Rapport PSTL

Djadji Traore

April 19, 2019

1 Introduction

Pour pouvoir utilisé des fonctions cryptographique de rust, nous avons choisis d'utiliser la librairie (en rust un **crate**) **rust-crypto**. Et pour pouvoir utilisé ce crate nous avons besoin de l'ajouté au dépendances du projet, dans notre fichier *Cargo.toml*.

```
[dependencies]
rust-crypto = "^0.2"
```

Le petit caret à côté de la version de rust-crypto, permet au programme de mettre à jour les versions de ses dépendances, tant que le chiffre le plus à gauches n'est pas modifié. Ici, ce caret nous permet d'aller jusqu'à une version strictement inférieur à la version 1.0.0.

2 Me familiariser avec Rust-crypto

Ma première tâche fut de me familiariser avec rust-crypto, notamment, le module **AesSafe**, qui implémente l'algorithme de AES, avec deux implémentation différente, une où le programme fait des opération sur un bloc à la fois et la deuxième sur 8 bloc en parallèles, et un bloc peut faire 128, 192 ou 256 bits. Pour me familiariser avec AesSafe, je me suis arrêté sur les structures utilisant 128 bits :

- AesSafe128Decryptor
- AesSafe128DecryptorX8
- AesSafe128Encryptor
- AesSafe128EncryptorX8

Chacune de ces structures, prend en arguments une clef sous forme de tableau d'entier non signé de 8 bits. Chacune de ces structures, a aussi une fonction *encrypt_block* ou *decrypt_block* suivit de *_x8* en fonction de son implémentation de AES. Pour ne pas être redondant dans l'explication du code, nous nous concentrerons sur un exemple, le **AesSafe128Encryptor**. Tout d'abord, nous

définissons quel mudule nous utilisons, un module rand nous permettant de générer des nombres aléatoires, différent module de rust-crpto, tel que les Block Encryptor et Decryptor, que nous verrons plus tard et AesSafe :

```
use rand::{OsRng, Rng};
use crypto::symmetriccipher::{BlockEncryptor, BlockDecryptor,
    BlockEncryptorX8, BlockDecryptorX8};
use crypto::aessafe::{AesSafe128Encryptor, AesSafe128Decryptor,
    AesSafe128EncryptorX8, AesSafe128DecryptorX8};
```

Voici une partie du main, qui, creé une instance de OsRng qui est un génétateur de nombre aléatoire, si ce dernier génère une erreur, renvoie un message d'erreur.On crée ensuite un vecteur d'entier **key**, avec une suite de nombre aléatoire, qui crée un tableau input rempli de 0 et qui passe toute ces variables dans une fonction encrypt.

```
//creation of a random generator
let mut random = OsRng::new().expect("Failed to get OS random
    generator");
let mut key: Vec<u8> = repeat(0u8).take(16).collect();
//fill the key with random unsigned int value
random.fill_bytes(&mut key[..]);
let input : [u8;16] = [0;16];
let crypter : [u8;16] = encrypt(&input, &key);
```

La fonction encrypt, prend en paramètre une clef, et un tableau d'entier de taille 16, input, à crypter et renvoie un tableau d'entier de taille 16, correspondant à l'input crypter. Dans cette fonction, nous instancions un AesSafe128Encryptor en lui passant la référence de la clé, puis nous utilisons la fonction *encrypt_block* de Aessafe.

```
/* Encrypt a array of unsigned int (128 bits)
return the encrypted array as an array of unsigned int
*/
fn encrypt(input : &[u8], key : &[u8]) -> [u8;16] {
    let mut output =  [0u8;16];
    //initialize the Encryptor
    let encryptor = AesSafe128Encryptor::new(&key);
    encryptor.encrypt_block(&input, &mut output);
    return output
}
```

Voici la fonction *encrypt_x8* :

```
/* Encrypt a array of unsigned int (128 bytes = 8 block of 128 bits
    )
return the encrypted array as an array of unsigned int
*/
fn encrypt_x8(input : &[u8], key : &[u8]) -> [u8;128] {
    let mut output =  [1u8;128];
    //initialize the Encryptor
    let encryptor = AesSafe128EncryptorX8::new(&key);
    encryptor.encrypt_block_x8(&input, &mut output);
    return output
}
```

Les autres fonctions, telles que *decrypt*, *encrypt_x8*, **decrypt_x8**, suivent le même procédé avec évidemment, les fonctions correspondantes.

