ANF-Based Satisfiability for Weil-Descent Cryptographic Attacks

Anthony Blomme[1,2], Sami Cherif[1], Sorina Ionica[1], Gilles Dequen[1]
[1] Laboratoire MIS UR 4290, *Université de Picardie Jules Verne*, Amiens, France
[2] Univ. Artois, CNRS, Centre de Recherche en Informatique de Lens (CRIL), F-62300 Lens, France
{firstname.lastname}@u-picardie.fr

***Abstract*— In recent years, SAT solvers have been increasingly used in cryptanalysis, for performing attacks both on symmetric and asymmetric schemes. More specifically, they are employed whenever one needs to compute a valid assignment for a logical formula when running the attack. Most often, these formulae are initially represented in Algebraic Normal Form (ANF). Solvers dedicated to solving such instances translate the input formulae into a conjunction of CNF and XOR clauses and use different techniques such as XOR recovery and manipulation and Gaussian elimination. In this paper, we aim to build a solver able to reason directly on ANF formulae derived from attacks using Weil descent on Semaev polynomials of elliptic curves, while integrating dedicated lazy structures inspired from SAT solvers.**

I. Introduction

Given a propositional logical formula, the Satisfiability Problem (SAT) consists in determining whether there exists an assignment of variables that satisfies it [1]. Over the last decades, algorithms for solving the SAT problem have gained enormously in efficiency thanks to powerful mechanisms, such as lazy data structures [2], [3], clause learning [4], [5] and the use of dedicated branching heuristics [6], [7] among many others. SAT solvers are widely employed to address a range of problems across various academic and industrial fields. Most contemporary solvers are designed to handle logical formulae in Conjunctive Normal Form (CNF). However, this form is not always suitable for certain domains, as it does not align with the natural representation of formulae in those fields. One such domain is logical cryptanalysis, which studies the security of cryptographic schemes, often involving the modelling of attacks as logical formulae which can ultimately reveal the secret key when solved through a dedicated SAT solver. However, such formulae are often naturally represented in Algebraic Normal Form (ANF). Despite the fact that the ANF formula can be transformed into a CNF one, it is well known that solvers dedicated to solving this type of formula generally convert only a part of it, resulting in a XNF formula composed of both OR and XOR clauses [8], [9] because they are more efficient this way.

Our goal is to develop a new solver able to directly reason on ANF formulae while integrating dedicated lazy data structures inspired from SAT solvers, notably watched literals [3]. In the context of SAT solvers, if all but one literal are falsified within a clause, the propagation of this last literal, i.e., assigning it to *True*, is necessary to satisfy the clause. Watched literals thus aim to minimize the traversal of entire clauses during propagation by monitoring two literals such that, upon falsification of one, a replacement is sought. In this paper, we will focus on their integration into a solver dedicated to ANF formulae. More precisely, in our solver, we perform a tree search directly on the ANF formula and we introduce two levels of watches to speed up the propagation process. Our aim is to efficiently tackle cryptographic instances derived from the index calculus attack on elliptic curves which are known to be resilient to algebraic techniques such as Gaussian Elimination [10].

This paper is organized as follows. In Section II, we introduce some fundamental notions and briefly review the family of cryptographic attacks that we aim to tackle. In Section III, we present our solver and we introduce our dedicated data structure mechanisms. Our experimental evaluation is presented in Section IV. Finally, we conclude and discuss future work in Section V.

II. Preliminaries

A. Algebraic Normal Form

Le V be a set of Boolean variables. A *literal* is either a Boolean variable $v \in V$ or its negation $\overline{v}$. Before introducing the Algebraic Normal Form (ANF), we recall that a formula in Conjunctive Normal Form (CNF) is a conjunction of clauses formed by disjunctions of literals. The *XNF* (*XOR-CNF*) format extends CNF formulae by allowing clauses with exclusive disjunctions (XOR) alongside typical CNF clauses. In ANF, we introduce the notion of *monomials*, which are formed by conjunction or multiplication of variables. For example, the monomial $x_1 \wedge x_2 \wedge x_3$ can also be represented under the form $x_1x_2x_3$. An *equation* (or a XOR constraint) is an exclusive disjunction (XOR) of monomials. It is possible to add the constant $\top$ to represent a monomial that is always satisfied. A formula in *Algebraic Normal Form* (*ANF*) is a conjunction of equations. An *assignment* $\alpha : V \to \{\top, \bot\}$ is a function that associates a truth value to the variables in V and can be represented as a set of literals. A monomial is satisfied if all its variables are satisfied. To satisfy an equation, we have to satisfy an odd number of monomials appearing in it. An ANF formula is satisfied if all its equations are satisfied. An assignment that satisfies a given ANF formula is called a *model*.

