Rapport

BLOT Elliott,BARRÉ Léo

fevrier 2017

Table des matières

Introduction

Dans la mesure où la sécurité de nombreux codages est liée à des problèmes de complexité NP (Merkle-Damgard, RSA, etc.), il paraît évident que les SAT-solvers, des programmes capables de déduire des instances du problème SAT, lui-même étant un problème NP-complet, furent envisagés comme outils de cryptanalyse.

L'idée est simple : traduire les chiffrements ou les configurations sous forme de formule que le programme sus-nommé pourra tenter de résoudre, et ainsi obtenir des informations supplémentaires sur la conversation. Mais dans des cas de chiffrement par flots (comme Trivium), la formule est traduite directement depuis la suite chiffrante, elle-même étant une énorme suite d'opérations XOR, mal gérées par les méthodes classiques de résolution de formule des SAT-solvers.

Dans le cadre de ce projet, nous nous sommes intéressés aux papiers du Dr. Soos, le créateur du SAT-solver **CryptoMiniSAT**, qui a implémenté un système interne capable de gérer lesdites opérations et de les traiter plus adéquatement entre elles.

En nous basant sur son travail, ainsi que sur la maquette Python du SAT-solver des Dr. Simon et Dr. Audemard (nommé pysat), nous avons tenté durant ce mois d'implémenter un pré-processeur capable de gérer ces mêmes calculs, afin de simplifier la formule donnée et alléger le travail du programme.

Chapitre 1

Contexte du projet

1.1 Rappel

Avant de rentrer dans le vif du sujet, il est nécessaire de faire quelques rappels.

Nous travaillons sur des logiciels appelés **SAT-solvers**, ainsi nommés car spécialisés dans la résolution d'instances du problème **SAT** sur des **formules propositionnelle**.

Ces dernières sont des formules logiques ayant cette forme :

$$x_1 \vee x_3 \wedge \neg x_4 \vee (x_2 \vee x_1)$$

où les différents x_i sont des variables booléennes.

Le problème SAT peut s'exprimer de cette façon :

ENTRÉE	**:** une formule propositionnelle
PROBLÈME	**:** existe-t-il une valuation des variables telle que la formule soit vraie

Ici, l'exemple choisit plus haut est satisfiable, puisqu'évaluer $x_1 = 1$ est suffisant pour valider la formule (l'opérateur $\wedge$ est prioritaire sur le $\vee$)

En général, les SAT-solvers lisent des formules sous forme **CNF** (Conjunctive Normal Form), qui consiste en une conjonction ($\wedge$) de clauses disjonctives ($\vee$) de littéraux (x_i ou $\neg x_i$).

Par exemple, la formule précédente sous forme CNF :

$$\begin{array}{ll} & (x_1 \vee x_2 \vee x_3) \\ \wedge & (x_1 \vee x_2 \vee \neg x_4) \end{array}$$

1.2 Quelques détails sur les SAT-solvers

Les entrées des SAT-solvers se font via des fichier .cnf, écris sous format **DiMACS**, construits de la manière suivante :

Sur la première ligne se trouve le header, sous la forme : ***p cnf n m***, avec n le nombre de variables et m le nombre de clauses. Chaque variable de la formule sera représentée par un entier entre 1 et n.

Ensuite, des lignes ressemblant à : $\boldsymbol{l_1\ l_2\ \ldots\ 0}$, qui coderont les clauses, avec les différents l_i à remplacer par la variable concernée (avec un moins si c'est un "not") et un "0" à la fin pour marquer la fin de la disjonction.

Par exemple $x_1 \vee x_3 \vee \neg x_4$ sera écrit comme ça : 1 3 -4 0.

Et éventuellement, des lignes de commentaires pourront être ajoutés dans le fichier, en les débutant par la lettre ***c***.

