Logical cryptanalysis with WDSat

Monika Trimoska, Gilles Dequen, Sorina Ionica

Logical cryptanalysis with WDSat

Monika Trimoska and Gilles Dequen and Sorina Ionica

Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France *

Abstract. Over the last decade, there have been significant efforts in developing efficient XOR-enabled SAT solvers for cryptographic applications. In [22] we proposed a solver specialised to cryptographic problems, and more precisely to instances arising from the index calculus attack on the discrete logarithm problem for elliptic curve-based cryptosystems. Its most prominent feature is the module that performs an enhanced version of Gaussian Elimination. [22] is concentrated on the theoretical aspects of the new tool, but the running time-per-conflict results suggest that this module uses efficient implementation techniques as well. Thus, the first goal of this paper is to give a comprehensive exposition of the implementation details of WDSat. In addition, we show that the WDSat approach can be extended to other cryptographic applications, mainly all attacks that involve solving dense Boolean polynomial systems. We give complexity analysis for such systems and we compare different state-of-the-art SAT solvers experimentally, concluding that WDSat gives the best results. As a second contribution, we provide an original and economical implementation of a module for handling OR-clauses of any size, as WDSat currently handles OR-clauses comprised of up to four literals. We finally provide experimental results showing that this new approach does not impair the performance of the solver.

1 Introduction

Due to the significant number of improvements in SAT-based parity reasoning over the last decade, SAT solvers are gaining popularity in cryptographic applications. More specifically, they are often used to tackle the solving phase in algebraic cryptanalysis of stream ciphers [18,14,11,21,19], and more recently, of public-key cryptosystems [10,23]. Algebraic cryptanalysis includes all attacks where the underlying problem of a cryptographic system is reduced to the problem of solving a multivariate polynomial system of equations. The resulting system is solved using algebraic techniques, such as Gröbner basis algorithms [9], exhaustive search [5], hybrid methods [2] or algorithms in the XL family [6]. Finding a solution to the polynomial system constitutes a successful attack and results in recovering (a part of) the secret key or the plaintext. Boolean polynomial systems may be easily re-written as SAT formulas, which are then solved using a SAT solver. This technique is referred to as logical cryptanalysis [17].

* We acknowledge financial support from the European Union under the 2014/2020 European Regional Development Fund (FEDER) and from the Agence Nationale de Recherche under project ANR20-ASTR-0011.

The transformation of a Boolean polynomial system into a CNF formula is done in three steps, each resulting in a propositional formula in different form. First, we obtain an Algebraic Normal Form (ANF) by replacing all multiplications over the binary field by a logical AND and all sums by the XOR operator. The next step is to eliminate all conjunctions through a linearization-like process that consists in replacing all occurrences of $(x_1 \wedge \ldots \wedge x_k)$ by a newly added variable $x_{1,\ldots,k}$ and adding the constraint $x_{1,\ldots,k} \Leftrightarrow (x_1 \wedge \ldots \wedge x_k)$ to the model in its CNF equivalence $(\neg x_{1,\ldots,k} \vee x_1) \wedge \ldots \wedge (\neg x_{1,\ldots,k} \vee x_k) \wedge (\neg x_1 \vee \ldots \vee \neg x_k \vee x_{1,\ldots,k})$. This step results in a so-called CNF-XOR formula which is a conjunction of both OR-clauses and XOR-clauses. Classically, a XOR-clause of size k can be rewritten either as a conjunction of 2^{k-1} k-OR-clauses or as a conjunction of 3-OR-clauses, if the XOR-clause is first cut up into 3-XOR-clauses.

Since the XOR operator is at the core of reasoning models obtained from cryptographic attacks, significant effort has been put into developing XOR-enabled SAT solvers that read formulas in CNF-XOR form and are adapted to reason directly on XOR constraints. In this paper, we give implementation details of one such solver, named WDSat, proposed in [22]. WDSat is a built-from-scratch DPLL-based SAT solver that is specifically designed for solving ANF instances derived from cryptographic attacks on public-key cryptosystems. These formulas have few variables, but are highly dense, i.e. they have very long XOR-clauses. The original proposal of WDSat shows experimental results on formulas derived from an attack on elliptic curve-based cryptosystems. In contrast, in this paper, we experiment with WDSat and other state-of-the-art SAT solvers using instances derived from the Multivariate Quadratic (MQ) problem, which is the problem of finding all common zeros of a multivariate quadratic system of polynomials. The following toy example shows an MQ system with 4 equations in 3 variables over the binary field.

$$\begin{aligned}
&\mathbf{x}_1\mathbf{x}_2 + \mathbf{x}_1\mathbf{x}_3 + \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + 1 = 0\\
&\mathbf{x}_1\mathbf{x}_2 + \mathbf{x}_2\mathbf{x}_3 + \mathbf{x}_1 + \mathbf{x}_3 = 0\\
&\mathbf{x}_1\mathbf{x}_2 + \mathbf{x}_3 + 1 = 0.
\end{aligned}$$

In addition, we propose an original technique with reduced amount of memory, which allows to handle large size clauses and thus, solve multivariate polynomial systems of any degree. We report experimental results with WDSat on multivariate polynomial systems of degree three and four. Solving the multivariate polynomial problem is at the core of algebraic cryptanalysis, as many cryptographic attacks can be reduced to the problem of solving a multivariate polynomial system of equations.

