DPLL+ROBDD Derivation Applied to Inversion of Some Cryptographic Functions

Alexey Ignatiev and Alexander Semenov

Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia
`alexey.ignatiev@gmail.com,biclop@rambler.ru`

Abstract. The paper presents logical derivation algorithms that can be applied to inversion of polynomially computable discrete functions. The proposed approach is based on the fact that it is possible to organize DPLL derivation on a small subset of variables appeared in a CNF which encodes the algorithm computing the function. The experimental results showed that arrays of conflict clauses generated by this mode of derivation, as a rule, have efficient ROBDD representations. This fact is the departing point of development of a hybrid DPLL+ROBDD derivation strategy: derivation techniques for ROBDD representations of conflict databases are the same as those ones in common DPLL (variable assignments and unit propagation). In addition, compact ROBDD representations of the conflict databases can be shared effectively in a distributed computing environment.

1 Introduction

We consider the problem of inverting functions that form a family of type

$$f_n : \{0,1\}^n \to \{0,1\}^*,$$

where $\{0,1\}^n$ is the set of all possible binary sequences of the length n, $n \in N_1$,

$$\{0,1\}^* = \bigcup_{n \in N_1} \{0,1\}^n .$$

Assume that there exists a program M for deterministic Turing machine which computes an arbitrary function f_n of the considered family, and this program is polynomial time. The problem of inverting a function f_n at point $y \in range\ f_n$ is the problem of finding such (an arbitrary) $x \in \{0,1\}^n$ that $f_n(x) = y$.

There exists an effective procedure (polynomial time in n) reducing this problem to SAT problem. With the use of Tseitin transformations [22] this procedure constructs a CNF-encoding of a circuit $S(f_n)$ over $\{\&, \neg\}$ (any other complete basis could be here) which emulates M on all the possible inputs of $\{0,1\}^n$. By $X = \{x_1, \ldots, x_n\}$ we denote a set of Boolean variables corresponding to n

inputs of $S(f_n)$. For each logic gate G some new auxiliary variable $v(G)$ is introduced. Every AND-gate G is encoded by a CNF-representation of a Boolean function $v(G) \leftrightarrow u\&w$. Every NOT-gate G is encoded by a CNF-representation of a Boolean function $v(G) \leftrightarrow \neg u$. Here u and w are the variables corresponding to inputs of G. The CNF-encoding of $S(f_n)$ is

$$\underset{G \in S(f_n)}{\&} C(G),$$

where $C(G)$ is a CNF-encoding of G. Then

$$C_y(f_n) = \left(\underset{G \in S(f_n)}{\&} C(G) \right) \cdot y_1^{\sigma_1} \cdot \ldots \cdot y_m^{\sigma_m}$$

is a CNF encoding the invertion problem of the function f_n at point $y = (\sigma_1, \ldots, \sigma_m)$. Here

$$z^\sigma = \begin{cases} \bar{z}, \text{if } \sigma = 0 \\ z, \text{if } \sigma = 1 \end{cases}$$

and $y_1, \ldots, y_m$ are Boolean variables corresponding to outputs of $S(f_n)$. If $y \in$ *range* f_n then CNF $C_y(f_n)$ is satisfiable, and in any of its satisfying assignments one can find effectively such a vector $x \in \{0,1\}^n$ that $f_n(x) = y$.

It is well-known that while searching for a satisfying assignment for $C_y(f_n)$ it is possible to restrict DPLL derivation to a set of variables denoting an input for $S(f_n)$. We refer to this derivation strategy as "core-DPLL". Along with clause learning and restarts core-DPLL is complete for CNFs which encode the inversion of discrete functions of the class described above.

It is shown in [12] that, generally speaking, core-DPLL cannot polynomially simulate DPLL (even without clause learning and restarts). Our aim is to show that, nevertheless, the use of core-DPLL in inversion of discrete functions provides a number of additional (or rather useful) technical capabilities. In particular, a number of problems difficult for modern DPLL-solvers can though be solved without removing learnt clauses. One can also observe that arrays of conflict clauses learnt by core-DPLL generally have small ROBDD representations (even for CNFs which encode cryptographic algorithms). The size of these ROBDD representations are hundreds of times smaller than the original clauses form. Therefore, it is possible to share effectively arrays of conflict clauses (in the ROBDD form) accumulated at various nodes of distributed computing environments.

A brief outline of the paper is given below. In the first section, we describe basic logical derivation mechanisms combining core-DPLL and a derivation technique for ROBDD representations of conflict databases. The second section describes a parallel implementation of DPLL+ROBDD solver made with the use of MPI. In the third section, we present results of numerical experiments on inversion of some cryptographic functions by the described solver.

