Combining CDCL, Gauss-Jordan Elimination, and Proof Generation

Mate Soos and Randal Bryant

July 19, 2022

Proof Generation for CDCL Solvers Using Gauss-Jordan Elimination

Mate Soos[1] and Randal E. Bryant[2] *

[1] National University of Singapore
soos.mate@gmail.com
[2] Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Randy.Bryant@cs.cmu.edu

Abstract

Traditional Boolean satisfiability (SAT) solvers based on the conflict-driven clause-learning (CDCL) framework fare poorly on formulas involving large numbers of parity constraints. The CryptoMiniSat solver augments CDCL with Gauss-Jordan elimination to greatly improve performance on these formulas. Integrating the tbuddy proof-generating BDD library into CryptoMiniSat enables it to generate unsatisfiability proofs when using Gauss-Jordan elimination. These proofs are compatible with standard, clausal proof frameworks.

1 Introduction

Consider Boolean formulas over a set of variables $X = \{x_1, x_2, \ldots, x_n\}$. A k-way *parity constraint*, also known as an *XOR constraint*, is an equation of the form:

$$x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_k} \quad = \quad p \tag{1}$$

where the phase p can be 1 (odd parity) or 0 (even parity.)

Although Boolean satisfiability (SAT) solvers based on conflict-driven clause learning (CDCL) have made steady improvements over the years, they fare poorly when the formula contains large numbers of parity constraints.

As an example, Urquhart devised a family of unsatisfiable formulas, consisting entirely of parity constraints, where both the number of variables and the number of clauses scale quadratically with the size parameter m, but any resolution proof of unsatisfiability must scale exponentially in m [27]. The smallest instance of this benchmark, having $m = 3$, consists of 153 variables and 408 clauses, encoding 102 three-way parity constraints. Kissat, a state-of-the-art CDCL solver [3], fails to terminate after running on this formula for 16 hours, even with proof generation disabled. On the other hand, by viewing constraints of the form of Equation 1 as linear equations over integers modulo 2, applying Gaussian elimination to an unsatisfiable set of parity constraints yields the infeasible equation $0 = 1$ in polynomial time.

Several CDCL solvers have been augmented with constraint solvers that can apply Gauss-Jordan elimination to parity constraints [12,13,17,24]. These solvers combine traditional clausal reasoning with parity reasoning to improve their performance on both satisfiable and unsatisfiable formulas. They operate by first detecting the parity constraints encoded in the input formula and delegating the parity constraints to the parity reasoning component, while retaining

*Supported by the U. S. National Science Foundation under grant CCF-2108521

the remaining clauses for the clausal reasoning component. During execution, the two components reason about their respective portions of the formula and coordinate via unit propagation and conflict detection. This mechanism is sometimes referred to as “CDCL(T),” reflecting its similarity to the handling of multiple theories by SMT solvers [9]. This approach can be extremely effective. For example, version 5.8.0 of CryptoMiniSat running on the Urquhart formula with $m = 3$ detects that the formula is unsatisfiable in just one second. Even scaling to $m = 40$, a formula that is over 200 times larger, with 33,120 variables and 88,320 clauses, the program can detect that the formula is unsatisfiable in less than five minutes.

1.1 Proofs of Unsatisfiability for SAT Solvers

Recent generations of CDCL SAT solvers can produce a proof of unsatisfiability when they encounter an unsatisfiable formula. Such a proof provides an independently checkable confirmation that the formula is truly unsatisfiable. Since 2016, entrants in the main track of the annual SAT competition receive credit for an unsatisfiable result only when 1) they produce a DRAT proof [28], and 2) this proof is successfully checked by a standard proof checker. Unsatisfiability proofs also ensure the integrity of the formal verification tools and mathematical proofs that employ SAT solving. This is especially important when the results are intended to provide assurances in high risk environments, such as aerospace, transportation, and cybersecurity.