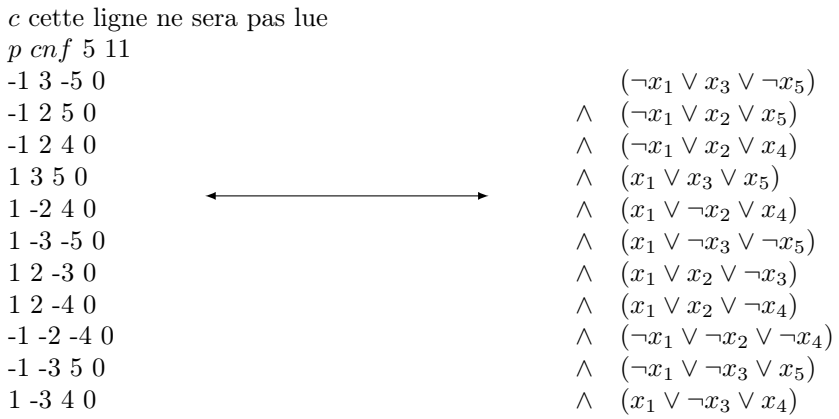

FIGURE 1.1 – un fichier .cnf tel qu'on en croise à chaque coin de rue

Enfin, pour rappeler un peu le contexte de résolution classique des SAT-solvers, ils utilisent généralement une méthode appelée subtilement, la **méhode de résolution**.

Celle-ci est simple : on part d'un ensemble de clauses qu'on va partitionner en trois groupes par rapport à une variable pivot x_p :

— celles ne contenant pas la variable (ex : $x_1 \vee \neg x_2$) $\Rightarrow \boldsymbol{C_0}$

— celles le possédant sous forme de littéral positif (ex : $\neg x_3 \vee x_p$) $\Rightarrow \boldsymbol{C_+}$

— celles le possédant sous forme de littéral négatif (ex : $x_2 \vee \neg x_p$) $\Rightarrow \boldsymbol{C_-}$

puis de construire un nouvel ensemble composé tel quel :

$$C_0 \wedge \{c_+ \backslash \{x_p\} \vee c_- \backslash \{\neg x_p\} \mid c_+ \in C_+,\ c_- \in C_-\}$$

où $c_+\backslash\{x_p\}$ représente une clause de C_+ dépourvue de son littéral x_p (pareil avec c_-).

exemple : $(x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_p) \wedge (x_2 \vee \neg x_p) \ \Rightarrow\ (x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_2)$

On peut voir que la formule originale implique la nouvelle (si il y a une solution pour la première, elle marche aussi pour la seconde), et que si la formule réduite est satisfiable, alors on peut déduire une valuation cohérente pour l'ancienne.

Maintenant, considérons cet exemple-ci :

$$
\begin{array}{ll}
 & (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \\
 & \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \\
\text{résolution par } x_2 \Rightarrow & (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \\
\text{résolution par } x_3 \Rightarrow & \boldsymbol{\neg x_1}
\end{array}
$$

En 2 étapes, on a pu déduire que $x_1 = 0$. Mais, comme on le verra plus bas, on aurait pu atteindre le même résultat en une seule (je sais, c'est pas folichon, mais c'est toujours ça de gagné).

Chapitre 2

Bases de l'implémentation

2.1 Bases de la base

Notre implémentation se base sur le SAT-solver en Python de Laurent Simon, **pysat**.

Celui-ci est composé de plusieurs modules :
— **genRandom** : génère un fichier cnf aléatoire,
— **prettyPrinter** : pour imprimer des fichiers .cnf
— **satheapq** : pour les heap
— **satboundedqueue** : le module de file utilisé
— **satutils** : les fonctions utilitaires générales
— **sattypes** : globalement, le module des clauses
— **pysat** : le programme principal, avec sa propre classe de solver
— **pysatdpll** : idem, mais en parcours en profondeur
(***Note*** *: Le programme en son intégralité est disponible sur* ***bitbucket*** *(lorensi/pysat). Son exécution est peu rapide, voire incomplète, mais il a été fait avant tout pour proposer aux curieux un SAT-solver facile à comprendre*)

Pour construire notre pré-processeur, nous nous sommes surtout inspirés de **satutils.py**, **sattypes.py** et **pysat.py**, ces derniers possédant les modèles qui nous seraient utiles au long de ce projet (à savoir, la classe Clause, le solver et les utilitaires), et nous avons utilisé **genRandom.py** pour générer les premiers tests d'extraction de XOR-clauses.

Notre code est d'ailleurs observable sur le GitHub de Léo : Shinigami-leo/xor_preproc (parce que l'originalité, c'est tellement mainstream !). On pourra d'ailleurs voir les vestiges de pysat sur les fichiers présents, puisque nous avons directement écrit sur des copies de ceux-ci, nous basant en grande partie sur le code déjà présent, à quelques exceptions près.