2 Basic Mechanisms of DPLL+ROBDD Derivation

Let $C_y(f_n)$ be a CNF encoding the problem of inverting a discrete function f_n (of the class described above) at an arbitrary point $y \in range\ f_n$. In this section we use binary decision diagrams (more precisely, ROBDDs) to represent arrays of conflict clauses accumulated by core-DPLL while finding a satisfying assignment for $C_y(f_n)$. General ideas to represent exhaustive DPLL derivation in the form of binary decision diagrams were considered in [11]. It should be also noted that there are examples of hybrid approaches combining DPLL with BDDs [1,8,4,9]. The methods we suggest here are based on the empirical fact that ROBDDs do compress arrays of conflict clauses learnt during the core-DPLL derivation.

Binary decision diagrams (BDDs) were introduced by C. Y. Lee in the article [14]. The importance of this fundamental data structure for discrete mathematics was realized after R. Bryant's work [3] coming out. In that paper he described a family of algorithms manipulating Boolean functions with the use of BDDs. One of the main theoretical results in [3] is the theorem about canonical representation of Boolean functions in the form of ROBDDs (a ROBDD is a reduced BDD without repeatable fragments). ROBDDs are often able to represent Boolean functions arising in applications in a very compact form.

Next, we will use two algorithms described in [3]. The first one is *Apply* which constructs a ROBDD representation of a function $f_1 * f_2$ using ROBDD representations $B(f_1)$ and $B(f_2)$ of functions f_1 and f_2, where "$*$" is an arbitrary binary logical operation. If the variable orderings in $B(f_1)$ and $B(f_2)$ are identical, then time complexity of *Apply* is $O\left(|B(f_1)| \cdot |B(f_2)|\right)$ (here and below by $|B|$ we denote a number of vertices in B). The second one is *Restrict.* Algorithm *Restrict* takes $(B(f), x, \alpha)$ as an input. Here $B(f)$ is a ROBDD representation of a function f defined by a Boolean formula $L(f)$, x is a variable appeared in $L(f)$ and α is a constant of $\{0, 1\}$. This algorithm produces a ROBDD representation of a function $f|_{x=\alpha}$ defined by a formula $L(f)|_{x=\alpha}$. Time complexity of *Restrict* is $O\left(|B(f)|\right)$.

Consider a CNF

$$C_y(f_n) \cdot D_1\left(x_1^1, \ldots, x_{r_1}^1\right) \cdot \ldots \cdot D_q\left(x_1^q, \ldots, x_{r_q}^q\right),$$

where $D_i\left(x_1^i, \ldots, x_{r_i}^i\right)$, $i \in \{1, \ldots, q\}$ are the conflict clauses learnt during q restarts of core-DPLL for $C_y(f_n)$. Thus,

$$\bigcup_{i=1}^{q} \{x_1^i, \ldots, x_{r_i}^i\} \subseteq X,$$

where $X = \{x_1, \ldots, x_n\}$ is a set of input variables for a circuit $S(f_n)$. Let's denote a ROBDD representation of a function defined by the formula

$$D_1\left(x_1^1, \ldots, x_{r_1}^1\right) \cdot \ldots \cdot D_q\left(x_1^q, \ldots, x_{r_q}^q\right) \qquad (1)$$

as B^*. We have the following fact.

Theorem 1. *Let $x \in \{0,1\}^n$ be a solution of the inversion problem for f_n at some point $y \in range\ f_n$. Then there exists such a path π in B^* from the root to the terminal "1", that $x \in A(\pi)$, where $A(\pi)$ is a subset of $\{0,1\}^n$ specified by π.*

Proof sketch. Let $x \in \{0,1\}^n$ be an arbitrary solution of the inversion problem considered. Suppose that there is no such a path from the root of B^* to "1", which contains x. Therefore, if we substitute x into (1) we get 0. Note that each clause $D_i\left(x_1^i, \ldots, x_{r_i}^i\right)$, $i \in \{1, \ldots, q\}$, is a logical consequence of CNF $C_y(f_n)$. However, if we substitute x into $C_y(f_n)$ then the satisfying assignment for $C_y(f_n)$ results from unit propagation [6]. Thereby, CNF $C_y(f_n)$ is made true by some assignment and CNF (1) (which is a logical consequence of $C_y(f_n)$) is made false by the same assignment. This contradicts our assumption, so we are forced to conclude that there is a path from the root of B^* to "1", which contains x. $\square$

This theorem provides a basis for the general hybrid DPLL+ROBDD derivation strategy considered below. During the derivation process a ROBDD representation of conflict databases is regarded as a formula. Therefore, one can assign some variables in the ROBDD, and certain variables can be implied from a unit propagation similarity. Just as in DPLL, the result of every conflict is some conflict clause learnt. In our case, every conflict clause contains only literals over a set of input variables for a function. The resulting conflict clauses are added to the ROBDD representation of a conflict database using *Apply* procedure. Let $B(f)$ be a ROBDD representation of an arbitrary Boolean function $f(x_1, \ldots, x_n)$. Each path from the root of $B(f)$ to a terminal vertex defines a family of sets of truth values for $x_1, \ldots, x_n$.

