: Schwa7

Nous pouvons utiliser schéma 7 pour créer une fonction de hashage qui accepte les entrées de n'importe quelle taille et qui renvoi une valeur dans $\mathbb{Z}_{p^3}$. Cette partie décrit la fonction de hachage, ə7 (prononcé 'schwa sept'), qui est basé sur schéma 7.

Etymologie
Le nom de ə7 est une amalgamation de trois choses - 'schéma 7', 'SHA', et la voyelle 'ə'. Schéma7 parce que, bien entendu, il est le moteur de « aléa » de notre fonction, SHA parce que ceci est une function de hachage, et finalement schwa est tout simplement un hômage au phonème adoré ə

Spécification
Afin de transformer la schéma 7 en fonction de hachâge, nous devons prescrire un protocole pour découper une entrée de n'importe quelle taille en morceaux que f_p peut consommer.

Pour ce faire, nous imaginons une première transformation de $\mathbb{Z}_{(p-1)(p-1)} \to \mathbb{Z}_p^\times \times \mathbb{Z}_p^\times$ qui serait outil après pour agrandir l'espace départ à $\{0,1\}^*$.

La transformation que nous choissisons peut être imaginé comme une décomposition de notre valeur d'éntrée en deux chiffres avec base (p - 1).

Prenons un exemple concret avec $p = 5$:

binary	decimal	x	y	$f_5(x,y)$
0000	0	1	1	1
0001	1	1	2	63
0010	2	1	3	42
0011	3	1	4	94
0100	4	2	1	2
0101	5	2	2	1

binary	decimal	x	y	$f_5(x,y)$
1000	8	3	1	3
1001	9	3	2	64
1010	10	3	3	1
1011	11	3	4	32
1100	12	4	1	4
1101	13	4	2	2

binary	decimal	x	y	$f_5(x, y)$
0110	6	2	3	84
0111	7	2	4	63

binary	decimal	x	y	$f_5(x, y)$
1110	14	4	3	43
1111	15	4	4	1

Cette enumération étant très naturelle, maintenant il nous reste à définir un protocole pour "condenser" des entrées que sont plus grand que $(p-1)(p-1)-1$

Nous définnissons la fonction de condensation de ə7 comme la suivante:

$$\begin{aligned} \varphi : \mathbb{Z}_p^\times \times \mathbb{Z}_p^\times &\to \mathbb{Z}_p^\times \\ (x, y) &\mapsto \big(f_p(x, y) \bmod p - 1\big) + 1 \end{aligned}$$

Cette fonction est un peu maladroite mais elle se sert à deux utilités:

1. Elle combine 2 valeurs en $\mathbb{Z}_p^\times$ dans une seule
2. Elle utilise une application à f_p pour en faire.

Les $(p-1)$ et le $+1$ c'est juste de la comptabilité pour que l'on puisse utiliser la sortie de φ dans des appels consécutives.

On est maintenant prêt à implémenter ə7, toujours paramétrisés par p. Nous décrivons l'algorithme en language naturel:

1. Décomposér une entrée de n bits en tranches de n_{p-1} bits, n_{p-1} étant la longeur de $(p-1)$ en bits.
2. Condenser la séquence $[t_0, t_1, ..., t_2]$ en enchaînant les appels à φ jusqu'a ce que il en reste que deux valeurs
3. Renvoyer $f_p(\cdot, \cdot)$ appliquée aux deux valeurs restant.

Exemple

Étudions le cas où $p = 7$ et prenons 697 comme entrée. Dans un premier temps, nous décomposons l'entrée 697 en tranches pour que nous finnissons avec une suite de valeurs $t_i \in \mathbb{Z}_7^\times$:

$$\text{decompose}(697) = [2, 3, 2, 4]$$

Pour implanter ce comportement, nous avons décomposé 697 en chiffres de base 6, ensuite nous avons rajouté 1 pour finir dans l'espace $\mathbb{Z}_7^\times$. En Python:

```python
def split_bytes_to_int(bs: bytes, radix: int) -> list[int]:
    """Break up bytes into a list of base-radix digits."""
    z = int.from_bytes(bs)

    lst_int: list[int] = []
    while z >= radix:

        r = z % radix
        lst_int.append(r)
        z = (z - r) // radix

    lst_int.append(z)

    return lst_int
```