Chapitre 3

Le monde merveilleux des Xlauses

Maintenant, entrons dans le vif du sujet, et parlons des XOR-clauses, qu'on appellera affectueusement **Xlauses**.

Une Xlause, déjà, c'est ça :

$$x_1 \oplus x_2 \oplus x_3 = 0$$

En gros, il s'agit d'une bête addition des variables dans $\mathbb{F}_2$. Mais l'avantage, c'est qu'elle est au moins 2 fois plus rapide pour retrouver des résultats comme le précédent.

3.1 Trouver les clauses XOR

Dans un SAT-solveur, les contrainte sont généralement de typer CNF, or une Xlause peut être écrite sous forme CNF. On vas donc chercher a retrouver les clauses impliquant une Xlause pour pouvoir ensuite les stocker à part.On s'occupe donc des Xlauses dans un pré-processeur, pour ensuite transmettre les informations obtenues au solveur. Les Xlauses peuvent être retrouvées très simplement depuis des clauses au format CNF. En effet, si on a 2^{n-1} négations différentes pour un ensembles de clauses de même variable de taille n, alors cet ensemble est transformé en une seul clause XOR.

$$\left.\begin{array}{r} x_1 \vee x_2 \vee \neg x_3 = \textbf{true} \\ x_1 \vee \neg x_2 \vee x_3 = \textbf{true} \\ \neg x_1 \vee x_2 \vee x_3 = \textbf{true} \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 = \textbf{true} \end{array}\right\} \Leftrightarrow x_1 \oplus x_2 \oplus x_3 = \textbf{false}$$

FIGURE 3.1 – exemple de Xlause à 3 variables

x_1	x_2	x_3	$x_1 \oplus x_2 \oplus x_3$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$\left.\begin{array}{rl} \Longrightarrow & \neg x_1 \vee \neg x_2 \vee \neg x_3 \\ \Longrightarrow & \neg x_1 \vee x_2 \vee x_3 \\ \Longrightarrow & x_1 \vee \neg x_2 \vee x_3 \\ \Longrightarrow & \neg x_1 \vee x_2 \vee \neg x_3 \end{array}\right\} \Leftrightarrow x_1 \oplus x_2 \oplus x_3 = \textbf{true}$$

FIGURE 3.2 – comment coder une Xlause en cnf

3.2 Trier les clauses

Comme on peut le voir dans notre code, pour simplifier la détection des Xlauses, on a besoin que les clauses soit triées dans l'ordre lexicographique de leurs variables, ainsi, les clauses formant une Xlauses seront à la suite et seront donc plus facilement et rapidement repérable. Pour plus de commodité, on supprimera aussi les doublons de clause.

Par exemple il est plus simple et rapide de trouver des Xlauses dans le cas de droite que celui de gauche

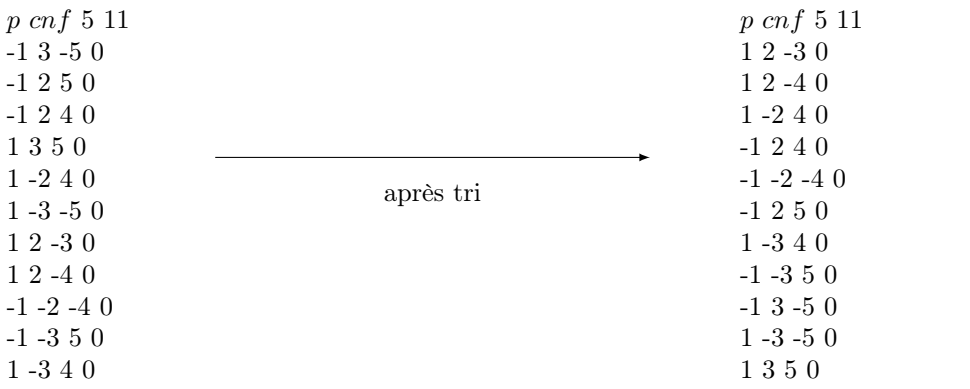

FIGURE 3.3 – exemple de trie d'un fichier .cnf

Sur l'exemple ci dessus on remarque donc qu'il y a deux Xlause :