Example 1: Let ϕ be the following ANF formula:

$$E_1 : x_1x_2 \oplus x_2x_3 \oplus x_4 \oplus \top$$
$$E_2 : x_2x_3 \oplus x_2x_4 \oplus x_3x_4 \oplus x_2$$

It is composed of two equations built from the monomials x_2, x_4, x_1x_2, x_2x_3, x_2x_4, x_3x_4 and $\top$. The assignment $\alpha = \{\overline{x_1}, x_2, \overline{x_3}, \overline{x_4}\}$ is a model of ϕ since it satisfies exactly one monomial in each equation, thus ϕ is satisfiable.

B. Cryptographic Instances

We consider ANF formulae derived from a Weil descent on Semaev's m-th summation polynomial of an elliptic curve. This polynomial, that we denote by S_m, is used to check that the sum of m points on the curve is zero. For elliptic curves defined over a binary field $\mathbb{F}_{2^n}$, the x-coordinate of a point on the curve is an element in $\mathbb{F}_{2^n}$ and is thus given by a binary vector of length n. A point on the elliptic curve can be decomposed into m other points by solving Semaev's $(m+1)$-th summation polynomial [11].The second and the third summation polynomials are defined as follows:

$$S_2(X_1, X_2) = X_1 + X_2, \tag{1}$$
$$S_3(X_1, X_2, X_3) = X_1^2X_2^2 + X_1^2X_3^2 + X_1X_2X_3 + X_2^2X_3^2 + 1.$$

For $m > 3$, the m-th summation polynomial is computed by using the following inductive formula:

$$S_m(X_1, \ldots, X_m) = \tag{2}$$
$$Res_X(S_{m-k}(X_1, \ldots, X_{m-k-1}, X), S_{k+2}(X_{m-k}, \ldots, X_m, X)),$$

where Res_X denotes the resultant of two polynomials with respect to the X variable and $k \in \{1, \ldots, m-3\}$. The zeros of this polynomial will thus give the x-coordinates of points on the elliptic curve as elements in $\mathbb{F}_{2^n}$.

A Weil descent consists in choosing a vector basis of $\mathbb{F}_{q^n}$, denoted by $\{\omega_1, \ldots, \omega_n\}$, and writing $X_i = \sum_{j=1}^{n} X_{ij}\omega_j$, $1 \leq i \leq m$, $1 \leq j \leq n$. We replace the X_i by these expressions in the Semaev equation, and end up with a Boolean multivariate system of polynomials in X_{ij}. Instead of using algebraic solvers to find the solutions in $\mathbb{F}_2$ to this polynomial system, we consider the equivalent ANF formula. When running the index calculus attack on the elliptic curve discrete logarithm problem [10], this process is repeated until approximately 2^l decompositions are found, for some $l < n$. Since not all points on the elliptic curve decompose as desired, most ANF formulae that are processed during this attack are unsatisfiable.

C. Related Work

Logic cryptanalysis using satisfiability-based solvers has become increasingly prominent in recent years due to its broad applicability in both symmetric and asymmetric cryptographic schemes [8], [9], [12]–[15]. In this context, SAT solvers are frequently employed to model cryptographic attacks as logical formulae, where solving these formulae reveals the underlying secret key, effectively breaking the cryptographic scheme. One of the main challenges in this domain is the representation of cryptographic problems, which are often naturally formulated in ANF form, as demonstrated in the previous section. ANF allows a concise and compact representation of multivariate polynomial systems over finite fields, making it a more suitable format for cryptographic problems compared to the more traditional forms. However, state-of-the-art solvers such as WDSat [9], [16] and CryptoMiniSat [8], [17] primarily handle formulae in the traditional CNF or the hybrid XNF form in an attempt to bridge the gap between CNF and the natural ANF representation of cryptographic problems, but this form may introduce inefficiencies due to the need for constant transformations between formats. Furthermore, while solvers like WDSat and CryptoMiniSat can manage certain aspects of ANF via these hybrid representations, they do not inherently reason on ANF formulae directly. This mismatch between the native form of cryptographic problems and the formats handled by modern solvers creates a bottleneck in performance and scalability, particularly for complex cryptographic schemes.