2 Background

WDSat was proposed at CP 2020 [22] as an XOR-enabled SAT solver dedicated to solving instances derived from a *Weil Descent*. A Weil descent is a technique, commonly used in cryptanalysis, for reducing the problem of finding

roots of a polynomial defined over an extension field to the problem of solving a multivariate polynomial system of equations defined over the base field. The WDSAT solver was particularly designed for the Weil descent steps performed in an attack on elliptic curve-based cryptosystems. The solver is built-from-scratch and based on the DPLL algorithm [7]. It is comprised of three reasoning modules that communicate with each other. One is used for reasoning on the CNF part of the formula and the other two are used for XOR reasoning. When an assumption of a truth value is made, the literal is first set in the CNF module. Then, all propagated literals are recovered and are set, together with the initial assumption, in the second module, called XORSET. Finally, all literals propagated by the CNF and XORSET modules are set in the third module, called XORGAUSS. If the XORGAUSS module results in more propagated literals, the process is repeated, until all modules can no longer propagate. Each module is equipped with a corresponding propagation, conflict detection and backtracking technique. The CNF and XORSET modules use classic techniques for unit propagation on OR and XOR-clauses respectively. Handling the XOR-clauses instead of breaking them down into a CNF is beneficial for SAT solving, as it allows us to use powerful techniques inspired from algebraic solving tools, such as the Gaussian Elimination (GE). Performing GE generally results in fewer conflicts, but is computationally expensive. Thus, the purpose of the XORGAUSS module is to perform (GE) on the XOR part of the formula efficiently. In this module, XOR-clauses are represented as Equivalence Classes (EC). A *representative* is chosen for each class and the GE technique consists in assigning a truth value to a variable while applying defined rules that ensure that the so-called unicity-of-representatives property is maintained. This property states that a representative of an EC will never be present in another EC. Thus, the notion of representative of an EC is analogous to the notion of *pivot* in linear algebra.

In addition, starting from the observation that existing SAT-based implementations of the GE are not as efficient as GE in algebraic tools, we proposed in [22] an extended version of the XORGAUSS module (XG-EXT). Indeed, in a Gröbner-basis based approach [13], when a variable $\mathbf{x}_i$ is set to 1, all occurrences of a monomial $\mathbf{x}_i\mathbf{x}_j$ are replaced by $\mathbf{x}_j$ and can be canceled out with other occurrences of $\mathbf{x}_j$. Recall that a monomial $\mathbf{x}_i\mathbf{x}_j$ from the initial Boolean polynomial system becomes $x_i \wedge x_j$ in the equivalent propositional formula and is replaced by a newly added variable $x_{i,j}$. The CNF block that we obtain from this substitution is $(x_i \vee \neg x_{i,j}) \wedge (x_j \vee \neg x_{i,j}) \wedge (\neg x_i \vee \neg x_j \vee x_{i,j})$. When we set x_i to TRUE and apply the unit propagation rules, we are left with the following OR-clauses:

$$\begin{aligned}&(x_j \vee \neg x_{i,j}) \wedge \\ &(\neg x_j \vee x_{i,j}).\end{aligned} \tag{1}$$

In the XOR part of the CNF-XOR formula, x_j and $x_{i,j}$ are two different variables and a possible cancellation of terms can be overseen. To fix this oversight, the following rule is added to WDSAT . When $x_{i,j} \Leftrightarrow (x_i \wedge x_j)$ and we set x_i to TRUE, $x_{i,j}$ is replaced by x_j. To perform the substitution of $x_{i,j}$ by x_j, propagation rules, similar to the ones for truth value assignment, are defined for

maintaining the unicity-of-representatives property. This constitutes the XG-EXT module.

Since this oversight is due to the CNF-XOR input form, it is common for all XOR-enabled SAT solvers that perform GE. However, the newest version of CryptoMiniSat (5.8.0) implements a technique called BIRD [20] that seems to fix the issue as well. The BIRD technique consists in (i) transforming XOR clauses into CNF, (ii) inprocess over CNF clauses, (iii) recover simplified XOR-clauses and (iv) perform CDCL coupled with GE on the CNF-XOR formula. Since this technique is performed during resolution, the recovery process in the third step should be able to recover the XOR-clause $(x_j \oplus x_{i,j} \oplus \top)$ from the two OR-clauses in Equation (1). Adding the recovered clause to the XOR system and performing GE should have a similar result as replacing $x_{i,j}$ by x_j.

3 Implementation details

Input forms The WDSat solver can read formulas in both ANF and CNF-XOR form. Reading a formula in ANF comes with two advantages. The first one is linked to branching rules and the second is that this form allows us to use the extension of the XG module. Since the direct encoding is shorter, in number of clauses, than in CNF-XOR modeling, the use of ANF comes more advantageous within the context of cryptographic problems.

Branching rules Reading a formula in ANF, the solver can store the information of which variables comprise the initial system, as opposed to variables that are added to substitute a conjunction. We can thus, distinguish *unary* variables from *substitution* variables. The truth value of a substitution variable is equal to the conjunction of the truth values of the corresponding unary variables. As a result, assigning truth values to all unary variables will necessarily propagate all other variables (see, for instance, Proposition 1 in [23]). In WDSat, only unary variables are considered in the binary search. Conflict-driven branching heuristics can not be used in WDSat, as the solver does not perform conflict analysis. In addition, there is a heuristic branching technique specific to SAT instances derived from Boolean polynomial systems developed for WDSat. This technique, inspired by the Minimal Vertex Cover problem from graph theory, determines the minimal subset of variables that need to be assigned to obtain a formula comprised only of XOR-clauses. This formula is then solved in polynomial time using GE. The technique is currently used only during preprocessing to provide a predetermined branching order that is optimal. Thus, the solver does not use heuristics to decide on the order of branching variables dynamically, but the order can be specified by the user. This feature is to be used if the user has more information on the system or if the preprocessing technique was applied.

