Recherche arborescente et apprentissage pour les problèmes cryptographiques

Anthony Blomme, Sami Cherif, Sorina Ionica, Gilles Dequen

Recherche arborescente et apprentissage pour les problèmes cryptographiques

Anthony Blomme **Sami Cherif** **Sorina Ionica** **Gilles Dequen**
Laboratoire MIS UR 4290, Université de Picardie Jules Verne, Amiens, France
{prenom.nom}@u-picardie.fr

Résumé

Les instances provenant du monde de la cryptographie sont représentées sous format ANF (Algebraic Normal Form). Les solveurs dédiés à la resolution de ces instances transforment les formules en une conjonction formée de clauses CNF et XOR et utilisent des techniques comme l'élimination Gaussienne. Dans ce papier, nous souhaitons introduire un solveur capable de raisonner directement sur une formule au format ANF tout en intégrant des mécanismes propres aux solveurs SAT, comme les watched literals et l'analyse de conflit.

Abstract

Instances from the world of cryptography are represented in the ANF (Algebraic Normal Form) format. Solvers dedicated to solving this type of instances translate the input formulae into a conjunction of CNF and XOR clauses and use techniques such as Gaussian elimination. In this paper, we want to build a solver able to directly reason on ANF formulae while integrating mechanisms specific to SAT solvers, such as watched literals and conflict analysis.

1 Introduction

Résoudre le problème de Satisfiabilité (SAT) consiste à déterminer, étant donné une formule en Forme Normale Conjonctive (CNF), s'il existe une affectation des variables qui la satisfait [2]. Au cours des dernières décennies, les solveurs SAT ont énormément gagné en efficacité notamment grâce à des mechanismes puissants qui leur sont dédiés tels que l'analyse de conflit (CDCL) [5] ou encore les watched literals [4]. Ainsi, ces solveurs sont couramment utilisés pour résoudre une variété de problèmes provenant de nombreux domaines académiques et industriels. Cependant, malgré le fait qu'il soit possible de traduire toute formule au format CNF, il existe certains domaines dans lesquels il semble intéressant de faire autrement. Par exemple, les formules obtenues après la modélisation des attaques en cryptanalyse sont représentées au format *ANF* (*Algebraic Normal Form*). Bien qu'il soit évidemment possible de convertir une formule ANF en CNF, les solveurs dédiés à la résolution de ce type de formules [7, 6, 9] sont plus performants soit lorsqu'ils raisonnent directement sur la formule ANF, soit lorsqu'ils n'en convertissent seulement qu'une partie (ce qui donne une formule composée à la fois de clauses CNF et XOR). Ces solveurs exploitent également des techniques algébriques telles que l'élimination Gaussienne.

Dans ce papier, notre objectif est de développer un solveur capable de raisonner directement sur des instances au format ANF tout en exploitant des techniques propres aux solveurs CDCL, comme par exemple les watched literals ou encore l'analyse de conflit. Cet article est organisé de la façon suivante. En section 2, nous rappelons certaines notions fondamentales. Ensuite, nous introduisons le principe et les différents mécanismes de notre solveur en section 3. Enfin, nous présentons nos résultats expérimentaux en section 4 avant de conclure et discuter les perspectives de notre travail en section 5.

2 Préliminaires

Une *variable* booléenne v peut être soit vraie ($\top$) soit fausse ($\bot$). Un *littéral* correspond soit à une variable v soit à sa négation $\neg v$. Un *monôme* est une conjonction (ou une multiplication) de littéraux tandis qu'une *clause* est une disjonction de littéraux. Par exemple, le monôme $x_1 \wedge x_2 \wedge x_3$ peut aussi s'écrire simplement sous la forme $x_1x_2x_3$. On appelle *contrainte XOR* (ou *équation*) une disjonction exclusive (OU exclusif) de monômes. Il est possible d'ajouter une constante $\top$ à une équation et cela correspond à un monôme toujours satisfait. Une formule sous Forme Normale Algébrique

(ou *ANF* pour *Algebraic Normal Form*) est une conjonction d'équations, c'est à dire de contraintes XOR sur des monômes.