Our work aims to address this limitation by introducing a solver that natively reasons on ANF formulae without requiring conversion into other forms. This approach is especially critical in the context of cryptographic attacks like those derived via Weil descent [10], [18], where preserving the algebraic structure of the problem leads to significant performance improvements. By retaining the compactness and integrity of ANF, we can avoid the overhead associated with transforming ANF to CNF or XNF, which often results in unnecessary duplication and bloated representations of the problem. Indeed, another key feature of solvers like WDSat and CryptoMiniSat is their integration of Gaussian Elimination (GE) as an inprocessing technique for handling XOR clauses during the search process. In these solvers, GE is applied by maintaining a Boolean matrix where each row corresponds to an equation in the formula. The matrix entries track the monomials still present in the system, and GE is performed by applying XOR operations between rows. While this approach can deduce additional variable assignments and identify conflicts, it has several downsides. Specifically, the constant need to maintain and update the Boolean matrix incurs significant computational overhead, especially when the matrix becomes large. This issue is particularly prevalent in cryptographic instances arising via Weil descent on Semaev's 4th polynomial, where the complexity of maintaining this structure outweighs the benefits of performing GE, as showcased in [16]. In light of these challenges, our approach intentionally avoids relying on GE and instead focuses on the integration of dedicated data structures that are tailored to ANF. These structures need to facilitate the propagation process by taking advantage of the natural structure of ANF formulae as explained in the following sections.

III. A Solver Dedicated to ANF Formulae

In this section, we introduce our ANF solver building upon a DPLL (Davis-Putnam-Logemann-Loveland) tree search algorithm [19], [20], which incrementally explores the search space by making decisions (i.e., assigning truth values to variables) and then propagating these assignments throughout the formula. Specifically, in our ANF solver, each decision leads to the propagation of values across the entire system of ANF equations. If a conflict arises, i.e., an equation is falsified under the current assignment, we trigger a backtrack step. During backtracking, we retract some of the variable

TABLE I

PROPAGATION RULES FOR ANF

Equation	Hypothesis	Result
$A \oplus x_1 \dots x_n$	$A = \bot$	$x_1 = \top \dots x_n = \top$
$A \oplus x_1$	$A = \top$	$x_1 = \bot$

assignments, thereby pruning the current search path and exploring alternative assignments. This systematic exploration continues until a valid model is found, or all possible assignments have been examined. The search halts as soon as we encounter the first model that satisfies the ANF formula.

One of the key distinctions of our solver is that it reasons directly on ANF formulae, unlike solvers such as WDSat and CryptoMiniSat, which convert parts of the formula into CNF or XNF. In the following sections, we will delve deeper into the mechanisms employed by our solver, particularly focusing on how we have adapted the concept of watched literals to efficiently perform propagations on ANF. In SAT solvers, watched literals enable efficient clause propagation by monitoring only a small subset of literals in each clause, reducing the need to repeatedly traverse large portions of the formula. For ANF Formulae, we have introduced a similar mechanism that takes into account the unique properties of monomials and equations to propagate assignments more effectively as will be showcased in the following sections.

A. Propagation Rules

First, we detail the propagation rules implemented in our solver. These rules are represented in Table III-A. In our case, we will only use equations to propagate. In an equation, if only one monomial is not assigned yet, we can deduce its value based on the number of satisfied monomials in the equation. If this number is even, then we have to satisfy the monomial and therefore satisfy all the variables it contains. Otherwise, we have to falsify it. In this case, we can propagate a variable to *False* only if the monomial contains exactly one unassigned variable. By doing this, we may not propagate as much as possible. Indeed, if we have to falsify the monomial $x_1x_2x_3$ and if x_1 is satisfied, then we cannot know at the moment whether we have to falsify x_2 or x_3 or both. However, by continuing the search on this branch, we may satisfy for example x_2 and, in this case, we know that we have to falsify x_3. To this end, we would need to remember that this monomial has to be falsified and, for now, we have decided to not take this case into account. In opposition to modern SAT solvers, we do not assign a literal directly when the value to propagate is found. Instead, we have a list of literals to propagate and we assign them in a dedicated function (see Algorithm 4).