3.1 Three reasoning modules

In this section, we give a description and implementation details of the three modules that make up the WDSat solver and we propose a novel CNF module

that can handle longer clauses. Each module has its own propagation stack, called the $CNF\_propagation\_stack$, the $XORSET\_propagation\_stack$ and the $XG\_propagation\_stack$, as well as a respective SET_IN function that sets a literal to TRUE in the corresponding module. These stacks are used for communication between the modules. For simplicity, we consider that these stacks and all other data structures relative to the modules are included in a structure F, simply referred to as the propositional formula.

CNF module In this module, responsible for unit propagation on OR-clauses, the OR-clauses are treated as lists of implications, following an idea of Heule *et al.* [12] for handling 3-OR-clauses, implemented in the March SAT solver. In addition, the method is extended to handle 4-OR-clauses. Hence, WDSAT is able to solve instances derived from Boolean polynomial systems of degree three at most.

Compressed CNF reasoning In this section, we propose an original method for handling OR-clauses, using a compact data structure and simple bitwise operations. Our module serves as an addition to the WDSAT solver, as it allows us to handle OR-clauses of any size. In this module, OR-clauses are stored as bit-vectors comprised of the following three parts: the *value* of the clause is the arithmetic sum of its literals in their *dimacs* representation, the *weight* of the clause is the number of unassigned literals left in the clause and the final part, referred to as the *sat assessment* is composed of only one bit that is set to 1 when the clause is already satisfied by one of its assigned literals, and to 0 otherwise. The *value* and the *weight* bit-vectors have a predetermined static length. The first two lines in Table 1 show an example of the representation of two OR-clauses. As an illustration, the *value* of $\neg x_1 \vee x_4 \vee \neg x_2$ is $(-1) + 4 + (-2) = 1$, and the *weight* equals 3.

Let k be the number of variables in a CNF, and let W be the length of the longest OR-clause. The length of a bit-vector representing a clause in this manner is given by the formula:

$$\lceil \log_2(2Wk) \rceil + \lceil \log_2(W) \rceil + 1. \tag{2}$$

Since the increase is asymptotically logarithmic, a 64-bit integer can easily represent very long clauses. Hence, a formula is an array of integers, denoted *clauses*, where each entry represents a clause. In the remainder of this section, we will use $|W|$ to denote the length (in bits) of the maximal *weight*.

To perform unit propagation, we need to have efficient access to the occurrences of each literal. More specifically, we allocate an array *occ_in_clause* indexed by signed literals. Each entry in the array holds a list of clauses in which the corresponding literal occurs. When we set a literal l to TRUE, we perform the following operations. As per the first rule of unit propagation, the *sat assessment* is set to 1 in all clauses from the list $occ\_in\_clause[l]$. As per the second unit propagation rule, $-l$ is subtracted from the *value* of all clauses from the list $occ\_in\_clause[-l]$, and the *weight* of these clauses is decremented.

Table 1: Example of two clauses in a CNF with 4 variables and maximum clause length 4.

OR-clause	Bit-vector			Decimal
	value	*weight*	*sat*	
$\neg x_1 \vee x_4 \vee \neg x_2$	00001	011	0	22
$x_1 \vee x_3$	00100	010	0	68
Set x_1 to FALSE.				
$\neg x_1 \vee x_4 \vee \neg x_2$	00001	011	**1**	23
x_3	00011	001	0	50
Propagation: x_3 is set to TRUE.				

With our compact representation, clauses are managed using only bitwise operations. More specifically, we use the following functions, where, as per the C syntax, $\ll$ and $\gg$ denote the left and right bitwise shift, & denotes the bitwise AND , and | denotes the bitwise OR. These functions are used in the SET_IN_CNF function, given in Algorithm 1.

- GET_CLAUSE_VALUE(cl) : $clauses[cl] \gg (|W| + 1)$;
- GET_CLAUSE_WEIGHT(cl) : $(clauses[cl] \gg 1)$ & $(2^{|W|} - 1)$;
- LITERAL_TO_CLAUSE(l) : $(l \ll (|W| + 1)) \mid 2$;
- IS_CLAUSE_SAT(cl) : $clauses[cl]$ & 1;
- SET_CLAUSE_TO_SAT(cl) : $clauses[cl] \leftarrow clauses[cl] \mid 1$;

As we can see in Algorithm 1, a propagation is detected when the *weight* of a clause is equal to 1. In this case, the *value* of the clause is equal to the *dimacs* representation of the only remaining literal that can satisfy the clause, and thus, the literal is directly propagated. A conflict occurs when we try to assign a variable that is already assigned to the opposite truth value. The second part of Table 1 shows an example of the changes that are made in the *clauses* structure from the execution of Algorithm 1.

This economical structure is adapted for the requirements of WDSAT and the classic DPLL algorithm. In this paradigm, it is never required to get all literals from a specific clause, or to check which literals are unassigned, unless there is only one unassigned literal left. Thus, it is not concerning that these operations can not be done efficiently in our CNF module.

XORSET module XORSET is a simple module for parity reasoning. In other words, this module performs unit propagation on XOR-clauses. The unit propagation rule can be informally defined as follows. When all except one literal in an XOR-clause are assigned, the remaining literal is given a truth value according to parity reasoning. Recall that an XOR-clause is satisfied if there is an odd number of literals that are set to TRUE.