Until now, generating clausal proofs while employing parity reasoning has been a major challenge. Standard clausal proof frameworks, such as DRAT, are well matched to CDCL solvers, but there has been limited success generating DRAT proofs while employing Gaussian or Gauss-Jordan elimination. As a result, parity reasoning has not been used by entrants in the main track of the SAT competitions in recent years.[1]

Given their inability to generate clausal proofs when using Gauss-Jordan elimination, most current SAT solvers disable parity reasoning when they are directed to produce proofs and instead rely purely on CDCL. In this mode, they fare no better than kissat on formulas containing parity constraints, including the Urquhart formulas.

1.2 Related Work

Some have proposed the adoption of other proof frameworks to enable proof generation by parity constraint solvers. For example, the VeriPB proof checker verifies proofs expressed in a logic of pseudo-Boolean constraints [11]. Using this framework as a target, Gocht and Nördstrom were able to integrate a Gauss-Jordan elimination solver into the MiniSat solver and have it generate proofs of unsatisfiability [12,23]. In a similar vein, Barnett and Biere proposed a proof framework based on binary decision diagrams (BDDs), where each proof step is guaranteed to be checkable with polynomial complexity [2]. Their framework could also enable proof generation from parity constraints. Getting the SAT community to adopt a new proof framework would require establishing and documenting a new set of file formats and creating a collection of proof checkers, preferably including ones that have been formally verified. Our approach has the advantage that it builds on the well-established DRAT framework.

In prior work, we devised a fundamentally different approach for generating unsatisfiability proofs when reasoning about *pseudo-Boolean* formulas, of which parity constraints are a special case [5]. We do so by combining Gaussian elimination with a proof generator based on BDDs. BDDs are especially effective for reasoning about parity constraints—the BDD representation

[1]The “nolimits” track allows for solvers to state that a formula is unsatisfiable without proof, and hence solvers with Gauss-Jordan elimination have continued to participate in this track.

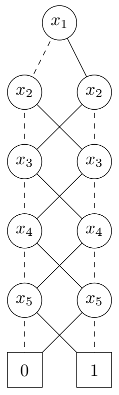

Figure 1: BDD Representation of a 5-variable parity constraint. The representation is linear in the number of variables for any ordering of the BDD variables.

of a k-way constraint has only $2k-1$ nonterminal nodes regardless of the BDD variable ordering. As an example, Figure 1 shows the BDD representation of the parity constraint

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \quad = \quad 1$$

The proof generator uses the ability to introduce extension variables into DRAT proofs, in the style of extended resolution [16, 26]. An extension variable is added for each BDD node generated, describing the logical relation between the node and its associated variable and children nodes [6,15,22]. Each recursive step of a BDD operation is justified by a short sequence of DRAT proof steps, with the effect that, when the Gaussian elimination solver detects an infeasible parity constraint, the overall proof consists of a sequence of implication-preserving clauses terminating with the empty clause. On the other hand, our BDD-based solver does not do well for many of the formulas that are readily handled by CDCL solvers, and a third class of problems requires both clausal and parity reasoning to detect that a formula is unsatisfiable. Integrating the proof generating capabilities of BDDs into a CDCL(T) solver expands the range of formulas for which it can generate unsatisfiability proofs.

Philip and Rebola-Pardo devised a method to generate DRAT proofs when manipulating parity constraints [21]. Their approach maintains clausal representations of parity constraints, using auxiliary variables to ensure that the representation of a k-ary constraint uses $O(k)$ clauses. With each step of a constraint solver, their method generates a proof that the clausal encoding of the newly generated constraint is logically implied by the encodings of the argument constraints. The auxiliary variables in the new constraint become extension variables in the proof. Operationally, this approach is similar to proof-generating BDD-based solvers, where extension variables are added for every BDD node generated. Their work is specialized to parity constraints, whereas ours can be applied to other classes of pseudo-Boolean constraints [5]. Gocht and Nördstrom incorporated this method into their extended version of MiniSat [12]. Although they were able to generate proofs for benchmark circuits that would be intractable for pure CDCL solvers, they found that the checking time did not scale very well. We speculate that this is due to the poor performance of the standard checker drat-trim [28] on formulas with large numbers of extension variables, a shortcoming that we also experienced.