B. Watched Literals

First, we need to know when a monomial is assigned. In the general case, a given monomial can be either satisfied or falsified in an equation. Thus, monomials cannot be used to perform propagations. Therefore, we only need to watch a unique variable in each monomial. We first consider the case of a falsified monomial. If a variable present in a monomial

Algorithm 1 FalsifiedMonomial(m, l, idx)

INPUT : monomial m, falsified literal l, index idx of l in m

1: **if** $WL(m)$ is not falsified **then**
2: remove m from the monomials watched by $WL(m)$
3: add m to the list of monomials that are watched by l
4: move the watched literal of m to l (index idx)

m is falsified, then we can directly conclude that m is also falsified. However, this information has to be made available via the watched literal. If the watched literal of a falsified monomial does not already watch a falsified variable, then we can directly move it to the newly assigned variable and update the lists of monomials that are watched by each literal. To do so, we need to know which variables are present in each monomial. We can use a dedicated structure to this end and we can also store the index of these variables in the monomials where they are present. We will refer to this process as $FalsifiedMonomial(m, l, idx)$ where m denotes a monomial, l a falsified literal and idx the index of l in m. Note that, if we have to move the watched literal, then we have to remove the monomial from a list of watched literals. As we do not know where it is, we have to go through the entire list. This process is detailed in Algorithm 1 where $WL(m)$ denotes the watched literal of monomial m. The complexity of this procedure is thus in $O(d)$ where d denotes the number of monomials in the formula. We consider that the watched literal of a monomial is directly accessible, thus changing the position of a watched literal is in $O(1)$.

Example 2: Consider the monomial $x_1x_2x_3x_4$. The watched literal is initially set to watch the variable x_1. Then, if we falsify x_1, no update is performed on the watched literal structure as we already watch a falsified variable. Now, if instead of x_1 we falsify x_4, then we have to move the watched literal from x_1 to x_4 to keep the information that the considered monomial is falsified. This behaviour is illustrated on the left side of Figure 1. After the first falsified variable inside a monomial, the watched literal will no longer move until we backtrack to a prior decision level.

Now, if we satisfy a watched literal present in a given monomial, we have to replace it with a variable that is not

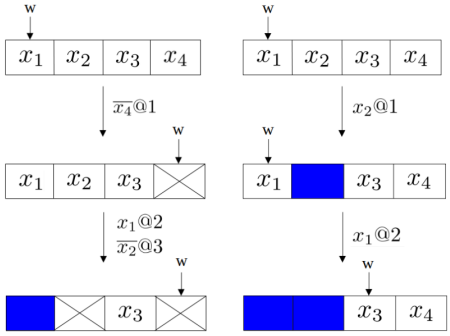

Fig. 1. Falsifying (left) or satisfying (right) a watched literal in a monomial. A satisfied (resp. falsified) variable is represented by a blue (resp. crossed) square. White squares represent unassigned variables. l@i indicates the order of assignment of literal l.

Algorithm 2 SatisfiedVariable(m, l)

INPUT : monomial m, satisfied literal l
OUTPUT : Is m assigned ?

```
1: for x ∈ m do
2:     if x is not assigned then
3:         add m to the list of monomials watched by x
4:         move WL(m) from l to x
5:         return False
6: return True
```

yet assigned. If we cannot find such a replacement, then all the variables present in the considered monomial are satisfied and thus the monomial is also satisfied. We will refer to this process as $SatisfiedVariable(m, l)$ where m is a monomial and l a satisfied literal, with the procedure returning *True* if m can be assigned a value and *False* otherwise. This process is detailed in Algorithm 2. Here, in the worst case, we have to go through the entire monomial to find a replacement or to show that it is assigned. Thus, the complexity of this procedure is in $O(k)$ where k is the size of the longest monomial in the formula.