Let's put in correspondence each variable x_i, $i \in \{1, \ldots, n\}$, and terminal vertex "0" with a set of the variable's truth values defined by all the paths in $B(f)$ from the root to "0". We denote this set by $\Delta^0(x_i)$. One can define $\Delta^1(x_i)$ in a similar manner.

Suppose, that in ROBDD $B(f)$ the following conditions for a variable $x_k \in X$, $X = \{x_1, \ldots, x_n\}$, hold:

1. Every path π in $B(f)$ from the root to "1" passes through a vertex marked by x_k.
2. $|\Delta^1(x_k)| = 1$.

Then variable x_k may take on exactly one value (the value of $\Delta^1(x_k)$) in any truth assignment over X that makes f assign true.

Definition 1. *The situation defined by conditions 1–2 is called a ROBDD-based consequence of a value of variable x_k.*

A ROBDD-based consequence of some variable in $B(f)$ presenting an array of conflict clauses is a similarity of unit propagation used in DPLL derivation. Further we make use of a modified version of *Restrict* which could assign a set of variables of X into $B(f)$ at the same time. As noted by R. Bryant in [3], time complexity of this algorithm is the same as time complexity of the original

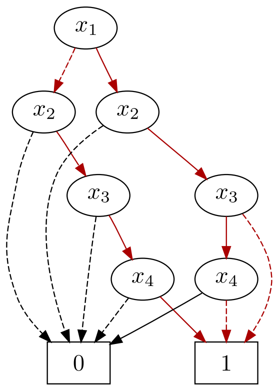


Fig. 1. ROBDD representation of a function $x_2 \cdot (x_1 \oplus x_3 \cdot x_4)$ using the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$. We have a ROBDD-based consequence of variable x_2 ($x_2 = 1$) here because each path from the root to "1" passes through a vertex marked by x_2 and $|\Delta^1(x_2)| = 1$.

Restrict, i. e. $O\left(|B(f)|\right)$. The basic idea of the procedure described below was proposed in [5]. However, the authors of that paper did not estimate its time complexity.

Theorem 2. *For a ROBDD $B(f)$ and the values $x_{i_1} = \alpha_{i_1}, \ldots, x_{i_m} = \alpha_{i_m}$, $m \leq n$, $\alpha_{i_j} \in \{0, 1\}$, $j \in \{1, \ldots, m\}$, time complexity of the procedure which substitutes given values into $B(f)$ and checks for ROBDD-based consequences of other variables is $O\left(|B(f)|\right)$.*

Proof sketch. Let's substitute $x_{i_1} = \alpha_{i_1}, \ldots, x_{i_m} = \alpha_{i_m}$ into $B(f)$. As it was said above, this process takes time bounded by $O(|B|)$. After making the substitutions we check for ROBDD-based consequences. Note that ROBDD-based consequence of some variable x_k results in exactly one of the following:

1. each vertex marked by x_k has "0" as the high-child;
2. each vertex marked by x_k has "0" as the low-child.

Therefore, we have a ROBDD-based consequence of $x_i = 1$ if and only if each vertex marked by x_i has "0" as the low-child and every path from "1" to the root passes through a vertex marked by x_i.

Using this fact, we go from "1" towards the root of the ROBDD. Let $V(1)$ be a set containing parents of "1". We also denote a set of variables marking vertices of $V(1)$ by $X(1) = \{x_{i_1}, \ldots, x_{i_r}\}$. We can choose from $V(1)$ all the vertices marked by such a variable x_{i_*} that $x_{i_*} \prec x_j \ \forall j \in \{i_1, \ldots, i_r\} \setminus \{i_*\}$ (according to the variable ordering in the ROBDD). Variable x_{i_*} is referred to as a minimal variable in $X(1)$ with respect to the variable ordering. It is obvious that for any variable of $X(1) \setminus \{x_{i_*}\}$ a ROBDD-based consequence is not possible. By $\tilde{V}(1)$ we denote a set of all vertices marked by variables of $X(1) \setminus \{x_{i_*}\}$. Next, move up from each vertex in $\tilde{V}(1)$ toward the root of the

ROBDD until the first vertex marked by variable x_k appears, such that either $x_k = x_{i_*}$, or $x_k \prec x_{i_*}$. A set of the ROBDD vertices generated in this sense by set $V(1)$ is denoted by $V(x_{i_*})$, and a set of variables to mark vertices of $V(x_{i_*})$ is denoted by $X(x_{i_*}) = \{x_{k_1}, \ldots, x_{k_s}\}$. If $x_{k_1} = \ldots = x_{k_s} = x_{i_*}$, then we check for each vertex of $V(x_{i_*})$ whether its low-child (or high-child) is "0". If yes, then we have a ROBDD-consequence of variable x_{i_*}. If not, then we should go on the procedure. It is not difficult to understand that the described algorithm finds all the possible ROBDD-consequences in one pass through the ROBDD. Hence, time complexity of the procedure which makes substitutions into $B(f)$ and checks for every possible ROBDD-based consequence is $O\left(|B(f)|\right)$. $\square$

This theorem implies the next corollary.