During the solving process, the solver counts the number of literals in a clause that are set to TRUE, and respectively the ones that are set to FALSE. In order to do this efficiently, the solver needs to have quick access to the occurrences of each

Algorithm 1 Function SET_IN_CNF(to_set, F) : Function that sets a list of literals to TRUE.

Input: A list of literals that need to be set to TRUE, the propositional formula F
Output: FALSE if unsatisfiability is detected with unit propagation, TRUE otherwise.

```
1: CNF_propagation_stack ← to_set.
2: while CNF_propagation_stack is not empty do
3:     l ← top element from CNF_propagation_stack.
4:     if assignment[l] ≠ TRUE then
5:         if assignment[l] = FALSE then
6:             return FALSE.
7:         end if
8:         assignment[l] ← TRUE.
9:         for each cl in occ_in_clause[l] do
10:            SET_CLAUSE_TO_SAT(cl).
11:        end for
12:        for each cl in occ_in_clause[−l] do
13:            if not IS_CLAUSE_SAT(cl) then
14:                clauses[cl] ← clauses[cl]− LITERAL_TO_CLAUSE(−l).
15:                if GET_CLAUSE_WEIGHT(cl) = 1 then
16:                    l_prop ← GET_CLAUSE_VALUE(cl).
17:                    add l_prop to CNF_propagation_stack.
18:                end if
19:            end if
20:        end for
21:    end if
22: end while
23: return TRUE.
```

literal. At the implementation level, the structure that keeps this information is an array indexed by both positive and negative literals that contains lists of clauses in which a literal appears. This is a classical technique for implementing basic XOR reasoning in a SAT solver.

XORGAUSS module As explained in Section 2, in this module, XOR-clauses are represented as equivalence classes. To obtain this representation, the first step is to *normalise* all clauses so that they contain only positive literals and do not contain more than one occurrence of each literal. To eliminate negative literals, normalised clauses may contain a $\top$ constant. All variables in a clause are considered to belong to the same equivalence class (EC), and one literal from the EC is chosen to be the representative. An XOR-clause $(x_1 \oplus x_2 \oplus ... \oplus x_n) \Leftrightarrow \top$ rewrites as $x_1 \Leftrightarrow (x_2 \oplus x_3 \oplus ... \oplus x_n \oplus \top)$. The initialization process of the XG module consists in performing the following steps for each XOR-clause : (i) put the clause in normal form, (ii) transform the clause into an EC and (iii) replace all occurrences of its representative in the system with the right side of the equivalence. Applying this transformation, we obtain a simplified system having the unicity-of-representatives property.

Example 1. Let us consider the following set of three XOR-clauses.

$$x_1 \oplus x_4 \oplus x_5 \oplus x_6$$
$$x_1 \oplus x_2 \oplus \neg x_4$$
$$x_2 \oplus x_3 \oplus \neg x_6$$

The steps of the initialization process of this formula are shown in Table 2. The left column shows the set of equivalence classes that grows with each step. The right column shows the set of remaining XOR-clauses. We consider that all clauses are already put in normal form. This set becomes smaller as each clause is transformed into an equivalence class.

Table 2: Equivalence classes initialization steps.

Set of equivalence classes	Set of XOR-clauses
$\emptyset$	$x_1 \oplus x_4 \oplus x_5 \oplus x_6$ $x_1 \oplus x_2 \oplus x_4 \oplus \top$ $x_2 \oplus x_3 \oplus x_6 \oplus \top$
$x_1 \Leftrightarrow x_4 \oplus x_5 \oplus x_6 \oplus \top$	$x_2 \oplus x_5 \oplus x_6$ $x_2 \oplus x_3 \oplus x_6 \oplus \top$
$x_1 \Leftrightarrow x_4 \oplus x_5 \oplus x_6 \oplus \top$ $x_2 \Leftrightarrow x_5 \oplus x_6 \oplus \top$	$x_3 \oplus x_5$
$x_1 \Leftrightarrow x_4 \oplus x_5 \oplus x_6 \oplus \top$ $x_2 \Leftrightarrow x_5 \oplus x_6 \oplus \top$ $x_3 \Leftrightarrow x_5 \oplus \top$	$\emptyset$

At the implementation level, XOR-clauses are represented as bit-vectors. If a variable is present in the clause, the corresponding bit is set to 1, otherwise it is set to 0. Plus, the first bit in the vector is used for the $\top/\bot$ constant. For a compact representation, bit-vectors are stored in an array of 64-bit integers. For instance, to store a k-bit vector, an array of $\lceil (k+1)/64 \rceil$ integers is allocated. Finally, the clauses represented in this manner are stored in an array indexed by the representatives. This array is the core structure of the XG module and it will be referred to as the EC structure. For an example of the EC structure, see Figure 1 that illustrates the set of equivalence classes that we obtain through the transformation in Table 2. In this illustration, each line represents one equivalency and is labeled with the representative. The columns are colored in gray if and only if the corresponding variable belongs to the right side of the equivalency. The constant is referenced in the first column.

To explain the implementation choices, in Table 3 we recall the inference rules from [22] for performing GE in the XG module of WDSAT. In this table, R denotes the set of representatives and C denotes the set of clauses. C_x is an XOR-clause in C that is represented by an EC with representative x. Finally, $var(C_x)$ denotes the set of literals (plus a $\top/\bot$ constant) in the clause C_x and

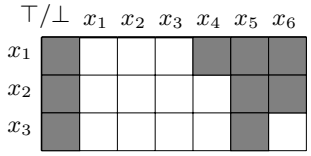


Fig. 1: The EC structure.

the notation $C[x_1/\phi]$ is used when the literal x_1 is replaced by ϕ in all clauses, where ϕ may be a clause, a variable or a constant.