Soit V l'ensemble des variables apparaissant dans une formule ϕ. Une affectation $\alpha : V \to \{\top, \bot\}$ est une fonction associant à chaque variable de V une valeur de vérité. Pour qu'un monôme soit satisfait par une affectation, il faut que tous les littéraux apparaissant dans celui-ci soient satisfaits. Pour qu'une contrainte XOR soit satisfaite, nous devons satisfaire un nombre impair de monômes apparaissant dans la contrainte. Finalement, pour satisfaire une formule au format ANF, nous devons satisfaire toutes équations qu'elle contient.

3 Un solveur pour les formules ANF

Dans cette section, nous allons revenir sur les techniques propres aux solveurs SAT modernes et que nous avons adaptées au cadre des formules ANF. Globalement, nous effectuons une recherche arborescente classique similaire à celle de l'algorithme DPLL [2] : nous prenons des décisions suivies de propagations jusqu'à ce qu'une solution soit trouvée. Un conflit est atteint lorsqu'une équation devient falsifiée et, dans ce cas, un retour en arrière (backtrack) est effectué. Nous nous arrêtons au premier modèle trouvé.

3.1 Watched literals

La première technique que nous avons adaptée est l'utilisation des watched literals. Dans notre cas, nous devons les utiliser non seulement au niveau des littéraux présents dans un monôme (nous utilisons alors le terme de *watched literals*) mais également au niveau des monômes apparaissant dans chaque équation (nous utilisons cette fois-ci le terme de *watched monomials*).

Concernant les watched literals, nous pouvons nous contenter de regarder un seul littéral par monôme. En effet, de manière générale, un monôme peut très bien être satisfait ou falsifié et un monôme en lui-même ne peut pas être utilisé pour propager. Nous nous intéressons donc au moment où la valeur d'un monôme peut être déterminée. Si l'un des littéraux présents dans un monôme est falsifié, alors nous pouvons directement en conclure que le monôme considéré est également falsifié. Cependant, nous devons rendre cette information disponible par l'intermédiaire du watched literal. Ainsi, si le watched literal d'un monôme falsifié ne pointe pas déjà sur un littéral falsifié, nous pouvons directement le déplacer sur le littéral nouvellement affecté. Cela suppose que nous puissions savoir quels littéraux apparaissent dans chacun des monômes. Si un littéral regardé est satisfait dans un monôme, alors nous devons lui trouver un remplaçant. Si nous n'en trouvons aucun, alors le monôme est affecté.

FIGURE 1 – Falsification d'un watched literal sur un monôme.

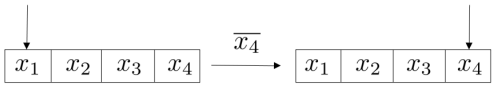
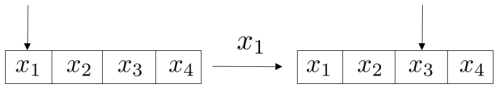


FIGURE 2 – Satisfaction de watched literal sur un monôme. Avant de satisfaire x_1, x_2 a été satisfait sans modifier le watched literal



Exemple 1 *Soit le monôme $x_1x_2x_3x_4$. Nous pouvons considérer par exemple que le watched literal pointe d'abord sur x_1. Si nous falsifions le littéral x_1 alors nous n'avons rien à faire. Par contre, si à la place de x_1, nous falsifions x_4, alors à ce moment-là, nous devons faire pointer le watched literal sur x_4 pour garder l'information que le monôme considéré a été falsifié. Ce comportement est illustré par la figure 1. Reprenons maintenant le cas où le watched literal pointe initialement sur x_1. Si nous commençons par satisfaire x_2, alors nous n'avons aucune opération à réaliser car x_2 n'est pas pointé par le watched literal. Ensuite, si nous satisfaisons x_1, alors nous devons trouver un remplaçant. En parcourant le monôme, nous remarquons que le littéral x_3 est disponible. Nous pouvons donc déplacer le watched literal de x_1 à x_3. Ce comportement est illustré à la figure 2. Si nous satisfaisons ensuite successivement x_4 et x_3, alors le monôme est satisfait car nous ne trouvons plus de remplaçant.*

Concernant maintenant les watched monomials, nous allons devoir cette fois-ci regarder deux monômes dans chaque équation. Dès qu'un monôme regardé est affecté, nous devons lui trouver un remplaçant. Si nous trouvons un monôme qui n'est pas encore affecté et qui n'est pas regardé, nous pouvons déplacer le watched monomial et nous arrêter. Par contre, si nous n'avons pas trouvé de remplaçant, nous devons prendre en compte l'autre watched monomial. Si ce dernier est déjà affecté, alors nous vérifions juste que nous avons bien satisfait un nombre impair de monômes dans l'équation considérée. S'il n'est pas encore affecté, alors nous pouvons en déduire sa valeur en fonction du nombre de monômes satisfaits, en prenant soin de prendre en compte la constante. Si ce nombre est pair, alors nous devons satisfaire le second watched monomial et donc satisfaire tous les littéraux qu'il comprend. Dans le cas contraire, nous devons le falsifier. Dans ce dernier cas, nous ne pouvons propager un littéral à faux que si le monôme ne contient qu'un seul littéral non affecté.