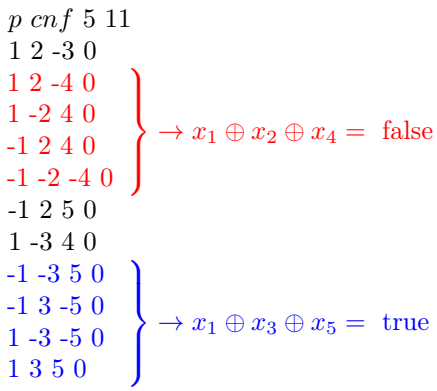

FIGURE 3.4 – localisation de Xlause dans le fichier .cnf

Les clauses utilisé sont ensuite supprimer de la liste des clauses. On obtient donc deux ensemble plus petit qui sont donc :

clauses :

$x_1 \vee x_2 \vee \neg x_3$
$\neg x_1 \vee x_2 \vee x_5$
$x_1 \vee \neg x_3 \vee x_4$

Xlause :

$x_1 \oplus x_2 \oplus x_4$
$x_1 \oplus x_2 \oplus x_5$

Chapitre 4

Simplification

4.1 Élimination de gauss

Une fois que les clauses ont été repérées, on les stocke dans une variable séparé, sous forme d'une matrice. Les lignes de la matrice représentent les clauses, tandis que les colonnes représentent les variables. si une variable i est dans la $jieme$ clause, alors on met un 1 en $ieme$ place de la $jieme$ colonne. la dernière colonne correspond au résultat de la Xlause.

Par exemple :

$$\begin{aligned} x_1 \oplus x_2 \oplus x_3 \oplus x_4 &= \textbf{false} \\ x_1 \oplus x_3 \oplus x_4 &= \textbf{true} \\ x_2 \oplus x_3 &= \textbf{true} \\ x_2 \oplus x_4 \oplus x_5 &= \textbf{true} \\ x_4 \oplus x_5 &= \textbf{false} \end{aligned}$$

FIGURE 4.1 – ensemble de Xlauses

On les stock sous forme matricielle :

$$\left(\begin{array}{ccccc|c} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right)$$

FIGURE 4.2 – matrice associé a l'exemple précèdent

Xorer deux Xlause ne modifie pas la solution, si elle existe, on peut donc manipuler les Xlause pour essayer d'obtenir certaine valeur. les Xlauses étant sous forme de matrice, il est très facile d'utiliser gauss dessus pour essayer de

fixer certaine variable, ou encore de déterminer qu'un problème est unsat très rapidement.

Par exemple sur la matrice précédente :

$$\left(\begin{array}{ccccc|c} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right)$$

$$\left(\begin{array}{ccccc|c} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right)$$

$$\left(\begin{array}{ccccc|c} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right)$$

$$\left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

FIGURE 4.3 – etapes de l'élimination de Gauss

Maintenant on peut voir facilement sur cet exemple la valeur de x_2 doit être fixé à **true** et celle de x_3 doit être fixé à **false**, ce qui accélérera donc la résolution du problème vu qu'il y a deux valeurs de moins à chercher.

De meme on peut facilement voir si un problème est unsat si on a par exemple deux lignes contradictoires, ou encore la ligne :

$$\left(\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

4.2 Séparation de matrice

Étant donnée que l'élimination de Gauss a une complexité polynomiale par rapport à la taille de la matrice traité, il peut être intéressant de séparer la matrice en plusieurs matrices séparées plus petite.

En effet, il arrive que dans la matrice, on puisse former plusieurs groupe de variables, ne pouvant pas influer sur les autre groupes (c'est à dire que toute variable x_i d'un groupe A, n'a aucune Xlauses en commun avec aucune variable en dehors du groupe A).

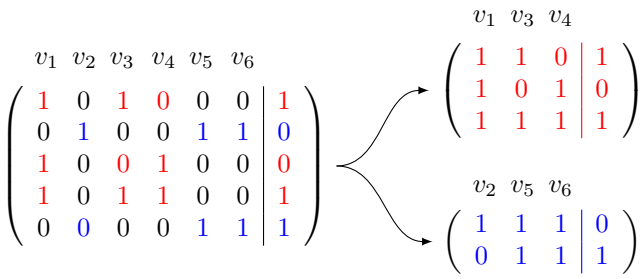

FIGURE 4.4 – exemple d'une matrice pouvant être séparée en deux

Il est ensuite plus rapide d'effectuer les élimination de Gauss sur les petites matrices que sur la grande, et on obtiendra exactement le même résultat.