Table 3: Inference rules for the substitution of x_1 by a TRUE/FALSE constant.

Premises	Conclusions on C	Updates on R
x_1, C $x_1 \notin R$	$C[x_1/\top]$	$_{N/A}$
x_1, C $x_1 \in R$ $x_2 \in var(C_{x_1})$	$C_{x_2} \leftarrow C_{x_1} \oplus x_2 \oplus \top$ $C[x_2/C_{x_2}]$	$R \leftarrow R \setminus \{x_1\}$ $R \leftarrow R \cup \{x_2\}$

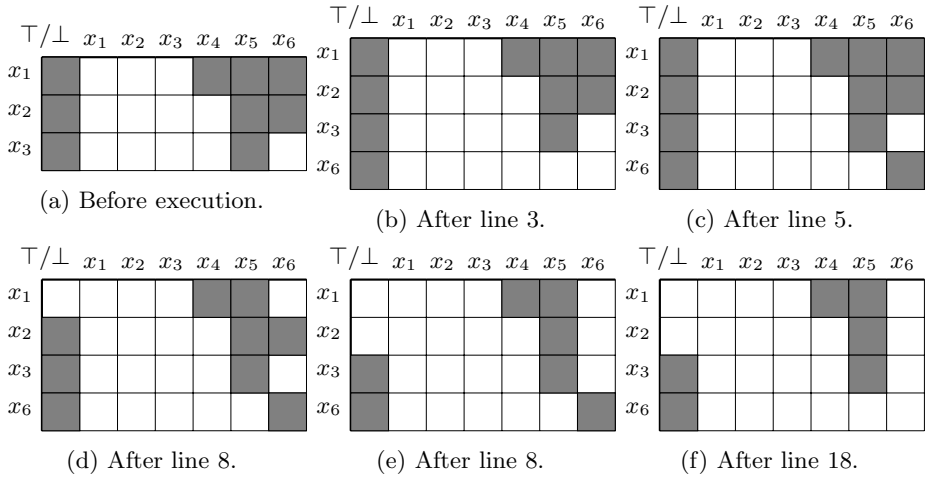
This representation of equivalence classes allows for an efficient implementation of the inference rules, where the main operations are XOR-ing bit-vectors and flipping the clause constant. The first rule, for whose application we give pseudo-code in Algorithm 2, corresponds to the case where x_1 is not a representative. In a bit-vector from the EC structure, individual bits can be set to 0, set to 1 or their value can be checked. We distinguish variable bits from the constant bit. Other operations that modify the EC structure are FLIP_CONSTANT, used simply to inverse the value of the constant bit, and the operator $\oplus$ that denotes the XOR-ing of two bit-vectors. Lines in Algorithm 2 that contain operations that modify the EC structure are in bold. For a better understanding of the infer algorithm, we provide an execution example in Figure 2. In this example, we show the contents of the EC structure after the execution of each line in bold. The infer function corresponding to the second inference rule, where x_1 is in the set of representatives, is detailed in Algorithm 3. In this algorithm, a RESET_VECTOR function is used that simply sets all the bits in a given bit-vector to 0. The execution example for this algorithm is given in Figure 3.

Finally, everything is linked together in the SET_IN function of the XG module, detailed in Algorithm 4. In this algorithm, the GET_PROPOSITIONAL_VARIABLE function extracts the propositional variable from a literal and the GET_TRUTH_VALUE function checks whether l is a positive or a negative literal. For instance, calling GET_PROPOSITIONAL_VARIABLE($\neg x_1$) would return x_1 and GET_TRUTH_VALUE($\neg x_1$) would return FALSE.

Algorithm 2 Function INFER_NON_REPRESENTATIVE(ul, tv, F) : Function that applies the first inference rule to the EC structure.

Input: Propositional variable ul, truth value tv, the propositional formula F
Output: The EC structure and the *XG_propagation_stack* are modified.

```
 1: add ul to R.
 2: if tv =TRUE then
 3:     FLIP_CONSTANT(EC[ul]).
 4: end if
 5: set ul to 1 in EC[ul].
 6: for each r in R do
 7:     if ul is set to 1 in EC[r] then
 8:         EC[r] ← EC[r] ⊕ EC[ul].
 9:         if all variable bits in EC[r] are set to 0 then
10:             if the constant bit in EC[r] is set to 1 then
11:                 add r to XG_propagation_stack.
12:             else
13:                 add ¬r to XG_propagation_stack.
14:             end if
15:         end if
16:     end if
17: end for
18: set ul to 0 in EC[ul].
```



(a) Before execution. (b) After line 3. (c) After line 5.

(d) After line 8. (e) After line 8. (f) After line 18.

Fig. 2: Setting x_6 to TRUE. Stream of changes on the EC structure after execution of the respective lines of Algorithm 2.