Theorem 3. *If some substitution into $B(f)$ implies a ROBDD-based consequence of $x_k = \alpha_k$, $\alpha_k \in \{0,1\}$ for some $x_k \in X$, then substitution of $x_k = \alpha_k$ in $B(f)$ cannot imply another ROBDD-based consequence.*

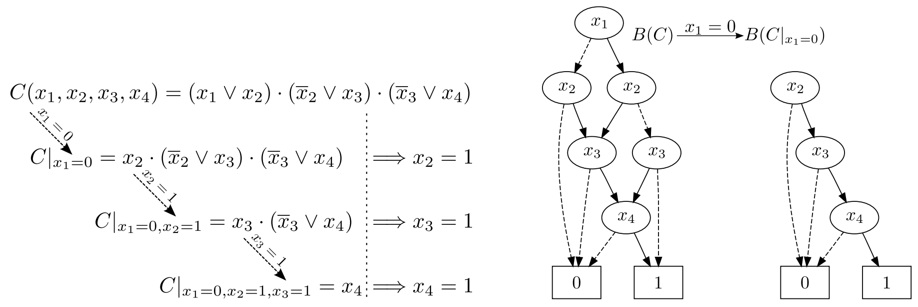
Proof sketch. Suppose, that some substitution into $B(f)$ implies a ROBDD-based consequence $x_k = \alpha_k$. Assume without loss of generality that $\alpha_k = 1$. In accordance with the above (see the first paragraph of theorem's 2 proof) this assumption means that the low-child of each vertex marked by x_k is "0". Substitution of $x_k = 1$ in $B(f)$ means that each vertex $u(x_k)$ hands over its high-child to its parents. However, the low-child of $u(x_k)$, that is, the terminal "0", is not handed over to any vertex. Thus, substitution of $x_k = 1$ into $B(f)$ cannot cause such a vertex in $B(f)$ *to appear*, that some of its children is "0" (but it does not mean that there is no such a vertex before the substitution). Similar arguments hold if $\alpha_k = 0$. $\square$

This fact shows a very useful feature of ROBDD considered as an array of Boolean constraints. It's known that substituting a variable's value into a CNF may lead to a situation where unit clause rule can be used several times. The procedure implementing iterative unit clause rule is the so-called Boolean constraint propagation (BCP). In the general case, BCP passes through the CNF many times. The obtained feature of ROBDDs means that ROBDD-based consequences implied by an arbitrary substitution cannot imply a new ROBDD-based consequence and, therefore, all the information implied by the substitution comes out as a result of a single pass through a ROBDD (see Fig. 2).

Another positive property of the hybrid derivation is the possibility to easily implement lazy computations (an analogue of well-known data structures used in BCP, i. e. "watched literals", [15]) using ROBDDs.

Let's consider the conditions determining a situation which in some sense is ambivalent to a ROBDD-based consequence.

3. For a variable $x_q \in X$, $X = \{x_1, \ldots, x_n\}$, every path π in $B(f)$ from the root to "0" passes through a vertex marked by x_q.
4. $|\Delta^0(x_q)| = 1$.


$C(x_1,x_2,x_3,x_4) = (x_1 \vee x_2)\cdot(\overline{x}_2 \vee x_3)\cdot(\overline{x}_3 \vee x_4)$
$x_1 = 0$
$C|_{x_1=0} = x_2 \cdot (\overline{x}_2 \vee x_3)\cdot(\overline{x}_3 \vee x_4) \Longrightarrow x_2 = 1$
$x_2 = 1$
$C|_{x_1=0,x_2=1} = x_3 \cdot (\overline{x}_3 \vee x_4) \Longrightarrow x_3 = 1$
$x_3 = 1$
$C|_{x_1=0,x_2=1,x_3=1} = x_4 \Longrightarrow x_4 = 1$
$B(C) \xrightarrow{x_1 = 0} B(C|_{x_1=0})$
x_1
x_2
x_3
x_4
0
1

(a) BCP (b) ROBDD-case

Fig. 2. On the left we show the BCP process in CNF $(x_1 \vee x_2)\cdot(\overline{x}_2 \vee x_3)\cdot(\overline{x}_3 \vee x_4)$ started by assigning $x_1 = 0$; on the right we demonstrate the result of substituting $x_1 = 0$ into ROBDD representation of the considered CNF — here a single pass through the ROBDD is required.