Exemple 2 *Soit le système d'équations suivant :*

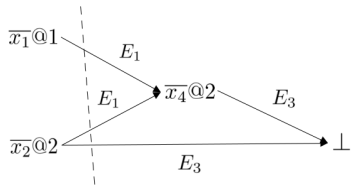
$E_1 : x_1x_2 \oplus x_2x_3 \oplus x_4 \oplus \top$
$E_2 : x_1x_3 \oplus x_2x_4 \oplus x_2x_3 \oplus \top$
$E_3 : x_2x_3 \oplus x_2x_4 \oplus x_3x_4 \oplus x_2$
$E_4 : x_1x_2 \oplus x_3x_4 \oplus x_2 \oplus \top$
$E_5 : x_1x_3 \oplus x_2x_3 \oplus x_1 \oplus x_2 \oplus \top$

Si on commence par prendre la décision $\overline{x_1}$, alors nous falsifions les monômes x_1x_2 et x_1x_3 qui sont regardés dans E_1, E_2, E_4 et E_5. Il faut donc mettre à jour les watched monomials sur ces quatre équations. Par exemple, sur E_1, nous pouvons remplacer x_1x_2 par x_4. Dans E_5, nous pouvons remplacer x_1x_3 par x_2 (x_1 n'est pas disponible car falsifié) etc. Ensuite, nous pouvons prendre une nouvelle décision. Nous choisissons par exemple $\overline{x_2}$. Cette décision falsifie également les monômes x_2x_3 et x_2x_4. Ici, nous devons faire des mises à jour sur toutes les équations. Pour E_1, nous n'allons pas trouver de remplaçant et nous allons propager $\overline{x_4}$. Pour E_2, nous n'allons pas non plus trouver de remplaçant et tous les monômes sont falsifiés. Cependant, la constante $\top$ va nous permettre de satisfaire l'équation. Concernant E_3, tous les monômes sont également falsifiés, ce qui contredit cette équation. Un conflit est donc détecté sur l'équation E_3.

3.2 Apprentissage

Concernant maintenant l'apprentissage, nous allons apprendre de nouvelles équations en effectuant des coupes dans le graphe d'implication [5] du solveur. Pour ce faire, pour chaque littéral propagé, nous devons enregistrer l'équation qui l'a propagé. Dès qu'il n'est plus possible de satisfaire l'une des équations de la formule, nous pouvons lancer une procédure d'analyse de conflit. Nous commençons par identifier les littéraux responsables de ce conflit. Nous pouvons effectuer cela en considérant chaque monôme du conflit. Si le monôme est satisfait, alors nous devons considérer tous les littéraux présents dans le monôme. Si le monôme est falsifié, nous n'avons besoin de considérer que le littéral regardé dans ce monôme. Ensuite, nous pouvons remonter dans les affectations du solveur et considérer les raisons des propagations rencontrées. Si le littéral propagé est présent dans les littéraux que nous avons enregistrés jusqu'à maintenant, alors nous devons considérer la raison associée et remplacer le littéral propagé par ceux qui ont contribué à sa propagation. Nous allons avoir un comportement similaire à ce qui a été présenté pour le conflit, sauf pour le monôme d'où est venue la propagation. Si ce monôme est satisfait, nous pouvons l'ignorer (aucun littéral présent n'a participé à la propagation) et s'il est falsifié, alors nous devons prendre en compte tous les littéraux présents satisfaits.

FIGURE 3 – Exemple de graphe d'implication. La ligne en pointillé représente une coupe possible.



Nous pouvons arrêter cette procédure lorsque nous retombons sur la dernière décision ou lorsqu'il ne reste plus qu'un seul littéral du niveau de décision courant. Pour l'instant, nous continuons d'utiliser le schéma 1-UIP [2] mais nous ne savons pas encore si cela est pertinent dans notre cadre.

