Projet Logique - Partie SAT

http://www.lsv.ens-cachan.fr/~hirschi/progLogique

Cryptanalyse logique

La première partie du projet logique s'intéresse aux SAT solveurs. Nous allons utiliser cet outil pour casser la fonction de hachage MD5. Une fonction de hachage transforme de façon déterministe une chaîne de bits arbitrairement longue en une chaîne de bits de taille fixée (*i.e.,* 128 bits pour MD5) que l'on nomme *digest* avec comme propriété attendue qu'il est pratiquement impossible de "revenir en arrière" (*i.e.,* trouver une pré-image ou trouver une collision à savoir deux messages qui ont le mêmes digest).

Comme nous le suggère l'article [3], nous allons encoder le problème de trouver une pré-image à un digest donné en une formule de la logique propositionnelle et appeler un SAT solveur pour nous trouver une solution (et donc une pré-image) *cassant* de ce fait MD5. Cette technique se nomme *cryptanalyse logique*.

À rendre pour le **11 mars**.

Table des matières

1 Définition du problème **2**
1.1 SAT-solveurs . . . 2
1.2 Fonctions de hachage MD5 . . . 2
1.3 Cryptanalyse logique . . . 3

2 Votre mission **4**
2.1 Objectifs . . . 4
2.2 Contraintes . . . 5
2.3 Evaluation . . . 6

3 Comment démarrer ? **6**
3.1 Formules . . . 6
3.2 Inverser WeakHash . . . 7
3.3 Conseils généraux . . . 7

1 Définition du problème

1.1 SAT-solveurs

Le problème SAT (venant de l'anglais *boolean SATisfiability problem*) consiste à décider si une formule de la *logique propositionnelle* est satisfiable ou non (*i.e.,* si il existe une assignation des variables propositionnelles vers les booléens qui rend la formule vraie). Ce problème est NP-complet. Il l'est également pour la classe des formules en *forme normale conjonctive*.

Définition 1. *Soit $\mathcal{V}$ un ensemble infini dénombrable de variables que l'on appelle* variables propositionelles*. L'ensemble $\mathcal{P}_0$ des formules du calcul propositionnel est le plus petit ensemble tel que :*

- *les constantes $\top$, $\bot$ ainsi que les variables de $\mathcal{V}$ sont dans $\mathcal{P}_0$;*
- *si $P_1, P_2 \in \mathcal{P}_0$ alors $\neg P_1$, $P_1 \vee P_2$ et $P_1 \wedge P_2$ sont dans $\mathcal{P}_0$.*

Définition 2. *Soit $P \in \mathcal{P}_0$, on dit que P est satisfiable si il existe une valuation $\rho : \mathcal{V} \mapsto \{\top, \bot\}$ tel que $P\rho$ est* logiquement vraie [1].

Définition 3. *Un* littéral *est une formule de $\mathcal{P}_0$ de la forme v ou $\neg v$ pour $v \in \mathcal{V}$. Les* clauses *sont inductivement définies par les deux règles suivantes :*

- *les littéraux sont des clauses ;*
- *si C_1, C_2 sont des clauses alors $C_1 \vee C_2$ en est une.*

Les formules en forme normale conjonctive *(CNF) sont inductivement définies par les deux règles suivantes :*

- *les clauses sont en forme CNF ;*
- *si F_1, F_2 sont en forme CNF alors $F_1 \wedge F_2$ l'est également.*

Comme nous l'avons dit, tous les algorithmes décidant CNF-SAT (*i.e.,* SAT restreint aux formules en forme CNF) ont une complexité exponentielle au pire cas. Cependant, de nombreux SAT-solveurs très efficaces *en pratique* ont été développés. Nous allons utiliser l'un d'eux nommé minisat.

1.2 Fonctions de hachage MD5

Les fonctions de hachages jouent un rôle central en cryptographie. On les utilise notamment pour authentifier et signer des messages. Par exemple, il est plus facile (*i.e.,* moins couteux en temps et bande passante) de signer le digest associé à un contrat de 500 pages que le contrat lui même.