Example 3: We consider again the monomial $x_1x_2x_3x_4$ with the watched literal initially set to watch x_1. If we first satisfy x_2 then the watched literal is not updated since x_2 is not being watched. Next, if we satisfy x_1, we have to replace the watched literal. By going through the monomial, we notice that the literal x_3 is available and we can move the watched literal from x_1 to x_3. This behaviour is illustrated on the right side of Figure 1. Then, if we successively satisfy x_4 and x_3, we can deduce that the monomial is satisfied because it is not possible to find another replacement.

C. Watched Monomials

First, we recall that the equations can be used to propagate new literals only when exactly one monomial is not yet assigned. Thus, we need to watch two different monomials in each equation. If a watched monomial is assigned (either satisfied or falsified), we have to replace it. If we find a monomial that is not yet assigned and that is not watched, then we can move the watched monomial and stop. However, if no replacement can be found, we have to take into account the second watched monomial. If the latter is already assigned, then we just need to check whether we have satisfied an odd number of monomials in the considered equation. If this is not the case, we have a conflict and we have to backtrack. If the other watched monomial is not assigned yet, we can deduce its value and propagate based on the propagation rules introduced in Section III-A. This process is detailed in Algorithm 3, in which we compute the number of satisfied monomials while looking for a replacement. The term m represents a monomial assigned in a given equation e and the term *other* represents the second watched monomial of e. Here, the worst case happens when there is no replacement. In this case, we have to go through the entire equation. Moreover, if the remaining monomial has to be satisfied, we have to go through this monomial to satisfy the variables it contains (lines 17 to 19). The

Algorithm 3 AssignedMonomial(e, m)

INPUT : equation e, monomial m
OUTPUT : Falsified equation if conflict, Undef otherwise

```
1: odd ← value(m) ⊕ constant(e)
2: for m' ∈ e such that m' is not watched do
3:     if m' is not assigned then
4:         add e to the list of equations watched by m'
5:         move the watched monomial from m to m'
6:         return Undef
7:     odd ← odd ⊕ value(m')
8: add e to the list of equations watched by m
9: if the second watched monomial (other) is assigned then
10:    if odd ⊕ value(other) = False then
11:        return e
12: else
13:    if odd = False then
14:        for v ∈ other do
15:            add v to the list of literals to propagate
16:    else if other contains one unassigned variable v then
17:        add v̄ to the list of literals to propagate
18: return Undef
```

complexity of this algorithm is thus in $O(l + k)$ with l the size of the longest equation in the formula and k the size of the longest monomial.

Example 4: Here, we revisit Example 1. In each equation, we consider that the two first monomials are initially watched. If we take the decision $\overline{x_1}$ then we falsify the monomial x_1x_2 which is watched in the equation E_1 and we have to find a replacement. By going through this equation, we notice that the monomial x_2x_3 is already watched and that the monomial x_4 is not yet assigned. We can thus move the first watched monomial of E_1 from x_1x_2 to x_4.

D. Propagation

With watched literals and watched monomials at hand, we may now detail the propagation process. During this process, we have to consider each literal p to propagate. If it is already satisfied then we can ignore it and continue with the next literal to propagate. If it is falsified, then we have a conflict on the equation that propagated the current literal p, which we refer to as $reason(p)$. Otherwise, we can assign p to *True*, update the watched literals and find the monomials that are assigned. We have to consider the monomials that contain $\overline{p}$ (in this case the monomial is falsified) and also the monomials whose watched literal watches p (in this case the monomial is satisfied only if it is not possible to find a replacement). Then, we have to find a replacement in each equation that watches a newly assigned monomial. If we find a conflict during this process, we have to stop the propagation and backtrack. This process is detailed in Algorithm 4. Here, for each variable to propagate, we potentially have to go through several lists of monomials (lines 10, 14 and 20). By taking into account the complexity of the previous algorithms, we obtain a total complexity of $O(n.e.d.(l+k))$ with n the number of variables, e the number of equations, d the number of monomials, l the size of the longest equation and k the size of the longest monomial. It is important to note that our structures do not have to be updated during