Une fois que nous avons une liste d'entiers dans $\mathbb{Z}_7^\times$, nous pouvons la condenser à l'aide de notre fonction φ. Pour rappel, voici la définition de notre fonction de condensation:

```
def condense(self: Schwa, x: int, y: int) -> int:
    """Use f_p to condense (x, y) -> f_p(x, y) mod (p - 1) + 1"""
    return (self.fn(x, y) % (self.fn.p - 1)) + 1
```

Nous réduisons une liste en accumulant à gauche:

$$\begin{aligned}\text{condense\_list}_7([2,3,2,4]) &\to [\varphi(2,3),2,4]\\ &\to [\varphi(\varphi(2,3),2),4]\\ &= [\varphi(5,2),4]\\ &= [3,4]\end{aligned}$$

Finalement, nous appliquons schéma 7 aux deux valeurs restantes:

$$f_7(3,4) = 258$$

Nous avons donc calculé:

$$ə7_7(697) = 258$$

Nous pouvons instancier une nouvelle fonction de hachage avec un appel à `Schwa`:

```
In [1]: from crypto_rat import Schwa
In [2]: s7 = Schwa(7)
In [3]: s7.hash(697, salt=False)
Out[3]: 258
```

Afin d'éviter que $ə_p(0)$ soit toujours égale à 1,

```
In [4]: s7.hash(0, salt=False)
Out[5]: 1
```

nous pouvouns préfixer une "salt" à la liste des entiers à condenser :

```
In [4]: s7.hash(0)
Out[4]: 2
```

Pour un p plus grand cela donne:

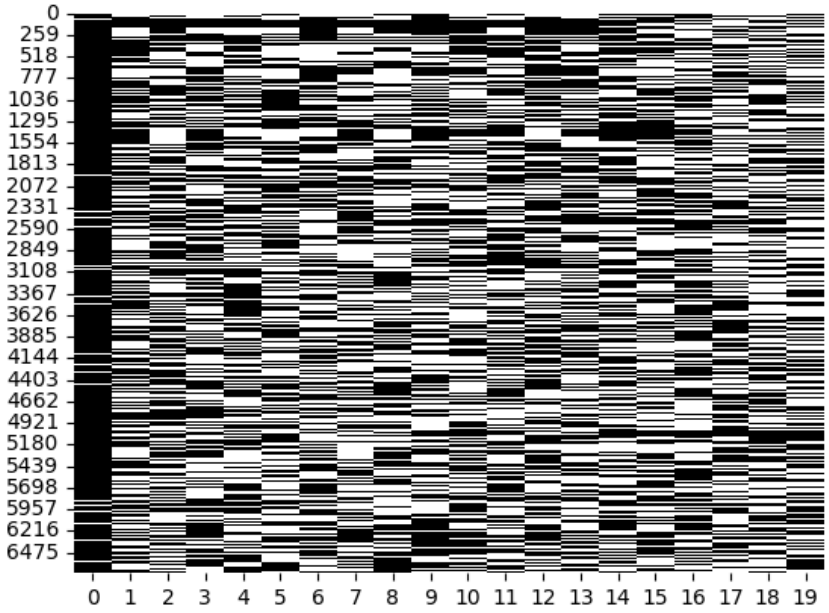
```
In [5]: Schwa(280129751383477787709680355788933400233).hash(0)
Out[5]: 159924924074128321776106280278662238741

In [6]: Schwa(280129751383477787709680355788933400233).hash(0, salt=False)
Out[6]: 1
```

Nous remarquons que cette fonction $ə7_p$, une fonction de hachage basé sur schéma7, est une fonction complètement distincte de f_p, avec ses propres propriétés, motifs, et espace de départ. De plus, le protocole pour réduire une liste d'entiers que nous avons choisit est complètement arbitraire.

Quoique pas très complexe, il était gratifiant d'imaginer et d'implanter cette fonction de hachage.

Pour p petit (prenons $p = 83$), nous pouvons jeter un coup d'oeuil sur la répartition des bits des valeurs sortants $ə7_p$:



La répartion des bits pour l'image de $\text{ə}7_{83}(n)$ sur $\mathbb{Z}_{82*82-1}$. L'axe des X désigne le j-ième bit alors que l'axe des Y correspond à l'entrée de $\text{ə}7_{83}(n)$. Une rectangle blanc indique une valeur de 1.