De façon générale, une fonction de hachage est une fonction déterministe $h : \mathcal{B}^* \mapsto \mathcal{B}^k$ (où $\mathcal{B} = \{0, 1\}$) pour un $k \in \mathcal{N}$ fixé. h transforme donc n'importe quel

1. Je vous laisse définir *"logiquement vraie"*.

bitstring (*i.e.,* élément de $\mathcal{B}^*$) de longueur arbitraire en un bitstring de taille fixée. Par un argument de cardinalité, on remarque qu'une fonction de hachage ne peut être injective. Cependant, les propriétés que l'on demande pour une fonction de hachage comprennent :

— *résistance de la pré-image* : il est calculatoirement impossible (*i.e.,* le problème est EXPTIME-hard) de calculer x en connaissant $h(x)$;
— *résistance de la seconde pré-image* : il est calculatoirement impossible de calculer x' en connaissant x tel que $h(x) = h(x')$.

En effet, en reprenant notre exemple du contrat c dont le digest est $h(c)$ et notre signature $\text{sign}(h(c))$, on ne voudrait pas qu'il soit possible qu'un des signataires change le contrat en c' tel que $h(c) = h(c')$ et qu'il nous attaque en prouvant que nous avons signé c' (en effet $\text{sign}(h(c)) = \text{sign}(h(c'))$).

MD5 Dans ce projet, nous allons plus précisemment nous intéresser à la fonction de hachage MD5 que l'on notera md. Elle est maintenant considérée comme cryptographiquement cassée mais reste (malheuresment [2]) largement utilisée. Dans le cas de MD5, la taille du digest est de 128 bits ($k = 128$). Je vous laisse découvrir la primitive lors de votre lecture de [3] ainsi que sur wikipedia. Vous remarquerez que cette primitive est constituée de 64 étapes réparties en 4 tours de 16 étapes chacun.

Sachez cependant que pour simplifier le problème, nous considérons uniquement des messages de 512 bits (*i.e.,* $\mathsf{md} : \mathcal{B}^{512} \mapsto \mathcal{B}^{128}$) et n'utilisons pas de *padding*. La primitive que l'on considère dans ce projet ne travaille donc qu'avec un chunk de 512 bits et n'ajoute ni le bit 1 final ni la taille du message à la fin.

1.3 Cryptanalyse logique

La *cryptanalyse logique* consiste à mesurer la sécurité d'une primitive cryptographique pour montrer qu'une garantie de sécurité attendue n'est pas vérifiée. Par exemple en mettant au point un algorithme trouvant des pré-images en temps sous-exponentiel.

La *cryptanalyse logique* se réfère à une technique qui consiste à casser une primitive en réduisant le problème sensé être EXPTIME-hard en une formule de $\mathcal{P}_0$ dont on teste ensuite la satisfaisabilité en utilisant un SAT-solveur. Dans le cas positif, on doit ensuite pouvoir extraire de la valuation une solution au problème.

Par exemple — et ça sera l'objectif de ce projet — on peut encoder le fait que x (inconnu) vérifie $\mathsf{md}(x) = d$ pour un $d \in \mathcal{B}^{128}$ fixé en une formule en forme CNF. Cette formule est nécessairement satisfiable et si un SAT-solveur nous donne une

2. La dernière attaque en date dû à l'utilisation de MD5 a été découverte pas loin d'ici : `http://www.mitls.org/pages/attacks/SLOTH`.

valuation possible alors on peut en extraire un m tel que $\mathsf{md}(m) = d$. Si on peut mener à bien cette tâche alors on aura casser (nous même) MD5 en mettant en place une attaque sur la propriété de la *résistance de la pré-image*.

Sachez enfin qu'une telle attaque sur MD5 pour un nombre d'étapes inférieur à 64 (par exemple pour 16 étapes) est tout de même significative : souvent les attaques sont incrémentales et les cryptographes s'inquiètent bien avant que l'on casse la primitive dans son intégralité.

2 Votre mission

En quelques mots, ce projet consiste à **mener une cryptanalyse logique de MD5**. Pour cela, vous vous appuierez sur l'article de recherche [3]. Vous y apprendrez les idées essentielles de la traduction vers $\mathcal{P}_0$. Vous pourrez aussi vous aider d'une structure de code OCaml [1] contenant de nombreuses fonctions déjà écrites.

2.1 Objectifs

Objectif 1 Vous devez écrire un programme qui quand on lui donne : (i) un digest (*i.e.,* un élément de $\mathcal{B}^{128}$) ainsi (ii) qu'un nombre de tours et d'étapes, nous renvoie une formule CNF codant le problème de la pré-image pour MD5 configuré avec ce nombre de tours et d'étapes ainsi qu'une de ses solutions.