Algorithm 4 Propagate()

OUTPUT : Falsified equation if conflict, Undef otherwise

```
1: for each literal p to propagate do
2:   if p is assigned to false then
3:     return reason(p)
4:   if p is not assigned yet then
5:     assign p to True
6:     assigned ← ∅
7:     for each monomial m with p̄ at position idx do
8:       FalsifiedMonomial(m, p̄, idx)
9:       assigned ← assigned ∪ {m}
10:    for each monomial m in which p is watched do
11:      remove m from the monomials watched by p
12:      if SatisfiedVariable(m, p) then
13:        assigned ← assigned ∪ {m}
14:    for all m ∈ assigned do
15:      for each equation e in which m is watched do
16:        remove e from the equations watched by m
17:        e' ← AssignedMonomial(e, m)
18:        if e' ≠ Undef then
19:          return e'
20: return Undef
```

backtrack which entails a constant complexity cost for their maintenance during the this process.

Example 5: Here, we continue Example 4. After deciding $\overline{x_1}$, if we decide $\overline{x_2}$ then we falsify the monomial x_2x_3 in E_1, and we have to find a replacement. As we cannot find a replacement, we have to take into account the second watched monomial, which is x_4. Until now, we have satisfied only the monomial $\top$ and thus we have to falsify the monomial x_4. The latter is unit, so we have to propagate $\overline{x_4}$. This propagation will lead to a conflict in equation E_2 because all its monomials become falsified. As such, we backtrack and flip the decision $\overline{x_2}$.

IV. Experimental results

We have implemented our approach on top of the Glucose solver[1] [21], [22]. In our experiments, we have considered instances from the Weil descent on Semaev polynomials[2]. More specifically, we search for decompositions into 3 points and consider three different sizes of this problem: `ECn1515` ($n = 15, l = 5$), `ECn1916` ($n = 19, l = 6$) and `ECn2919` ($n = 29, l = 9$). For the two first families, we have a set of 25 instances and for the last one, we have a set of 50 instances. Note that the number of variables for these formulae is greater than $3l$, but once the $3l$ variables corresponding to 3 points on the elliptic curve are assigned, the remaining ones can be assigned through propagation [10]. The tests were performed on an AMD EPYC 7513 32-Core Processor colocked at 2.6GHz , with a timeout of 2 hours for each instance.

First, we perform an ablation study by comparing four different versions of our solver, one using both watched literals and watched monomials (*DPLL-ANF*), one with neither structures (*No-Watch*), and the last two versions using only one of the structures independently (*Literals* and *Monomials*). The results obtained in terms of solving time on our benchmark families are showcased in Figure 2. We have not represented the results obtained on the family `ECn2919` as only our approach integrating all the structures was able to solve instances from this family. The observed differences between the complete approach and the others can be explained by the fact that removing either watched literals or watched monomials forces the solver to consider all the equations to find propagations. Specifically, if watched literals are not used, the solver does not know which monomials are assigned. On the other hand, if watched monomials are not used, it does not know in which equations an assigned monomial is present. When both structures are utilized, the solver can easily determine when a monomial is assigned and in which equations it appears. We thus clearly observe that the combination of both watched literals and watched monomials enhances solver performance, with a gain of at least 68% in terms of solving time, i.e., with respect to the second best version where only watched literals are introduced (resp. 79% gain with respect to the worst version where no watched structure is introduced).

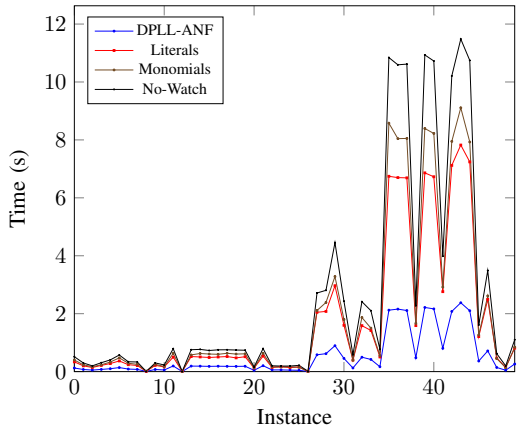

Fig. 2. Comparison of solving times (in seconds) for the families `ECn1515` (first half) and `ECn1916` (second half).