4.3 Réduction de l'espace

On peut réduire le nombre de lignes et de colonnes de la matrice précédente de différentes manières pour accélérer le processus par la suite. L'une d'elle est la suivante :
Si une variable (par exemple x_1) est présente dans une seule Xlause (par exemple $x_1 \oplus x_2 \oplus x_3$) et dans aucune clause, alors on peut enlever cette Xlause du solveur, et calculer la valeur de x_1 qu'une fois que les autres valeurs de la solution auront été retrouvés.

On peut aussi étendre cette technique pour qu'elle fonctionne si la variable est dans deux Xlauses seulement en Xorant ces deux Xlauses : parce qu'un exemple est beaucoup plus parlant que moi sur mon clavier :

(1) $x_1 \oplus x_{10} \oplus x_{11} =$ **true**
(2) $x_1 \oplus x_{20} \oplus x_{21} =$ **true**
(1) $\oplus$ (2) $= x_{10} \oplus x_{11} \oplus x_{20} \oplus x_{21} =$ **false**

On obtient alors une seul Xlause au lieux de deux, et la variable x_1 ne sera calculé qu'à la fin, quand toutes les autres variables seront calculées.

4.4 Dernière étape

Une fois qu'on à réduit au maximum les Xlauses, on a potentiellement obtenu certaines valeurs de variables , c'est à dire, si on a une ligne de la matrice sous l'une des formes :

$$0 \quad 1 \quad 0 \quad 0 \mid 1$$

ou encore

$$0 \quad 1 \quad 0 \quad 0 \mid 0$$

Dans ce cas, on rajoute la clause x_i ou $\neg x_i$ correspondante pour que le solveur puisse la fixer par la suite (par exemple sur les deux lignes précédentes on voit que $x_2 =$ **true** dans le premier cas et $x_2 =$ **false** dans le deuxième cas).

Cependant il y a de grandes chances qu'elles ne soient pas toutes fixées. Il est alors possible de généraliser ce qu'on vient de voir précédemment, au cas ou il ne reste que deux variables sur une ligne de la matrice, c'est à dire si on a une ligne de la forme :

$$0 \quad 1 \quad 1 \quad 0 \mid 1$$

ou encore

$$0 \quad 1 \quad 1 \quad 0 \mid 0$$

Dans ce cas on sait alors que par exemple $x_2 = x_3$, donc on va essayer de remplacer x_2 dans les clauses par x_3, on aura ainsi une variable qui reviendra plus souvent dans les clauses, et sera donc plus facile à fixer par la suite.

$x_1 \oplus x_3 =$ **false**	$x_2 \oplus x_3 =$ **true**
$x_1 \vee x_5 \Rightarrow x_3 \vee x_5$	$x_2 \vee x_5 \Rightarrow \neg x_3 \vee x_5$

FIGURE 4.5 – exemples de cas où deux variable sont dépendante l'une de l'autre

Conclusion

Pour conclure, le pré-processeur permet de rendre le SAT-solver plus rapide, dans le cas où il y a des XOR dans les clauses. Cela est principalement dû au fait que l'élimination de Gauss nous permet d'obtenir rapidement des informations sur certaines variables, ce qui réduit le nombre de clauses qui reste ensuite à vérifier par le solver.

Même si notre programme reste assez lent (car codé en python, et les algorithmes ne sont sûrement pas optimaux), il permet tout de même une résolution plus rapide d'une partie des benchs que nous avons testé, en utilisant **CryptoMiniSAT** comme corps du solver. Cependant il y a certains cas où la résolution est plus longue, il faudrais donc pouvoir déterminer avec précision si on est dans un cas où le pré-processeur accélérera la résolution ou la ralentira.

Le code produit peut être récupéré sur le github : `https://github.com/Shinigamileo/xor_preproc` .

Name of the protocol	Authentication of A	Authentication of B	Secrecy
AbadiNeedham.spthy	0 :00.67 attack found	0 :00.54 attack found	0 :00.64
WooLam.spthy	0 :00.63 attack found	0 :00.55	0 :00.70

Name of the protocol	Authentication of A	Authentication of B	Secrecy
AbadiNeedham.spthy	0 :00.79 attack found	0 :00.60 attack found	0 :00.55
WooLam.spthy	0 :00.75 attack found	0 :00.69	0 :00.60