Theorem 4. *Let $B(f)$ be an arbitrary ROBDD and there be such a variable x_q in $B(f)$ so that conditions 3–4 hold for x_q. Then there are no possible ROBDD-based consequences of any variable from $X \setminus \{x_q\}$ in $B(f)$. Time complexity of procedure which checks whether conditions 3–4 hold is $O(|B(f)|)$.*

Proof sketch. Let conditions 3–4 hold for some variable $x_q \in X$ in $B(f)$. Assume without loss of generality that $\Delta^0(x_q) = \{1\}$. By analogy with the proof of theorem 2 the assumption means that each vertex marked by x_q has "1" as the low-child.

Let $x_p \in X \setminus \{x_q\}$ be an arbitrary variable. There are two possible alternatives for its location relative to x_q with respect to the variable ordering in $B(f)$ (variable x_1 marks the root of $B(f)$):

$$1 : x_1 \prec \ldots \prec x_q \prec \ldots \prec x_p \prec \ldots$$
$$2 : x_1 \prec \ldots \prec x_p \prec \ldots \prec x_q \prec \ldots$$

Consider the first case. As it was said above, the low-child of each vertex marked by x_q is the terminal "1". This means that there is such a path from the root of $B(f)$ to "1" that does not pass through vertices marked by x_p. In other words, the ROBDD-based consequence of x_p is not possible.

Consider the second case. Assume there is a ROBDD-based consequence of x_p in $B(f)$, i.e. conditions 1–2 hold for x_p. Then one of the children of each vertex marked by x_p is "0". However, this means that there are such paths from vertices marked by x_p to "0" which do not pass through vertices marked by x_q. This contradicts the fact that conditions 3–4 hold for x_q.

It is not difficult to understand that validity of conditions 3–4 can be checked by a procedure which is similar to the procedure described in the proof of theorem 2 and has the same time complexity — $O(|B(f)|)$. □

This theorem provides a possibility to formulate mechanisms of lazy computations while assigning variables implied during the hybrid derivation process.

If conditions 3–4 hold for some x_q in a ROBDD B^*, and a value of x_k, $k \neq q$, is derived from a CNF, then it is not necessary to substitute this value into B^* because no new ROBDD-based consequences will be implied. It is sensible to store up all the variables to assign until the moment of assigning x_q and after that to substitute them all into B^* at the same time checking every possible ROBDD-based consequence (see theorem 2).

3 Parallel DPLL+ROBDD Solver Sharing Arrays of Conflict Clauses in the ROBDD Form

As already mentioned, in practice even for hard cryptographic tests core-DPLL generates conflict databases which have compact ROBDD representations (one can use the variable ordering defined by a current state of accumulated variable activities [17]). This fact leads us to an idea of a parallel solver to accumulate arrays of conflict clauses in the ROBDD form at different computing nodes and to share them effectively between the nodes. It is a small size of a ROBDD representation of conflict clauses that provides the efficiency.

In more detail, the solver consists of two components. A core-DPLL component is implemented as a modification of MiniSat-C v1.14.1 [7] named "coresat". Conflict analysis made by coresat uses information only on those function's input variables which are responsible for a conflict. It is based on the use of characteristic vectors (this technique is similar to the one proposed in [13]). As a result, there is no need to use an implication graph [16] to determine a reason for the conflict. Another solver's component encloses the process constructing ROBDD representations of conflict databases and derivation procedures for ROBDDs based on the algorithms described above.

The interaction between core-DPLL and ROBDD components of the hybrid solver is implemented in compliance with the schema shown in Fig. 3.

Under this schema, the hybrid DPLL+ROBDD solver is an iterated procedure determined by the actions listed below.

1. At the initial stage only coresat operates. The result is an array of learnt conflict clauses, each contains literals over a set of input variables for the function.
2. The solver suspends coresat and starts to construct a ROBDD representation of the array of conflict clauses learnt during the first step (for this purpose we use algorithm *Apply* by R. Bryant). It is reasonable to use the variable ordering defined by variable activities which were accumulated by this moment.
3. The result of each iteration is a new ROBDD obtained using *Apply* to a previous one and the ROBDD representation of the conflict database constructed during the current iteration (see step 2). The variable ordering can differ in the two ROBDDs. Therefore, before running *Apply* we need to reorder the old ROBDD according to the new variable ordering.
4. The process continues iteratively and is terminated if a satisfying assignment is found or it is proven that the CNF instance is unsatisfiable.