Suite à cette analyse de conflit, nous obtenons une nouvelle contrainte à apprendre. Cependant, celle-ci a une forme de clause et il nous faut alors la traduire sous forme d'équation. Soit $(x_1 \vee \ldots \vee x_n)$ une nouvelle contrainte à apprendre. Elle est équivalente à $\overline{(\overline{x_1} \wedge \ldots \wedge \overline{x_n})}$, ce qui revient à apprendre l'équation $\overline{x_1} \ldots \overline{x_n} \oplus \top$. Lorsque nous créons cette nouvelle équation, nous devons créer un watched literal pour l'unique monôme présent ainsi qu'un unique watched monomial. Nous devons placer le watched literal là où il se trouverait si l'équation apprise avait été présente dès le début de la recherche. Dans notre cas, nous devons donc le placer sur le littéral que nous allons propager après le backjump.

Exemple 3 *Si nous reprenons l'exemple précédent, nous avons trouvé un conflit au niveau de l'équation E_3. Ce dernier est survenu après les décisions $\overline{x_1}$ et $\overline{x_2}$ (qui a d'ailleurs propagé $\overline{x_4}$). Dans l'équation considérée, seuls $\overline{x_2}$ et $\overline{x_4}$ ont participé au conflit. Nous les enregistrons donc et ils sont tous les deux du niveau de décision courant. En revenant en arrière dans les affectations, nous trouvons le littéral $\overline{x_4}$ qui est présent dans les littéraux enregistrés. Nous pouvons donc le supprimer et ajouter les littéraux qui l'ont propagé. La raison de $\overline{x_4}$ est l'équation $x_1x_2 \oplus x_2x_3 \oplus x_4 \oplus \top$. Les affectations ayant propagé $\overline{x_4}$ sont $\overline{x_1}$ et $\overline{x_2}$. À ce niveau, nous avons donc enregistré $\overline{x_1}$ et $\overline{x_2}$. Nous pouvons nous arrêter car seul $\overline{x_2}$ est du niveau de décision courant. La clause à apprendre est donc $x_1 \vee x_2$, qui est équivalente à $\overline{x_1}\,\overline{x_2} \oplus \top$. Au niveau de décision de $\overline{x_1}$, nous devons donc propager x_2. La figure 3 représente le graphe d'implication de notre exemple. La contrainte que nous avons obtenue correspond à la coupe sur la figure.*

4 Résultats expérimentaux

Nous avons implémenté notre approche à partir du solveur Glucose [1, 3]. Nous avons considéré les deux

TABLE 1 – Moyennes des conflits et des temps de résolution en secondes sur des instances Rainbow et ECDLP. Un tiret indique que la limite de 30 minutes a été dépassée pour au moins une instance du benchmark. Les meilleurs temps d'exécution sont surlignés en gras.

			WDSat		WDSat (EG)		CryptoMiniSat		DPLL-ANF		CDCL-ANF	
Instances	#Vars	#Eqs	Conflits	Temps	Conflits	Temps	Conflits	Temps	Conflits	Temps	Conflits	Temps
Rainbow_o18	16	35	20079	0.037	124	**0.018**	2095	0.048	20075	0.026	20007	9.182
Rainbow_o28	26	55	37797885	102.677	39544	**2.060**	5607856	961.161	37798012	35.642	-	-
ECDLP_Xn15	42	42	19422	0.130	13062	0.732	71591	6.530	19420	**0.117**	19312	20.800
ECDLP_Xn19	51	52	116190	1.152	83438	7.922	685367	78.558	116186	**0.958**	-	-

architectures DPLL et CDCL. Pour la première, nous utilisons les watched literals et watched monomials. Pour le CDCL, nous avons ajouté l'apprentissage. Au niveau de l'heuristique, nous décidons négativement les variables dans l'ordre lexicographique. Nous avons testé ces deux approches sur des instances issues d'une attaque sur le schéma de signature Rainbow [1] ainsi que de l'attaque du calcul d'indice pour le problème du logarithme discret sur des courbes elliptiques (ECDLP) [2]. Pour chaque type d'attaque, nous avons considéré deux ensembles de 25 instances chacun, totalisant ainsi 100 instances. Celles-ci sont représentées au format ANF. Les instances de l'attaque sur Rainbow sont toutes satisfiables alors que les instances de l'attaque sur ECDLP peuvent être satisfiables ou insatisfiables. Nous avons imposé un temps limite de 30 minutes pour chaque instance. Nous nous sommes comparés aux solveurs WDSat [8, 9] et CryptoMiniSat [6, 7]. WDSat est également capable de considérer des instances au format ANF. Ce n'est pas le cas de CryptoMiniSat, pour qui nous avons dû traduire les instances utilisées au format XOR-CNF avant exécution.