4 Applications in cryptanalysis

At the core of algebraic cryptanalysis, as well as multivariate public-key cryptography is the problem of solving a multivariate polynomial system, which is

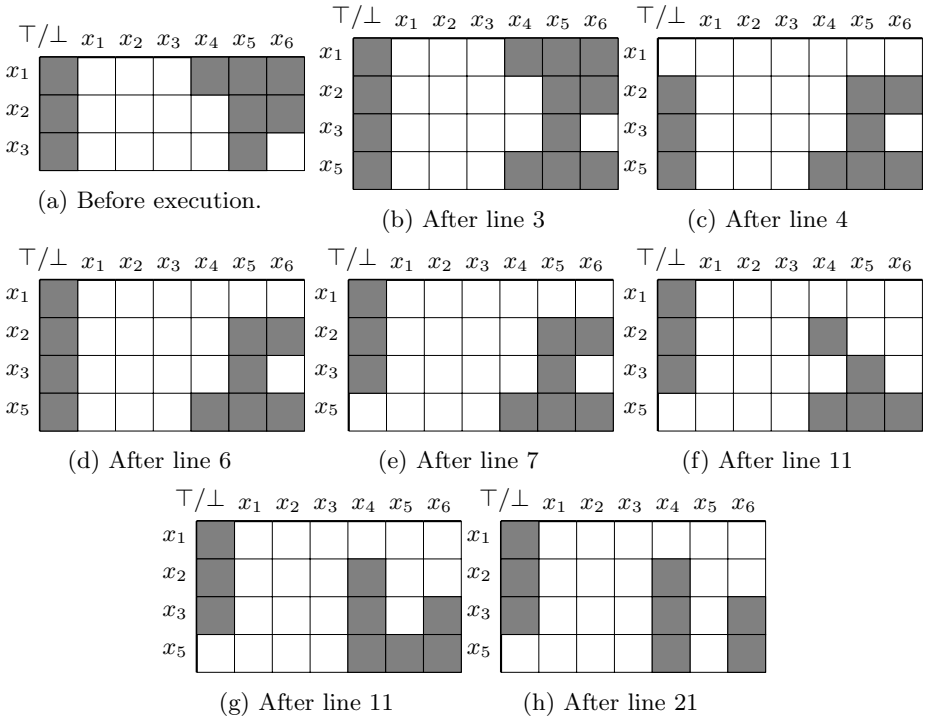
Algorithm 3 Function INFER_REPRESENTATIVE(ul, tv, F) : Function that applies the second inference rule to the EC structure.

Input: Propositional variable ul, truth value tv, the propositional formula F
Output: The EC structure and the $XG\_propagation\_stack$ are modified.

```
 1: new_r ← CHOOSE_NEW_REPRESENTATIVE(EC[ul]).
 2: add new_r to R.
 3: EC[new_r] ← EC[new_r] ⊕ EC[ul].
 4: RESET_VECTOR(EC[ul]).
 5: if tv = ⊤ then
 6:     FLIP_CONSTANT(EC[ul]).
 7:     FLIP_CONSTANT(EC[new_r]).
 8: end if
 9: for each r in R do
10:     if new_r is set to 1 in EC[r] then
11:         EC[r] ← EC[r] ⊕ EC[new_r].
12:         if all variable bits in EC[r] are set to 0 then
13:             if the constant bit in EC[r] is set to 1 then
14:                 add r to XG_propagation_stack.
15:             else
16:                 add ¬r to XG_propagation_stack.
17:             end if
18:         end if
19:     end if
20: end for
21: set new_r to 0 in EC[new_r].
22: if all variable bits in EC[new_r] are set to 0 then
23:     if the constant bit in EC[new_r] is set to 1 then
24:         add new_r to XG_propagation_stack.
25:     else
26:         add ¬new_r to XG_propagation_stack.
27:     end if
28: end if
```

considered to be NP-hard. The crucial parameters in evaluating the hardness of a multivariate polynomial system are the number of variables, denoted by n, the number of equations, denoted by m and their ratio. The case of $m = n$ is considered to be the hardest, whereas overdetermined systems are easier to solve.

For our experimental work, we generate instances with parameters $m = 2n$ and with (pseudo)random solutions, where all coefficients are randomly generated following the uniform distribution. The process of generating one random instance follows these steps: (i) Fix parameters m and n. (ii) Choose randomly an n-bit solution vector. (iii) For each equation, choose randomly all coefficients except the 0/1 constant, and then compute the constant according to the solution vector chosen in the previous step. This generation approach results in dense polynomial systems, as each monomial has probability 1/2 to appear in each equation. Heuristically, we expect most instances obtained in this way to



(a) Before execution. (b) After line 3 (c) After line 4

(d) After line 6 (e) After line 7 (f) After line 11

(g) After line 11 (h) After line 21

Fig. 3: Setting x_1 to TRUE (x_5 is chosen as the new representative). Stream of changes on the EC structure after execution of the respective lines of Algorithm 3.

be hard and to have no underlying structure. For MQ systems, currently the best solving tools are algebraic tools, such as the Joux-Vitse hybrid algorithm [13] and the libFes library [4] based on Bouillaguet *et al*'s algorithm [5]. According to experimental results reported in [13], WDSat does not outperform algebraic solving tools for MQ instances.

Complexity analysis The complexity analysis in this section concerns MQ systems, but it can be extended for systems of higher degree. Let v denote the number of variables in a SAT instance. Since WDSAT is DPLL-based, we consider that the worst-case time complexity is, in general, $\mathcal{O}(2^v)$. However, as explained in Section 3, WDSAT has an advantage over generic SAT solvers when it reads a formula in ANF, as it decides to branch only on variables that are present in the initial Boolean polynomial system. Thus, the complexity is $\mathcal{O}(2^n)$ where $n \leq v$ (with equality only in the case where the initial Boolean system is linear). Moreover, for instances derived from the MQ problem, we can make a more precise estimation. The following analysis concerns the WDSAT XG-EXT variant, as this variant was found to be the most efficient for MQ systems. Our

Algorithm 4 Function SET_IN_XG(to_set, F) : Function that sets a list of literals to TRUE.