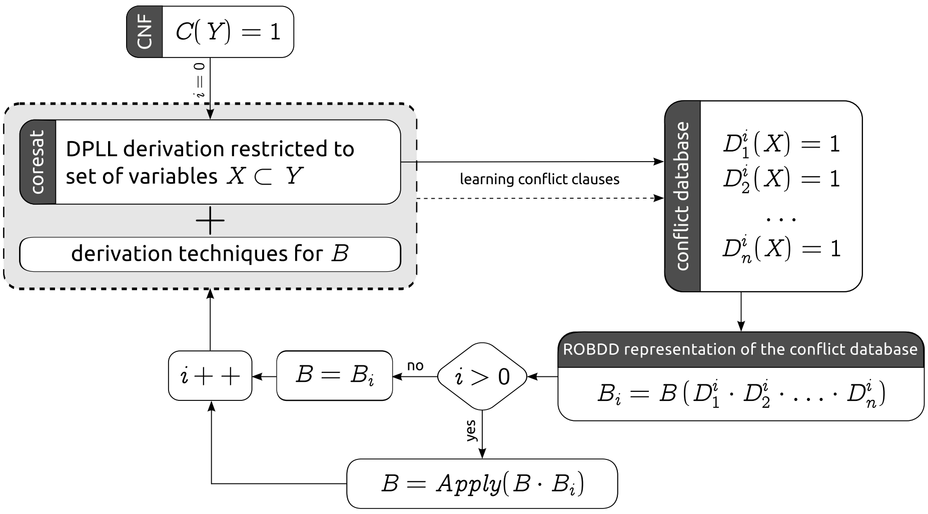

CNF
$C(Y) = 1$
$i = 0$
coresat
DPLL derivation restricted to set of variables $X \subset Y$
+
derivation techniques for B
learning conflict clauses
conflict database
$D_1^i(X) = 1$
$D_2^i(X) = 1$
...
$D_n^i(X) = 1$
ROBDD representation of the conflict database
$B_i = B(D_1^i \cdot D_2^i \cdot \ldots \cdot D_n^i)$
$i > 0$
no
yes
$B = B_i$
$i{+}{+}$
$B = Apply(B \cdot B_i)$

Fig. 3. Schema of the hybrid DPLL+ROBDD solver

A sequential variant of the hybrid solver is referred to as "hsat". A parallel version of the hybrid solver (we name it "mhsat") is implemented as an MPI application and is a bunch of hsat instances, which work simultaneously and periodically share their conflict databases in the ROBDD form. To ensure that hsat instances start to solve the problem differently from each other, we choose unique initial variable activities for each of them.

Operating of mhsat can be seen as a serial implementation of the following steps:

1. The stage of accumulating conflict clauses in the ROBDD form. Each node generates conflict clauses irrespective of each other and constructs its local ROBDD in accordance with its current variable activities.
2. The stage of merging accumulated conflict databases. There is a number of alternatives on how to make this step. Here we describe the simplest one:
   (a) Exchanging local variable activities to construct the common variable ordering;
   (b) Reconstructing each local ROBDD according to the common variable ordering;
   (c) Exchanging conflict databases and joining them (we use recursive doubling [21] and *Apply* for this purpose). The result of this stage is a final ROBDD which is constructed on some computing node and represents the complete array of conflict clauses with respect to the common variable ordering;
   (d) Sharing the final ROBDD to all the other nodes;
   (e) Reconstructing the final ROBDD according to a local variable ordering on each of the computing nodes.

It should be noted that joining the ROBDDs with the use of *Apply* is optional. Instead it is possible to make each node store local copies of all the ROBDDs made by other nodes. In this case, each of the nodes has an array of the ROBDDs and runs a derivation process for all the ROBDDs separately. Such approach can improve the solver's performance when the serial use of *Apply* leads to an exponential growth of the output ROBDD's size.

4 Experimental Results

We experimented on CNFs which encode a cryptanalysis of the weakened keystream generator used in the cipher A5/1. This generator is used to encrypt the traffic in GSM networks. The authors of [19] minutely described procedures for constructing a CNF encoding cryptanalysis of the generator A5/1. They also presented results on coarse-grained approach to logical cryptanalysis of the generator in a Grid system. This approach is based on the technology of decomposition of a SAT problem encoding the generator algorithm into a family of SAT problems of lower dimension. By $C(A5/1)$ we denote the CNF encoding the algorithm of the generator A5/1, and by $X(A5/1)$ we denote the set of Boolean variables appeared in $C(A5/1)$. In accordance with the technique described in [19], from $X(A5/1)$ one can select a subset of Boolean variables, each corresponds to initial contents of a cell of a register of the generator. Cardinality of this subset is d, $d \leq 64$. This set is called a decomposition set and denoted by X_d. Substituting all possible truth values for variables of X_d in $C(A5/1)$