3 Benchmark avec Criterion.rs

Criterion.rs est une bibliothèque d'analyse comparative de Rust qui vise à conférer une fiabilité statistique solide à l'analyse comparative du code de Rust, elle nous aide à écrire un code rapide en détectant et en mesurant les améliorations ou régressions de performances, même les plus petites, rapidement et avec précision. Criterion permet également de connaître la vitesse à laquelle une portion de code traite les données, en *Bytes* ou *Elements* par secondes, tout deux défini par la taille de l'élément à traiter, en utilisant le throughput.

Pour commencer avec Criterion.rs, on doit ajouter ce qui suit à notre fichier Cargo.toml:

```
[dev-dependencies]
criterion = "0.1.2"
[[bench]]
name = "my_benchmark"
harness = false
```

4 Mesure des fonctions de cryptage de Aessafe

En mesurant les performances des fonctions d'Aessafe, nous voulions vérifier que les fonctions d'Aessafe faisant des opérations sur huit bloc en parallèles sont bien au moins huit fois plus rapide que leur fonction classique, ne prenant que un bloc à la fois. Nous avons donc repris les fonction *encrypt* et *encrypt_x8*, vu plus haut et nous les avons passé dans un *bench*. En rust,il y a quelques principes fondamentaux, l'un est le principe d'*ownership* et un autre est le principe de *reference*:

- Chaque valeur a un seul propriétaire. Donc, si nous passons une valeur dans une fonction et que nous ne lui rendons pas sa valeur, à la fin de la fonction, cette valeur sera perdu car elle n'aura plus de propriétaire.
- Mais en Rust, nous avons aussi le principes de réferences, qui nous permet de *prêter* la valeur de notre variable. C'est pourquoi des lignes *2* à *4*, nous avons des variables static contenant des réferences aux valeurs, que nous passons dans nos fonctions de benchmark.

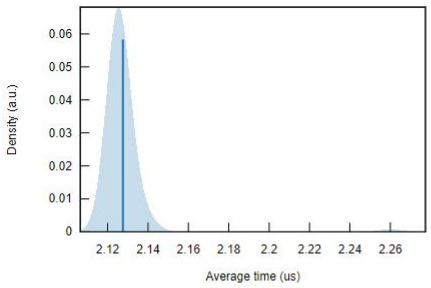
```
fn encrypt_benchmark(c: &mut Criterion) {
    static INPUT_X8 : &[u8;128] = &[0;128];
    static INPUT : &[u8;16] = &[0;16];
    static KEY: &[u8;16] = &[0;16];
    c.bench(
        "throughput Encrypt",
        Benchmark::new(
            "classic Encrypt",
            |b| b.iter(|| encrypt(INPUT, KEY)),
        ).throughput(Throughput::Bytes(INPUT.len() as u32)),
    );
    c.bench(
        "throughput Encrypt x8",
```

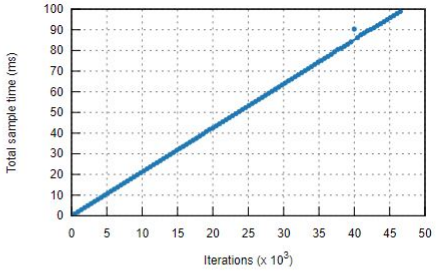
```
        Benchmark::new(
            "x8 Encrypt",
            |b| b.iter(|| encrypt_x8(INPUT_X8, KEY)),
        ).throughput(Throughput::Bytes(INPUT_X8.len() as u32)),
    );

}
```

A la fin de notre benchmark, Criterion génère des dossiers, en fonction du nom de nos benchmark, contenant toutes les données générer lors des mesures et parmi eux une page contenant deux graphiques, les moyennes de temps par itérations et la régression linéaire des mesures, cette page contient également d'autres statistiques. Elle se présentent sous la fome :

