BLEUZE Baptiste
GARCIA Cédric

Introduction to cryptology
TP

Question 1 :

cf implantation

Question 2 :

En l'état actuel, une clé de 128 bits est *a priori* suffisante pour être préservée d'une attaque par force brute(probabilité de 2^{-128} de trouver la clé) . De plus, comme SipHash est certifiée pseudo-aléatoire, c'est le seul moyen pour trouver la clé.
En revanche, on ne peut considérer que SipHash soit résistante aux collisions car elle produit un tag de 64 bits seulement. Il est donc possible de trouver une collision pour une clé fixée avec une probabilité d'au moins 1/2 après avoir haché 2^{32} messages distincts selon le paradoxe des anniversaires. En effet, ce paradoxe nous dit que si les images d'une fonction sont indépendantes des unes des autres et uniformément réparties (c'est le cas de SipHash qui est une fonction pseudo-aléatoire) alors on trouve une collision après $2^{n/2}$ tentatives en moyenne (avec n la longueur en bits de l'image).

Question 3 :

Nous avons choisi comme fonction d'association d'une clé de 32 bits à une clé de 128 bits la fonction suivante :

soit b_i pour i allant de 0 à 3 les octets qui composent la clé de 32 bits en entrée :

clé de 32 bits en entrée : b3 b2 b1 b0

clé de 128 bits en sortie : b3 0 0 0 b2 0 0 0 b1 0 0 0 b0 0 0 0

Question 4 :

Nous avons choisi d'implanter l'algorithme qui consiste à stocker les hashs dans un ensemble :
Pour chaque message, on calcule son hash et on l'insère dans l'ensemble s'il ne s'y trouve pas, sinon cela signifie que l'on vient de trouver une collision.
Pour un tag de 32 bits, on trouve en moyenne une collision après $2^{16} = 65536$ itérations.
Une liste serait suffisante pour stocker les hashs mais pour accélérer la recherche, nous avons décider d'opter pour une table de hachage qui permet l'insertion et la recherche d'un éléments en temps constant en moyenne. Nous avons utilisé la structure de données « unordered_set » de la STL du langage C++ qui convient parfaitement à nos besoins.
Enfin, pour minimiser le nombre de collisions et de possibles rehachages de notre table, on initialise autant de tables que nous devons faire de tests tout en réservant pour chaque table un nombre de « buckets » égal à 2^{16} car on s'attend en moyenne à stocker 65536 hashs par table.

Question 5 :

Après avoir testé 1000 clés choisies aléatoirement, on trouve une collision après 81976 itérations en moyenne (3666 pour la valeur min, 216565 pour la valeur max). 81976 étant compris entre $2^{16} = 65536$ et $2^{17} = 131072$, ce résultat confirme notre estimation.

Question 6/7 :

cf Implantation

Question 8 :

twine_fun1 utilise une clé de 32 bits qui est bien inférieur au standard actuel de 128 bits recommandé pour se prémunir d'une attaque par force brute. Il est donc possible de trouver la clé en un temps raisonnable. Quant aux collisions, le bloc renvoyé étant de longueur 32 bits, on s'attend comme pour la fonction SipHash à trouvé une collision après $2^{16} = 65536$ itérations.

Question 9 :

Après avoir testé 1000 clés choisies aléatoirement, on trouve une collision après 83663 itérations en moyenne (2089 pour la valeur min, 245742 pour la valeur max). 81976 étant compris entre $2^{16} = 65536$ et $2^{17} = 131072$, ce résultat confirme notre estimation. La résistance aux collisions de twine_fun1est la même que celle de sip_hash_fix32.

Question 10 :

cf Implantattion

Question 11 :

Stats de twine_fun2_fix16 :

moyenne : 55614
max : 65536
min : 1912

Stats de twine_fun2_fix32 :

moyenne : 319
max : 1001
min : 5

On constate que l'on trouve beaucoup plus facilement de collisions par rapport à sip_hash_fix32 et twine_fun1. Cela est dû au fait que twine_fun2 réduit davantage l'espace des sorties possibles et donc augment sensiblement la probabilité de trouver une collision.