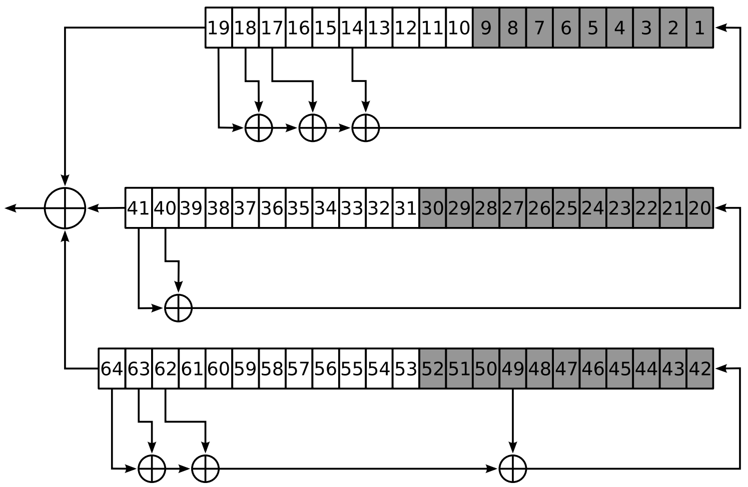


Fig. 4. Schema of the A5/1 keystream generator which consists of 3 LFSRs, given by the following connection polynomials over GF(2): LFSR 1: $X^{19}+X^{18}+X^{17}+X^{14}+1$; LFSR 2: $X^{22}+X^{21}+1$; LFSR 3: $X^{23}+X^{22}+X^{21}+X^{8}+1$. The algorithm of A5/1 keystream generator is encoded by CNF in accordance with the technique described in [19].

generates a decomposition family consisting of 2^d CNFs. This family forms a parallel task list that can be processed in a distributed computing environment. Inter-processor communications are extremely rare here.

The coarse-grained approach shows the best results in the case of decomposing by 31 variables. In Fig. 4 shown below the cells corresponding to this set of 31 variables are dark shaded.

In our experiments we used the decomposion set X_{20} shown in Fig. 5. Substituting all possible truth values for variables of X_{20} in $C(A5/1)$ generates a decomposition family consisting of 2^{20} CNFs. As the test material we considered 50 CNFs, chosen randomly from this decomposition family. All selected in such a way CNFs were unsatisfiable. Tests were run on a platform of Intel Xeon E5345 (4 cores, 2.33 GHz), 8 GB RAM. To evaluate efficiency of the hybrid DPLL+ROBDD derivation we used approaches listed below:

1. Coarse-grained parallelization without sharing clauses. For each of the fifty CNFs we constructed 4 simpler CNFs obtained by substituting all the possible values of two variables x_{23} and x_{45} into the original one. Thus, each of the 4 CPU cores solved its own fifty SAT problems irrespective of other cores. In this series of experiments we used the following solvers: hsat, dminisat [19] and MiniSat 2.2.0 [7].
2. The use of solvers with parallel architecture. In this series of experiments there were involved multi-threaded solvers MiraXT 1.1 [18] and ManySAT 1.1 [10], as well as mhsat, which is an MPI application.
3. Sequential solving all the considered tests by hsat using one CPU core.

We emphasize that the original versions of ManySAT and MiniSat cannot cope with tests of the set under consideration. But it is possible to solve this problem by assigning nonzero values to initial activity for those variables which correspond

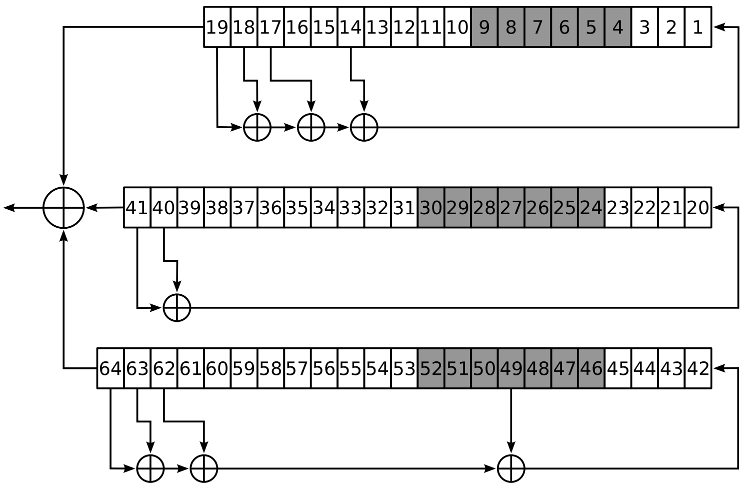


Fig. 5. Schema of the decomposition set X_{20}

Table 1. Average solving time for each of the solvers

place	solver	mode of operating	number of cores	avg. time (seconds)
1	mhsat	parallel	4	569.016
2	hsat	coarse-grained	4	644.254
3	MiraXT (mod)	parallel	4	1639.192
4	hsat	sequential	1	2385.578
5	dminisat	coarse-grained	4	2750.486
6	MiraXT (orig)	parallel	4	3214.178
7	ManySAT (mod)	parallel	4	3378.078
8	MiniSat (mod)	coarse-grained	4	5836.782

to initial contents of A5/1's registers (in Table 1 this modification is denoted by "mod"). In contrast to ManySAT and MiniSat even the original version of MiraXT can handle the considered tests. However, increasing initial activity of the same variables doubles its performance on average.