Figure 1: Page du report de la fonction encrypt





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	2.1263 us	2.1300 us	2.1368 us
Throughput	7.1410 MiB/s	7.1637 MiB/s	7.1763 MiB/s
R^2	0.9876357	0.9880546	0.9866874
Mean	2.1257 us	2.1276 us	2.1307 us
Std. Dev.	2.6010 ns	13.803 ns	23.214 ns
Median	2.1248 us	2.1248 us	2.1250 us
MAD	309.37 ps	523.49 ps	860.64 ps

Additional Plots:

- Slope
- Mean
- Std. Dev.
- Median
- MAD

Ici, on voit que le throughput de la fonction encrypt est de **7.14 Mib/s**, le throughput de la fonction encrypt_x8 est de **48.18 Mib/s**. On peut donc en conclure que les fonction d'Aessafe qui prennent huit bloc en parallèles sont bien au moins huit fois plus rapide. Etant donnée que dans les fonctions que nous mesurons, nous calculons aussi le fait d'instancier une structure de Aessafe, le temps n'est pas le vrai temps de calcule pour faire seulement les opérations de cryptage des bloc, mais on peut imaginer que cela n'impact pas de manière déraisonable nos résultat.

5 Client: mesure la latence

Pour analyser le temps du système de l'analyse, nous avons besoin d'une infrastructure d'évaluation client/serveur. De nombreux clients se connectent

au serveur, demandant le cryptage de certaines données pendant que le serveur tente de maximiser le débit.
Du coté du client, lors de l'envoi d'un message, on enregistre le temps. Dès réception de la réponse du serveur, on indique le temps aller-retour (RTT) et calcule la moyenne et l'écart type pour le RTT.

```
    // Upon sending a message, record a timestamp
    let start = Instant::now();
    //write
    stream.write(&r).unwrap();
    //read
    let mut buffer = [0; 8];
    stream.read(&mut buffer).unwrap();
    // when receive its response, report the round trip time (RTT)
    let duration = start.elapsed();
    println!("Time elapsed is {:?}", duration);
```

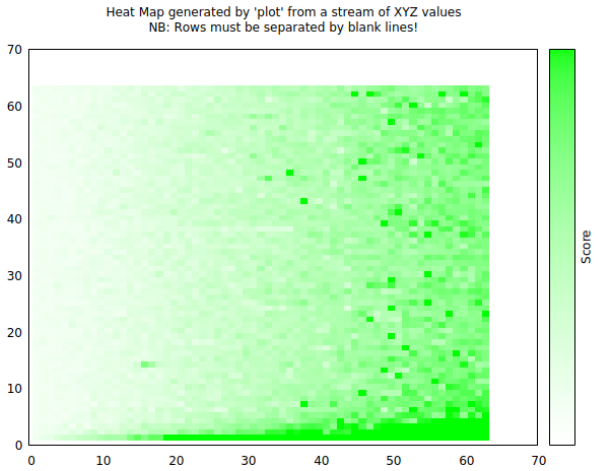
Pour configurer le nombre de demandes simultanées envoyées par le client, on a utilisé de threads pour envoyer le message simultanément. On fait RTT comme le retour de fonction associé au thread. Après le retour de la jointure, On récupere le resultat de RTT retourné et l'ajoute dans un vecteur.

```
let mut vec_data = Vec::new();
//Send n requests at the same time
for _i in 0..n {
    let handle = thread::spawn(move || {
     ......
     duration.as_micros()
}
for t in vec_thread {
    let d = t.join().unwrap();
    vec_data.push(d);
}
```

Et on calcule le moyen de RTT pour n requêtes simultanées et ensuite l'écrit dans un fichier.

On a effectué un post-traitement sur les résultats de latence obtenus en combinant le serveur avec m travailleurs et le client avec n requêtes simultanées. On trace un heatmap en utilisant Gnuplot sur ces résultats:

Example of positioning figures


Heat Map generated by 'plot' from a stream of XYZ values
NB: Rows must be separated by blank lines!
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70
Score