Rapport du projet de logique

Julien Rixte

11 mars 2016

1 Introduction

L'objectif de ce projet était de calculer l'antécédent par la fonction de hachage MD5 d'un digest quelconque. Cet antécédent devait également satisfaire certaines contraintes. Dans ce rapport, je décrirai dans un premier temps la démarche que j'ai suivie. Puis, j'expliquerai la manière dont mon code est structuré. Enfin, je proposerai des améliorations possibles pour l'implémentation que j'ai écrite.

2 Démarche

Comme suggéré dans le sujet du projet, j'ai commencé par écrire les fonctions de manipulation de formules logiques. La principale fonction était la conversion d'une formule quelconque en formule en forme normale conjonctive. Bien que cette partie ne présente pas de réelle difficulté, il était fondamental de la tester correctement car je me suis rendu compte par la suite qu'il était difficile de déboguer le code lors de l'inversion de hachage.

Mes tests étaient d'ailleurs insuffisants car en inversant WeakHash, je me suis rendu compte, après une longue phase de débogage que la fonction de conversion n'était pas correcte.

Le temps que j'ai perdu sur cet erreur n'aura pas été totalement inutile car je me suis forcé par la suite à tester toutes les fonctions que j'écrivais, en particulier au moment de l'inversion de MD5. Le fait d'avoir à effectuer ces tests m'a d'ailleurs obligé à écrire des fonctions calculant des sous formules sur des variables quelconques (et non uniquement sur les variables pour lesquelles elles sont destinées), ce qui rend ces fonctions bien plus modulaires.

3 Structure du code

3.1 Organisation des variables

Le premier problème qui se pose est la manière d'organiser les variables de la formule que l'on souhaite donner à minisat. Comme le bootstrap le suggère, j'ai représenté les variables par un entier.

Tout d'abord, les variables sont organisées en groupes de 32 bits. Un groupe de 32 bits est représenté par l'entier n correspondant à la première variable du groupe. Ainsi, $n + i$ donne la i^{eme} variable du groupe représenté par n.

Ensuite, ces groupes (à l'exception des variables de l'input et des quatre additions finales de MD5) sont organisés en steps, chaque step représentant les variables nécessaires à l'inversion de ce step.

Input (16 blocs)	a_0,c_0,d_0	Dernières additions (8 blocs)	$Step_0$ (6 blocs)	$Step_1$	$Step_2$	...	$Step_k$

Chaque step s est composé de 6 blocs :

— b_s contient les variables correspondant aux calculs successifs de b dans la boucle principale de md5. On peut aisément connaître a_s, c_s et d_s en remarquant que, en posant $b_{-1} = c_0$, $b_{-2} = d_0$ et $b_{-3} = a_0$, on a $a_s = b_{s-3}$, $c_s = b_{s-1}$ et $d_s = b_{s-3}$.

— non_lin_s est le résultat de la fonction non linéaire
— $carry41_s, carry42_s$ sont les deux retenues nécessaires à l'addition de quatre blocs.
— $sum4$ est le résultat de l'addition des quatre blocs
— $carry\_lr$ est la retenue pour l'opération d'addition-rotation. Il est à noter que l'on a pas besoin de variables représentant le résultat de l'addition-rotation car celui-ci sera stocké dans b_{s+1}

b	non_lin	$carry41$	$carry42$	$sum4$	$carry\_lr$

3.2 Découpage des formules

J'ai découpé la formule permettant de casser en trois parties :
— initialisation
— steps
— additions finales

L'initialisation consistait à l'origine à forcer la valeur de certaines variables, notamment les variables correspondant au digest, les quatre blocs a_0, b_0, c_0 et d_0 ainsi que les informations partielles sur l'input. Cependant, pour accélérer le calcul de minisat, le mieux est d'utiliser la substitiution afin de diminuer le nombre de clauses et de variables. Je me suis donc contenter d'initialiser l'information partielle sur l'input car la substitution ne suffit pas : on doit dire à minisat que certaines variables de l'input sont déjà connues.

Chaque step s est la conjonction des formules suivantes :
— Fonction non linéaire : cette formule permet d'inverser la fonction non linéaire correspondant au step qu'on est en train d'inverser. Le résultat est stocké dans non_lin_s.
— Addition de 4 blocs : permet d'additionner a_s, le résultat de la fonction non linéaire, la composante s du vecteur k et le bloc de l'input correspondant au step que l'on souhaite inverser. On aurait pu ici combiner simplement des additions de blocs deux à deux mais le nombre de variables aurait alors significativement augmenté : on aurait eu besoin de trois additions et donc de trois blocs pour les retenues et trois blocs pour les résultats. Ici, on divise donc par deux le nombre de variables utilisées en se contentant de deux blocs de retenue et d'un bloc de résultat. Le résultat de l'addition est stocké dans $sum4_s$.
— Addition-rotation : permet de combiner la rotation du résultat de l'addition de 4 blocs et l'addition de cette rotation à b_s. Le résultat est stocké dans b_{s+1}. Ici, on évite d'utiliser des variables intermédiaires pour stocker le résultat de la rotation car on peut directement faire la rotation au moment de l'addition. On remarque que cette fonction est également utilisée au moment de l'addition finale avec une valeur de rotation nulle.

3.3 Implémentation des formules

Chacune des formules énumérées ci-dessus est calculée par une fonction indépendante qui prend en paramètre les blocs considérés. L'affectation et la fonction non linéaire ne posent pas de difficultés : l'affectation est une conjonction de littéraux et la fonction non linéaire est déjà écrite sous forme logique dans MD5. Détaillons plutôt le calcul de la formule de l'addition de 4 blocs.

Pour chaque variable du bloc de résultat, j'ai besoin d'une conjonction de 32 formules. En effet, selon la valeur des trois termes additionnés et des deux retenues, la sommes ou les retenues seront différentes. Ainsi, pour chacune des distributions possibles pour les 5 variables sus-citées, on donne sous la forme d'une implication la valeur de vérité des retenues suivantes et de la somme.

4 Bilan et améliorations

Mon implémentation permet de casser honest-digest dans les instances indiquées dans le tableau ci-dessous. Les temps d'exécution, qui sont ici à titre indicatif, correspondent aux temps d'exécution sur mon ordinateur personnel (on aurait presque pu diviser les temps d'exécution par deux en utilisant

un ordinateur de la salle 411). Ils sont indiqués en secondes et sont écris sous la forme "temps de génération de la formule + temps de résolution par minisat".

Nombre de rounds	Nombre de steps	Temps (sans info)	Temps (avec info)
1	16	1,99 + 0.15	2,61 + 0,03
2	12	2,72 + 3,48	3,54 + 0,97
3	8	2,83 + 6,71	3,68 + 16,87
4	5	2,42 + 1,88	X

Pour améliorer le code, il serait possible d'une part de diminuer le nombre de formules pour encoder le problème de l'addition mais aussi d'implémenter les améliorations que Florian Legendre, Gilles Dequen et Michaël Krajecki dans l'article *Inverting thanks to a SAT-Solver*

Les performances que j'ai obtenues sont à peine moins bonnes que celles obtenues par Florian Legendre, Gilles Dequen et Michaël Krajecki sans l'amélioration finale : ceux-ci parviennent à inverser 2 rounds 13 steps. Les idées d'amélioration précédentes permettrait donc de me faire gagner environs 3 steps.