Crypto Engineering (GBX9SY03)

TP — Generic second preimage attacks on long messages for narrow-pipe Merkle-Damgård hash functions

Moran Antoine && Gindrier Clément

Structure :

Pour exécuter les tests, lancer : `make test`
Pour lancer l'attaque, lancer : `make attack` (attention, l'attaque de base contient très peu d'information sur le déroulement de l'attaque).

Pour lancer l'attaque, lancer : `make attack_verbose`.

Une fois l'attaque terminée, 2 fichiers texte sont générés, contenant les hash intermédiaires des 2 messages afin que vous puissiez vérifier leur bon fonctionnement ainsi que leur structure interne.

Les fichiers `.c` sont dans le dossier `src`
Les fichiers `.h` sont dans le dossier `include`

Le fichier `src/second_preim_48_fillme.c` contient les fonctions pour implémenter l'attaque. Ces fonctions vont être utilisées dans `test.c` et `attack.c`.
Le fichier `src/test.c` contient le `main` pour les tests ainsi que les fonctions de test. Le fichier `src/attack.c` contient le `main` pour l'attaque.

Part two: the attack

Question 1:

Nous avons choisi de stocker les hashes dans un tableau que l'on trie, puis faire une recherche par dichotomie.
Pour optimiser la recherche d'une collision, nous suivons le principe du paradoxe des anniversaires, et nous calculons donc 16 millions d'éléments car cela correspond à la racine carré de toutes les sorties possibles, soit $\sqrt{(2^{48})} = \sqrt{(2^{24} = 16777216)}$ éléments. Puis nous les trions en 5 secondes. Et ensuite, nous générons de nouveaux messages aléatoires

que nous hashons avec `get_cs48_dm_fp`, et on regarde si le hash est dans le tableau avec de la dichotomie. Il faut en moyenne 16 millions d'essais avant d'en trouver un qui correspond. Le programme arrive à tester plus d'1 million de message par seconde.
Le tri sera fait en O(nlog n) par un tri rapide, implémenté par la fonction qsort. Il prend 5 secondes. La dichotomie fait en moyenne 23 boucles, et en tout, la recherche de collision prend en moyenne 20 secondes (mais c'est très variable).
Le temps pour l'attaque est raisonnable, donc on ne l'a pas optimisé. Néanmoins, quelques idées d'optimisations que l'on a essayé mais qui n'ont jamais abouties :

- Utiliser des tables de hachage.
- Optimiser la dichotomie avec un algorithme d'"interpolation search" en O(log log n). Puisque les éléments sont réparties avec une loi uniforme. On pourrait statistiquement chercher la position de l'élément, et faire beaucoup moins de boucles. Pour donner un exemple, une collision vient d'être trouvé pour le hash 94732686660748 à la position 5644256. Proportionnellement, en considérant que les éléments sont répartis de façon homogène, $94732686660748/2^24 \approx 5646508$, ce qui donne 0.04% d'erreur sur la position du hash dans le tableau. Puis on continue l'algorithme. On peut donc utiliser l'algorithme dans ce papier pour aller plus vite : http://www.cs.technion.ac.il/~itai/publications/Algorithms/p550-perl.pdf
- Paralléliser: On pourrait facilement paralléliser, car les essaies sont indépendants. À noter que la fonction `find` de dichotomie est celle qui prend le plus de temps.
  À noter aussi que pour faire la moyenne, on fait :

  ```
  index = left_index - ((left_index - right_index) >> 1)
  ```

  Pour éviter de faire un overflow (même si normalement ça ne devrait pas arriver).

Question 2:

De la même façon, on effectue un semblant de "man-in-the-middle" pour la question 2.

- Pour trouver un bloc cm qui collisionne avec un hash intermédiaire h_i quelconque, on stocke chaque bloc de message avec son hash et son index que l'on trie en $O(n\ log(n))$.
- Ensuite, on génère un bloc aléatoire et par recherche dichotomique, continue jusqu'à coïncider avec un hash quelconque de la chaine.\
- Une fois ceci fait, on reconstruit le message expansible afin d'avoir la même longueur que le message original et que le padding soit le même.

Sur une moyenne de 20 attaques, cette technique met 769.98 secondes soit environ 12 minutes.