1.3 Contributions

In this work, we describe our recent experience integrating the proof-generating capabilities of BDDs into the Gauss-Jordan (G-J) solver component of CryptoMiniSat, a highly tuned CDCL(T) solver. We do so using tbuddy [4], a proof-generating BDD library that was implemented by extending the buddy [19] BDD package.[2] Tbuddy generates proof steps to justify the unit propagations and conflicts inferred during G-J elimination. This capability was added via a loose integration, with only one new data structure added to CryptoMiniSat.

A bigger challenge was in generating proofs that can be checked efficiently. Drat-trim, the standard checker for DRAT proofs [28], cannot handle the large proofs generated by tbuddy, and especially their heavy use of extension variables. Fortunately, we were able to achieve acceptable performance with a checker for the hybrid FRAT proof format [1], where many of the proof steps, including all steps generated by tbuddy, are annotated with hints, greatly simplifying the operation of the checker. Although FRAT proofs use a different file format than the standard DRAT format, the associated checker frat-rs can generate detailed proofs in the same LRAT format [14] as does drat-trim. In this way, both checkers can make use of the variety of available LRAT proof checkers, including several that have been formally verified [14, 25].

This paper describes the integration of tbuddy into CryptoMiniSat and provides an experimental evaluation using two families of benchmark problems.

2 Solver/Prover Integration

For parity constraint P having the form shown in Equation 1, let $\mathsf{Var}(P) = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ consist of the variables in the constraint and $\mathsf{Phase}(P) = p$ be the phase. The sum of constraints P_a and P_b is also a parity constraint P_c, written as $P_c = P_a \oplus P_b$, with $\mathsf{Var}(P_c) = \mathsf{Var}(P_a) \triangle \mathsf{Var}(P_b)$ (their symmetric difference) and $\mathsf{Phase}(P_c) = \mathsf{Phase}(P_a) \oplus \mathsf{Phase}(P_b)$.

Let ϕ denote the set of input clauses and $P_1, P_2, \ldots, P_m$ denote the set of parity constraints embedded in the input formula. For $S \subseteq \{1, 2, \ldots, m\}$, let $\Sigma(S)$ denote the parity constraint formed by summing all constraints P_i such that $i \in S$. A *literal* ℓ is either a variable or its complement, with its *phase* being 1 in the former case and 0 in the latter.

When integrated into a CDCL framework, Gauss-Jordan elimination provides a way to systematically sum parity constraints and thereby detect conflicts and unit propagations implied by the constraints [13, 17]. Suppose that the state at some point in the search is characterized by a *trace* given by as a conjunction of literals $\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_k$. *A conflict* arises when there is a sum $\Sigma(S)$ such that 1) the variables in $\Sigma(S)$ are a subset of the variables occurring in the trace, and 2) the phase of $\Sigma(S)$ does not match what would be obtained by summing the phases of the corresponding trace literals. A *unit propagation* arises if all but one of the variables in $\Sigma(S)$ occurs in the trace, and the implied literal is either this variable or its complement, such that the sum of the phases of the literals matches that of the constraint. Both of these inferences can be characterized by a *reason* clause C. For a conflict, C is the *conflict clause*, consisting of the complements of some subset of the literals in the trace. In other words, satisfying ϕ requires contradicting at least one assignment in the trace. For unit propagation, C consists of the complements of some of the trace literals plus the literal being inferred by unit propagation. In other words, any satisfying assignment to ϕ compatible with the trace must also satisfy the implied literal.

[2]Tbuddy is available at https://github.com/rebryant/tbuddy-artifact.