Objectif 2 On considère ici un scénario précis issu d'un scénario typique que l'on rencontre en crypographie et dans nos navigateurs internet [3]. Mais nous l'avons ici simplifié pous ne pas rentrer dans les détails des primitives cryptographiques.

Lors d'une session d'un protocole X entre le navigateur de Bob et sa banque (que l'on appelera Alice par tradition), Alice envoie *en clair* (*i.e.,* non protégé) à Bob le texte s suivant :

```
Send your PIN at b@bank.com
```

Le texte est en clair car ce message n'a pas vocation a être gardé secret et parce qu'Alice envoie simultanément le digest de ce texte (*i.e.,* $\mathsf{md}(s)$) de façon protégé [4] (c'est à dire que l'on est sûr que personne ne peut l'intercepter ni le modifier sans que Bob ne s'en rende compte). Par contre, le texte envoyé en clair est non protégé et donc, notamment parce que Bob est connecté à un hot-spot Wifi, n'importe qui possédant un ordinateur avec Wifi peut remplacer ce message par un autre s'.

3. Par exemple la très récente attaque SLOTH (cf. `http://www.mitls.org/pages/attacks/SLOTH`) est de ce type.

4. Les détails du "comment" ne nous intéresse pas pour ce projet.

Cependant, Bob s'en apercevra car il observera que $h(s) \neq h(s')$. Votre objectif est de trouver un message s_{fin} tel que le message s' débutant par :

```
Send your PIN at b@hack.me
```

suivi de deux retours à la ligne puis de s_{fin} vérifie [5] :

1. $s' \in \mathcal{B}^{512}$;
2. $\mathsf{md}(s') = \mathsf{md}(s)$.

2.2 Contraintes

Langage Le code "bootstrap" que je vous donne ici [1] est écrit en OCaml il vous aidera énormément pour toutes les tâches pas très intéressantes comme le parsing et l'affichage de bitstrings, conversions ascii, hexadecimal, etc. Il vous donnera également une structure que je vous invite à suivre.

Cependant, vous avez le choix du language de programmation tant que ce choix est raisonnable. Je vous demande donc de m'en parlez avant de commencer (si vous ne choisissez pas OCaml).

Usage du programme L'exécutable de votre programme doit respecter des conventions. Heureusment je les ai déjà mises en place dans l'archive bootstrap. Pour ceux qui n'écriront pas d'OCaml, l'exécutable `hackMD` doit pouvoir être appelé comme ceci :

```
./hackMD -r 1 -s 16 -p 224 exemples/malicious-partial-input.hex exemples/honest-digest.hex
./hackMD -r 1 -s 16 exemples/0-digest.hex
```

où les fichiers dans `exemples` sont des bitstrings de la longueur appropriée codés en hexadecimal (regroupé par mots 32 bits). Allez voir ces fichiers et faites `./hackMD -h` dans l'archive boostrap pour comprendre le sens des options et vous assurez vous que votre exécutable peut être appelé comme celà.

Le programme peut afficher ce qu'il veut mais il **doit** créer les fichiers suivant :

1. un fichier contenant la formule CNF correspondant au problème encodé en format dimacs ;
2. un fichier contenant une pré-image trouvée (elle doit correspondre à la première réponse de minisat sur la formule du premier fichier).

Une description du format dimacs peut par exemple se trouver ici [2]. Le nommage des fichiers créés doit respecter les conventions fixées par le code du bootstrap.

5. Si vous aviez été capable de trouver un tel message en moins de 2 secondes en de pareille circonstance, vous auriez pu ... embêter Bob.

2.3 Evaluation

Au terme de ce projet, vous m'enverrez votre code ainsi qu'un rapport d'une à trois pages. Le rapport décrira votre démarche, les problèmes rencontrés et comment vous les avez surmontés. Vous pourez également discuter vos choix algorithmique et d'implémentations. Votre code devra compiler et pouvoir être utilisé comme décrit en section 2.2 au moins avec certains paramètres (nombre de tours et d'étapes). Le code doit être suffisament (docu)commenté pour que je puisse comprendre votre algorithme.