Furthermore, we also compare our approach with the state-of-the-art solvers WDSat [9], [16] and CryptoMiniSat [8], [17]. Our solver and WDSat are both able to read instances in the ANF format although WDSAT internally reasons on the XNF format. It is not the case for CryptoMiniSat, for which we had to translate the instances into XNF formulae. The results are presented in Table II where a dash indicates that the time limit of 2 hours has been exceeded for at least one instance of the corresponding benchmark. For each family, we report the number of variables ($\#V$), equations ($\#E$) and satisfiable/unsatisfiable (S/U) instances. The best execution times are highlighted in bold. If we compare our approach with WDSat without Gaussian Elimination, it seems that these two solvers perform a relatively similar search. However our solver DPLL_ANF performs this search faster than WDSat. Indeed, if we look at the family `ECn2919`, we have obtained a speed-up of 30% (resp. 35%) on average on

[1]Our solver code and benchmark are available at `https://github.com/antektek/DPLL_ANF`

[2]generated using the software at `https://github.com/mtrimoska/EC-Index-Calculus-Benchmarks`

TABLE II

RESULTS IN TERMS OF AVERAGE NUMBER OF CONFLICTS AND SOLVING TIMES IN SECONDS ON ECDLP INSTANCES.

				CryptoMiniSat		WDSat		WDSat (GE)		WDSat (SYM)		DPLL-ANF		DPLL-ANF (SYM)	
Instances	#V	#E	#I	Conflicts	Time	Conflicts	Time	Conflicts	Time	Conflicts	Time	Conflicts	Time	Conflicts	Time
ECn15l5 (S)	42	42	16	40637	2.744	11486	0.074	7189	0.397	3658	0.081	11480	0.068	3650	**0.034**
ECn15l5 (U)			9	113707	11.640	31061	0.211	21649	1.222	5530	0.073	31063	0.188	5530	**0.045**
ECn19l6 (S)	51	52	17	255754	26.584	50631	0.446	32284	2.865	19242	0.193	50623	0.390	19235	**0.148**
ECn19l6 (U)			8	1598294	189.002	255502	2.652	192140	18.669	43999	0.476	255509	2.166	43991	**0.377**
ECn29l9 (S)	78	80	27	-	-	26458588	647.259	-	-	9357380	205.516	26458578	451.885	9356878	**144.307**
ECn29l9 (U)			23	-	-	133844879	3704.768	-	-	22400272	558.515	133844883	2395.759	22399894	**369.192**
Total				-	-	160652147	4355.410	-	-	31830081	764.854	160652136	2850.456	31829178	**514.103**

satisfiable (resp. unsatisfiable) instances compared to WDSat. This seems to indicate that the use of our watched literals and watched monomials is relevant. Concerning WDSat, we can see that the use of the Gaussian Elimination helps to reduce the number of conflicts encountered but it also slows down the solver. Among all the tested approaches, our solver achieves the best solving times on these instances.

Finally, we have implemented the symmetry breaking technique introduced in [10]. The model corresponds to n/l binary vectors of length l, denoted by x_i, which yield the x-coordinates of points in the decomposition. Since any permutation of these points also represents a valid decomposition, a permutation of these vectors is also a model of the ANF formula. To avoid redundancy, we assume that $x_1 \leq x_2 \ldots \leq x_{n/l}$, where $\leq$ is the lexicographic order on length l Boolean vectors with $\bot \leq \top$. Consequently, it is possible to ignore some branches of the search tree. The results clearly show that the use of this symmetry breaking technique (SYM) can drastically improve the performances of both WDSAT and DPLL_ANF. Our solver still performs better, with about 30% gain in terms of solving time on the largest family `ECn29l9` compared to WDSat.

V. CONCLUSION

In this paper, we have introduced a new solver which is able to reason directly on ANF formulae while integrating mechanisms inspired from SAT solvers. More specifically, we have devised a two-level watched data structure within a DPLL-based search dedicated to ANF formulae, in order to speed-up propagation. Our experiments show that our approach achieves superior results on cryptographic attacks on the discrete logarithm problem for binary elliptic curves compared to state-of-the-art solvers, for which Gaussian Elimination (GE) is known to be inefficient. As future work, it would be interesting to study whether our structures could be exploited to incorporate GE into our solver therefore enabling to explore other types of attacks. Our work also paves the way for integrating other techniques dedicated to ANF inspired from modern SAT solvers such as clause learning or dedicated branching heuristics.