Input: A list of literals that need to be set to TRUE, the propositional formula F
Output: FALSE if unsatisfiability is detected with unit propagation, TRUE otherwise.

```
1: XG_propagation_stack ← to_set.
2: while XG_propagation_stack is not empty do
3:     l ← top element from XG_propagation_stack.
4:     if assignment[l] ≠ TRUE then
5:         if assignment[l] = FALSE then
6:             return FALSE.
7:         end if
8:         assignment[l] ← TRUE.
9:         ul ← GET_PROPOSITIONAL_VARIABLE(l).
10:        tv ← GET_TRUTH_VALUE(l).
11:        if x_1 ∈ R then
12:            INFER_REPRESENTATIVE(ul, tv, F).
13:        else
14:            INFER_NON_REPRESENTATIVE(ul, tv, F).
15:        end if
16:    end if
17: end while
18: return TRUE.
```

complexity analysis is based on estimating the DPLL-tree level on which conflicts are found.

Recall that the EC structure in the XORGAUSS module of the WDSAT solver can be viewed as a matrix whose columns are all monomials and unary variables in the system, and the lines are linear XOR-clauses, similar to the Macaulay matrix [16]. Thus, this matrix holds a linearized system that will have a unique solution when the number of lines is equal to the number of columns. This is true because the GE that is performed on each step ensures that all remaining XOR-clauses represent linearly independent equations. For an MQ system, the number of columns in the EC structure, supposing that all monomials have at least one occurrence, is $n(n+1)/2$. It is well-known that overdetermined systems where $m \geq n(n+1)/2$ are solvable in polynomial time. Let n' be the number of remaining variables in the system after the solving process has started. Then, at level h of the binary search tree, we have that $n' = n - h$. As per our previous analysis, the system is solved or a conflict is met when $m \approx n'(n'+1)/2$, i.e. at level $h \approx n - \sqrt{2m}$. We conclude that the complexity of WDSAT with XG-EXT for solving instances derived from the MQ problem is

$$\mathcal{O}(2^{n-\sqrt{2m}}). \tag{3}$$

Even though this analysis is strongly linked to the GE, it does not necessarily hold for other SAT solvers that perform GE, such as CRYPTOMINISAT. If a solver does not apply the XG-EXT technique, it can not be guaranteed that the number

of remaining columns in the EC matrix with entries different from 0 will be $n'(n'+1)/2$ on level h.

Experimental results Table 4 shows a comparison between different approaches for solving MQ systems. These experiments were performed on a 2.40GHz Intel Xeon E5-2640 processor, all results are an average of 100 runs and running times are in seconds. The first four entries show the performance of non XOR-

Table 4: Comparing different approaches for solving the MQ problem.

n	m	Input form	#Vars	#Clauses	Solver	Runtime	#Conflicts
25	50	CNF	8301	33006	MiniSat	11525.24	40718489
					Glucose	2384.99	10982657
					Kissat	2118.52	6622284
					Relaxed	3014.22	10353009
		CNF-XOR	325	920	CryptoMiniSat 5.6.5	2598.66	9806242
					CryptoMiniSat 5.6.5 + GE	383.06	2007847
					CryptoMiniSat 5.8.0	2870.81	9197978
					CryptoMiniSat 5.8.0 + GE	594.48	2407635
					WDSat	57.85	14177200
					WDSat + GE	23.77	1046328
		ANF	25	50	WDSat + XG-EXT	**0.82**	**21140**
30	60	CNF-XOR	465	1365	CryptoMiniSat 5.6.5 + GE	28954.14	116013784
					WDSat	2774.44	483437900
					WDSat + GE	1223.16	34718415
		ANF	30	60	WDSat + XG-EXT	**17.71**	**379346**

enabled SAT solvers, namely MiniSat [8], Glucose [1], Kissat [3], which is the winner in the main track of the latest SAT competition [15] in 2020, and Relaxed_LCMDCBDL_newTech [24], the winner in the main track on satisfiable instances. These solvers take as input a CNF and the number of variables and clauses shown in the table are an average of the 100 instances for the chosen parameters. We note that, for these specific instances, Kissat has the best performance among the non XOR-enabled solvers.

For the XOR-enabled solvers, CryptoMiniSat and WDSat, we tested different versions, specifically to see whether performing GE results in better running times for solving instances derived from the MQ problem. First, we conclude that the best version of WDSat is the one with the XG-EXT technique. Then, we can see that CryptoMiniSat gives better results when the GE is turned on, however, CryptoMiniSat 5.8.0, which is the most recent version seems slower for these instances than CryptoMiniSat 5.6.5. For this reason, we report results for both versions. Note that in CryptoMiniSat 5.8.0 GE is used by default and it is automatically disabled if the solver detects that it performs badly. This version is denoted CryptoMiniSat 5.8.0 in Table 4, whereas CryptoMiniSat 5.8.0 + GE denotes experiments where CryptoMiniSat 5.8.0 is executed with

the option –**autodisablegauss 0**, which ensures that GE is used throughout the entire solving process.

For versions of the solvers that have good performances (namely approaches with GE, plus the simplest version of WDSAT) we were able to increase the parameters and results are shown at the end of Table 4. We conclude that WDSAT outperforms all other solvers for these instances. Finally, to confirm our complexity analysis for the XG-EXT version of WDSAT, we checked the level at which conflicts occur, and found that it is either $\lfloor n-\sqrt{2m} \rfloor$ or $\lceil n-\sqrt{2m} \rceil$, with no exceptions.