J'évaluerai alors principalement deux choses. Premièrement, je testerai les performances de votre solution : remplissez-vous les objectifs pour au moins un nombre d'étapes très faible, jusqu'à quelle nombres de tours et d'étapes pouvez-vous aller, quelle temps de calcul a besoin minisat pour résoudre les formules que vous générez, *etc.* Deuxièment, j'évaluerai la clareté et la pertinence du rapport ainsi que du code.

3 Comment démarrer ?

Les objectifs que je vous ai donné vous paraissent peut-être inatteignables mais on va commencer doucement et découvrir que le bootstrap va nous faire tout le sale boulot. Ensuite, la lecture de [3] va beaucoup vous apprendre sur la traduction que vous devrez menez. Je vous demande de lire ce papier avant la seconde séance et de profiter de cette première séance pour découvrir le bootstrap et y coder ce que je décrit en Section 3.1. Commencer avant tout par lire les fichiers du bootstrap en commençant par le `README` puis `main.ml` puis les fichiers `.mli`.

3.1 Formules

Une première étape dans votre projet pourrait s'intéresser aux manipulations de formules. Si vous écrivez maintenant une fonction de traduction de formules de $\mathcal{P}_0$ (et pourquoi pas enrichies de nouveaux connecteurs) vers des formules en forme CNF alors par la suite vous pourrez vous autoriser à utiliser les formules sans contrainte. Ecrivez donc le code la fonction `Formula.formulaeToCnf` (`Formula.simple` peut vous y aider).

Plus tard, il y a de grandes chances que vous ayez besoin d'instancier certaines variables par certaines valeurs dans une formule. La fonction `Formula.susb` fait celà, vous pourriez écrire son code.

Les formules en forme CNF que vous allez générer devront être affichées au format `dimacs`. Apprenez en plus sur minisat et ce format en lisant [2] puis écrivez le code de `Formula.displayCnf`.

Vous pouvez maintenant tester votre implémentation du module `Formula`. Ecrivez vos tests dans la fonction `Formula.test` et lancez vos batteries de tests en tapant `./hackMD -t`.

3.2 Inverser WeakHash

Vous avez maintenant une bonne base pour poursuivre le projet. Avant de se lancer directement dans l'inversion de MD5, il pourrait être judicieux de commencer par plus facile.

Ce que je vous propose ici consiste à remplir le premier objectif pour une fonction de hachage beaucoup plus simple que MD5 (mais qui n'existe pas à part ici). Cette fonction pourrait s'écrire comme suit en pseudo code (pour une entrée $m \in \mathcal{B}^{512}$) :

```
d = nouveau tableau de taille 128 rempli de 0
pour i de 0 à 10 faire:
  d[i] = (d[i*42 + i mod 10] xor m[i*13 mod 512]
          xor m[i*14+1 mod 512] xor m[i*15+2 mod 512])
fin pour
renvoyer le digest d
```

Seules les 10 premiers bits du digest vont changer.

Commencer par ce problème plus simple va vous permettre de vous concentrer sur certaines problèmes en évacuant la complexité de MD5.

3.3 Conseils généraux

Il faut anticiper et réfléchir aux façons que vous aurez de tester votre code. Dites-vous que si votre code qui génère des formules et buggé alors il vous sera difficile de déceler le problème dans la formule qu'il génère si elle code le problème pour MD5 tout entier. Pensez-y.

Je vous conseille fortement de commencer par écrire l'algorithme qui n'inverse pas mais qui calcule normalement (*i.e.,* la fonction `Md.compute` dans bootstrap) en gardant en tête que vous pourriez utiliser ce code comme point de départ pour coder la fonction qui inverse (*i.e.,* la fonction `Generate.genCNF`). Premièrement ça vous permettra de débugger votre code plus facilement par la suite (pensez au premier conseil) et deuxièment ça vous permettra de tester votre inversion très facilement et rapidement (tout est déjà en place dans le bootstrap).

Bon courage et amusez-vous bien ;)

Références

[1] Archive bootstrap : `http://www.lsv.ens-cachan.fr/~hirschi/enseignements/progLogique/SAT/bootstrap.tar.gz`.

[2] Tutoriel sur minisat : `http://www.dwheeler.com/essays/minisat-user-guide.html`.

[3] Gilles Dequen, Michaël Krajecki, and Florian Legendre. Inverting thanks to sat solving-an application on reduced-step md*. In *SECRYPT*, pages 339–344, 2012.