Nous donnons la moyenne des résultats obtenus dans la table 1. Concernant WDSat, nous avons testé le solveur sans et avec élimination Gaussienne (EG). Pour la version sans élimination, nous pouvons observer que notre DPLL semble effectuer une recherche plutôt similaire en terme de conflits à WDSat. Cependant, on peut remarquer que notre recherche arborescente est plus efficace en terme de temps. Cela semble indiquer que l'utilisation de nos structures dédiées (watched literals et watched monomials) est pertinente. Il est important de noter que l'ajout de l'élimination Gaussienne dans WDSat n'est pas toujours pertinent. En effet, même si son utilisation a permis de réduire le nombre de conflits rencontrés par WDSat, cela ne signifie pas pour autant que cette version du solveur est plus performante. Plus précisément, EG permet de rendre WDSat drastiquement plus rapide sur les instances Rainbow mais elle diminue sa performance sur les instances ECDLP. Pour ces dernières, notre DPLL semble être l'approche la plus efficace. Notre CDCL est plus lent que tous les autres solveurs car nous n'éliminons pas les équations apprises. Au vu du nombre de conflits rencontrés, et donc de nouvelles équations à ajouter, il semble normal qu'à ce stade cela ralentisse l'exécution du solveur.

1. https://github.com/VladaSedlacek/mq-comparison-suite
2. https://github.com/mtrimoska/EC-Index-Calculus-Benchmarks

5 Conclusion

Dans cet article, nous avons présenté des travaux préliminaires visant à introduire un solveur dédié à la résolution d'instances au format ANF. Dans ce solveur, nous avons intégré certaines techniques propres aux solveurs SAT modernes. Si la version DPLL semble plutôt efficace grâce à nos structures dédiées, il reste encore beaucoup de progrès à réaliser quant à la version CDCL. En effet, pour rendre cette dernière efficace, il serait intéressant de capitaliser sur les progrès récents dans la résolution de SAT en adaptant d'autres mécanismes connus, notamment des heuristiques de choix de variables, le nettoyage de la base d'équations apprises ou encore les redémarrages. Il serait également intéressant de voir s'il serait possible d'intégrer l'élimination Gaussienne dans notre nouveau solveur et d'étudier les systèmes de preuves qui régissent notre mécanisme d'apprentissage.

Références

[1] G. AUDEMARD et L. SIMON : On the Glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001 :1–1840001 :25, 2018.

[2] A. BIERE, M. HEULE, H. van MAAREN et T. WALSH, éds. *Handbook of Satisfiability - Second Edition*. IOS Press, 2021.

[3] N. EÉN et N. SÖRENSSON : An extensible sat-solver. *In SAT 2003*, p. 502–518. Springer, 2003.

[4] I. LYNCE et J. MARQUES-SILVA : Efficient data structures for backtrack search SAT solvers. *Ann. Math. Artif. Intell.*, 43(1):137–152, 2005.

[5] J. P. M. SILVA et K. A. SAKALLAH : GRASP - a new search algorithm for satisfiability. *In ICCAD 1996*, p. 220–227. IEEE-CS / ACM, 1996.

[6] M. SOOS : Enhanced Gaussian Elimination in DPLL-based SAT Solvers. *In POS-10.*, p. 2–14. EasyChair, 2010.

[7] M. SOOS, K. NOHL et C. CASTELLUCCIA : Extending SAT solvers to cryptographic problems. *In SAT 2009*, p. 244–257. Springer, 2009.

[8] M. TRIMOSKA, G. DEQUEN et S. IONICA : Logical cryptanalysis with WDSat. *In SAT 2021*, p. 545–561. Springer, 2021.

[9] M. TRIMOSKA, S. IONICA et G. DEQUEN : Parity (XOR) reasoning for the index calculus attack. *In CP2020*, p. 774–790. Springer, 2020.