Table 5 shows running time comparisons between WDSAT XG-EXT using the original CNF module and WDSAT XG-EXT using the compressed CNF module that we propose in Section 3.1. We conclude that running times are comparable and that replacing the CNF module in WDSAT by our proposed CNF module does not impair the performance of the solver, while allowing us to solve higher degree polynomial systems. For instance, we used the compressed CNF module to solve systems of degree (d) four and the results are shown at the end of Table 5.

Table 5: Comparing WDSAT's original CNF module with our compressed CNF module for solving multivariate polynomial systems.

d	OR-clause size	n	m	#Vars CNF	#Clauses CNF	CNF module	Runtime	#Conflicts
2	3	25	50	325	920	original	0.82	21140
						compressed	0.98	
		30	60	465	1365	original	17.71	379346
						compressed	21.26	
3	4	20	40	1350	5130	original	30.65	57597
						compressed	30.54	
		25	50	2625	10100	original	3413.09	2095437
						compressed	3529.71	
4	5	15	30	1940	8960	compressed	4.86	4333
		18	36	4047	19023	compressed	180.52	39204

5 Conclusion

In this paper, we gave implementation details of the WDSAT solver and showed that it has a broader range of cryptographic applications than the one it was initially designed for. Several cryptographic attacks can be reduced to the problem of solving a Boolean multivariate polynomial system, and when the derived systems are dense, experimental results suggest that WDSAT gives the best performance among state-of-the-art SAT solvers. In addition, our novel CNF module completes WDSAT, so that it can tackle Boolean polynomial systems of any degree. This paper does not alter the overall state-of-the-art, as for MQ instances, algebraic tools are still faster than XOR-enabled SAT solvers.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 399–404 (2009)
2. Bettale, L., Faugère, J., Perret, L.: Hybrid approach for solving multivariate systems over finite fields. J. Mathematical Cryptology **3**(3), 177–197 (2009)
3. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
4. Bouillaguet, C.: LibFES-lite. https://github.com/cbouilla/libfes-lite (2016)
5. Bouillaguet, C., Cheng, C., Chou, T., Niederhagen, R., Yang, B.: Fast exhaustive search for quadratic systems in $\mathbb{F}_2$ on FPGAs. In: Lange, T., Lauter, K.E., Lisonek, P. (eds.) Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8282, pp. 205–222. Springer (2013)
6. Courtois, N., Klimov, A., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: Preneel, B. (ed.) Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding. Lecture Notes in Computer Science, vol. 1807, pp. 392–407. Springer (2000)
7. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing. pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
9. Faugère, J.C.: A new efficient algorithm for computing Gröbner basis (F4). Journal of Pure and Applied Algebra **139**(1-3), 61–88 (1999)
10. Galbraith, S.D., Gebregiyorgis, S.W.: Summation polynomial algorithms for elliptic curves in characteristic two. In: Meier, W., Mukhopadhyay, D. (eds.) Progress in Cryptology - INDOCRYPT 2014 - 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8885, pp. 409–427. Springer (2014)
11. Han, C.S., Jiang, J.H.R.: When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 410–426. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
12. Heule, M., Dufour, M., van Zwieten, J., van Maaren, H.: March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In: Hoos, H.H., Mitchell, D.G. (eds.) Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004. Lecture Notes in Computer Science, vol. 3542, pp. 345–359. Springer (2004)
13. Joux, A., Vitse, V.: A crossbred algorithm for solving boolean polynomial systems. In: Kaczorowski, J., Pieprzyk, J., Pomykala, J. (eds.) Number-Theoretic Methods in Cryptology - First International Conference, NuTMiC 2017, Warsaw, Poland, September 11-13, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10737, pp. 3–21. Springer (2017)

14. Laitinen, T., Junttila, T.A., Niemelä, I.: Conflict-driven XOR-clause learning (extended version). In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7317, pp. 383–396. Springer (2012), http://arxiv.org/abs/1407.6571
15. van Maaren, H., Franco, J.: The International SAT Competition Web Page. http://www.satcompetition.org/, accessed: 2020-05-27
16. Macaulay, F.S.: The Algebraic Theory of Modular Systems. Cambridge Tracts in Mathematics and Mathematical Physics, University Press (1916), https://books.google.fr/books?id=uA7vAAAAMAAJ
17. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. J. Autom. Reasoning **24**(1/2), 165–203 (2000), http://dblp.uni-trier.de/db/journals/jar/jar24.html#MassacciM00
18. McDonald, C., Charnes, C., Pieprzyk, J.: An algebraic analysis of Trivium ciphers based on the boolean satisfiability problem. IACR Cryptol. ePrint Arch. **2007**, 129 (2007), http://eprint.iacr.org/2007/129
19. Soos, M.: Enhanced Gaussian elimination in DPLL-based SAT solvers. In: POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010. EPiC Series in Computing, vol. 8, pp. 2–14. EasyChair (2010)
20. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. pp. 1592–1599. AAAI Press (2019)
21. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 244–257. Springer (2009)
22. Trimoska, M., Ionica, S., Dequen, G.: Parity (XOR) reasoning for the index calculus attack. In: Simonis, H. (ed.) Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12333, pp. 774–790. Springer (2020)
23. Trimoska, M., Ionica, S., Dequen, G.: A SAT-based approach for index calculus on binary elliptic curves. In: Nitaj, A., Youssef, A. (eds.) Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12174, pp. 214–235. Springer (2020)
24. Zhang, X., Cai, S.: Relaxed Backtracking with Rephasing. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 16–17. University of Helsinki (2020)