Note the fact that mhsat taking 4 CPU cores is more than 4 times faster than its sequential version (hsat).

In addition to the parallel solvers listed above, we tried to use the well-known solvers CryptoMiniSat 2.9.0 [20] and Plingeling 276 [2]. However, these solvers could not cope with the tests in a reasonable time.

5 Conclusions and Future Work

According to the experimental results we can conclude that the hybrid DPLL+ +ROBDD derivation techniques described in the paper may be useful in solving the function inversion problems that are difficult for the solvers performed better on the well-known test libraries.

We suppose that our hybrid methods have potential to be heavily improved. In particular, some improvements of the basic hsat's algorithms are expected in the near future. In addition, we also project to analyze various alternatives on inter-process sharing the arrays of conflict clauses generated by different nodes of a large-scale distributed computing environment.

Despite the interesting experimental results we realize that they are not enough to justify the efficiency of our approach to a wide class of functions. Therefore, we hope to succeed in expanding the class of tests, which can be solved by the described algorithms much more efficiently in comparison with traditional DPLL-based derivation methods.

Acknowledgements

The authors would like to thank Alexei Hmelnov,a Dmitry Bespalov and Stepan Kochemazov (ISDCT SB RAS) for their help and numerous valuable advices.

This work is supported by Russian Foundation for Basic Research (Grant No. 11-07-00377-a).

References

1. Aloul, F.A., Mneimneh, M.N., Sakallah, K.A.: ZBDD-Based Backtrack Search SAT Solver. In: Proceedings of International Workshop on Logic and Synthesis (IWLS), pp. 131–136 (2002)
2. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Tech. Rep. 10/1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2010)
3. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
4. Chatalic, P., Simon, L.: Zres: The old Davis-Putnam procedure meets ZBDDs. In: McAllester, D. (ed.) CADE 2000. LNCS(LNAI), vol. 1831, pp. 449–454. Springer, Heidelberg (2000)
5. Damiano, R.F., Kukula, J.H.: Checking satisfiability of a conjunction of BDDs. In: 40th Design Automation Conference, DAC 2003, pp. 818–823 (2003)
6. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. The Journal of Logic Programming 1(3), 267–284 (1984)
7. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
8. Ganai, M., Gupta, A.: SAT-Based Scalable Formal Verification Solutions. Series on Integrated Circuits and Systems. Springer-Verlag New York, Inc., Secaucus (2007)
9. Gopalakrishnan, S., Durairaj, V., Kalla, P.: Integrating CNF and BDD based SAT solvers. In: IEEE International High-Level Design, Validation, and Test Workshop, pp. 51–56 (2003)
10. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a Parallel SAT Solver. Journal on Satisfiability, Boolean Modeling and Computation, Special Issue on Parallel SAT Solving 6, 245–262 (2009)
11. Huang, J., Darwiche, A.: The Language of Search. Journal of Artificial Intelligence Research 29, 191–219 (2007)
12. Järvisalo, M., Junttila, T.: Limitations of restricted branching in clause learning. Constraints 14(3), 325–356 (2009)
13. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. IEEE Transactions on Computer-Aided Design 21(12), 1377–1394 (2002)
14. Lee, C.Y.: Representation of Switching Circuits by Binary-Decision Programs. Bell Systems Technical Journal 38, 985–999 (1959)
15. Lynce, I., Marques-Silva, J.: Efficient data structures for backtrack search SAT solvers. Annals of Mathematics and Artificial Intelligence 43(1), 137–152 (2005)
16. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
17. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Annual Design Automation Conference, DAC 2001, pp. 530–535. ACM, New York (2001)
18. Schubert, T., Lewis, M., Becker, B.: PaMiraXT: Parallel SAT Solving with Threads and Message Passing. Journal on Satisfiability, Boolean Modeling and Computation, Special Issue on Parallel SAT Solving 6, 203–222 (2009)

19. Semenov, A., Zaikin, O., Bespalov, D., Posypkin, M.: Parallel algorithms for SAT in application to inversion problems of some discrete functions, arXiv:1102.3563v1 [cs.DC]
20. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009)
21. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of Collective Communication Operations in MPICH. Int'l Journal of High Performance Computing Applications 19(1), 49–66 (2005)
22. Tseitin, G.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic 2, 234–259 (1968)