ACKNOWLEDGEMENT

This work is supported by the projects ANR-20-ASTR-0011 (POSTCRYPTUM) and ANR-24-CE23-6126 (BforSAT) funded by the French National Research Agency. This work was also granted access to HPC resources of "Plateforme MatriCS" within University of Picardie Jules Verne, co-financed by the European Union with the European Regional Development Fund (FEDER) and the Hauts-De-France Regional Council among others.

REFERENCES

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability - Second Edition*. IOS Press, 2021.

[2] H. Zhang, "SATO: an efficient propositional prover," in *CADE-14*. Springer, 1997, pp. 272–275.

[3] I. Lynce and J. Marques-Silva, "Efficient data structures for backtrack search SAT solvers," *Ann. Math. Artif. Intell.*, vol. 43, no. 1, pp. 137–152, 2005.

[4] J. P. M. Silva and K. A. Sakallah, "GRASP - a new search algorithm for satisfiability," in *ICCAD 1996*. IEEE-CS / ACM, 1996, pp. 220–227.

[5] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability - Second Edition*. IOS Press, 2021, pp. 133–182.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC 2001*. ACM, 2001, pp. 530–535.

[7] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *SAT 2016*. Springer, 2016, pp. 123–140.

[8] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *SAT 2009*. Springer, 2009, pp. 244–257.

[9] M. Trimoska, S. Ionica, and G. Dequen, "Parity (XOR) Reasoning for the Index Calculus Attack," in *CP2020*. Springer, 2020, pp. 774–790.

[10] ——, "A SAT-Based Approach for Index Calculus on Binary Elliptic Curves," in *AFRICACRYPT 2020*. Springer, 2020, pp. 214–235.

[11] I. A. Semaev, "Summation polynomials and the discrete logarithm problem on elliptic curves," *IACR Cryptology ePrint Archive*, vol. 2004, p. 31, 2004. [Online]. Available: http://eprint.iacr.org/2004/031

[12] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *SAT 2009*. Springer, 2009, pp. 244–257.

[13] A. Ramamoorthy and P. Jayagowri, "The state-of-the-art boolean satisfiability based cryptanalysis," *Materials Today: Proceedings*, vol. 80, pp. 2539–2545, 2023, sI:5 NANO 2021.

[14] E. Bellini, A. De Piccoli, R. Makarim, S. Polese, L. Riva, and A. Visconti, "New records of pre-image search of reduced sha-1 using sat solvers," in *Proceedings of the Seventh International Conference on Mathematics and Computing*. Singapore: Springer Singapore, 2022, pp. 141–151.

[15] A. D. Dwivedi, M. Kloucek, P. Morawiecki, I. Nikolic, J. Pieprzyk, and S. Wójtowicz, "Sat-based cryptanalysis of authenticated ciphers from the CAESAR competition," in *ICETE 2017*. SciTePress, 2017, pp. 237–246.

[16] M. Trimoska, G. Dequen, and S. Ionica, "Logical cryptanalysis with WDSat," in *SAT 2021*. Springer, 2021, pp. 545–561.

[17] M. Soos, "Enhanced Gaussian Elimination in DPLL-based SAT Solvers," in *POS-10*. EasyChair, 2010, pp. 2–14.

[18] M. A. Huang, M. Kosters, and S. L. Yeo, "Last fall degree, hfe, and weil descent attacks on ECDLP," in *CRYPTO 2015*, ser. Lecture Notes in Computer Science, vol. 9215. Springer, 2015, pp. 581–600.

[19] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[20] M. Davis, G. Logemann, and D. W. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[21] G. Audemard and L. Simon, "On the glucose SAT solver," *Int. J. Artif. Intell. Tools*, vol. 27, no. 1, pp. 1 840 001:1–1 840 001:25, 2018.

[22] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*. Springer, 2003, pp. 502–518.