Adding proof generation for parity constraints to CryptoMiniSat requires three new capabilities: **1)** generating proof steps justifying that the initial parity constraints are encoded in the input formula, **2)** tracking the subset of parity constraints S that give rise to each reason clause C, and **3)** generating proof steps to justify reason clause C based on those constraints.

The first and third capabilities are provided by tbuddy, while the second involves adding a tracking capability to CryptoMiniSat.

2.1 Trusted BDDs (TBDDs)

The tbuddy package supports *trusted* BDDs (TBDDs) as its core data structure. A trusted BDD represents a Boolean function that is guaranteed to yield 1 for any assignment that satisfies the input formula. Each trusted BDD $\dot{\mathbf{u}}$ is represented by 1) root node $\mathbf{u}$ in the BDD data structure, 2) extension variable u associated with BDD node $\mathbf{u}$, and 3) a series of proof steps leading to a step for the unit clause $[u]$, indicating that any assignment that satisfies the formula must also assign 1 to u. Performing a sequence of TBDD operations leading to the generation of trusted BDD $\dot{\mathbf{T}}_0$, where $\mathbf{T}_0$ is the terminal BDD node for constant 0, has the effect of generating a sequence of proof steps leading to the empty clause. Tbuddy supports several key operations on TBDDs [4]:

conjunction has argument TBDDs $\dot{\mathbf{u}}$ and $\dot{\mathbf{v}}$. It generates a TBDD $\dot{\mathbf{w}}$ representing the conjunction of the functions represented by BDDs $\mathbf{u}$ and $\mathbf{v}$.

upgrade has as arguments trusted BDD $\dot{\mathbf{u}}$ and regular BDD $\mathbf{v}$. It upgrades $\mathbf{v}$ to TBDD $\dot{\mathbf{v}}$ based on a proof that $u \rightarrow v$.

clause justification has as arguments trusted BDD $\dot{\mathbf{u}}$ and clause C. It generates one or more proof steps leading to a step consisting of clause C based on the implication $u \rightarrow C$.

2.2 Justifying the Initial Parity Constraints

CryptoMiniSat uses heuristic methods to detect parity constraints encoded in clausal form. When it detects an input parity constraint P_i over k variables, it has tbuddy generate a TBDD representation $\dot{\mathbf{u}}_i$ of P_i. To do so, tbuddy simply adds the 2^{k-1} clauses encoding P_i to the proof. Each of these should follow from some subset of the input clauses by reverse unit propagation (RUP), and therefore the proof checker can add the necessary hints to the proof. Based on these, tbuddy uses its conjunction operation to form a TBDD $\dot{\mathbf{v}}$ representing the conjunction of the added clauses. It also directly generates a BDD representation $\mathbf{u}_i$ of P_i, having the form shown in Figure 1, but with the specified set of variables and the specified phase. It then uses $\dot{\mathbf{v}}$ to upgrade $\mathbf{u}_i$ to TBDD $\dot{\mathbf{u}}_i$. In most cases, $\mathbf{u}$ and $\mathbf{v}$ are identical, and so the implication is a tautology.

2.3 Tracking Constraint Origins

The standard implementation of G-J elimination in CryptoMiniSat [23] need not track which of the original parity constraints contribute to a conflict or propagation. Instead, it can directly derive the reason clause from its *parity matrix*, the representation of parity constraints it maintains during Gauss-Jordan elimination. In order to add tracking, the modified code keeps a binary *shadow matrix* M, having dimension $n \times n$ for a parity matrix with n rows. At startup, M is initialized to the identity matrix. The shadow matrix, as its name implies, shadows the operations on the parity matrix. When two rows are swapped in the parity matrix, the shadow matrix also swaps rows, and when two rows are summed, the shadow matrix also sums the corresponding two rows. Hence, at all times, entry i, j of M is set to 1 under the condition that, in forming what is now constraint i, initial constraint j was added an odd number of times. (Observe that summing a constraint an even number of times effectively cancels it out.) With the shadow matrix, the program can determine the origin of a row of the parity matrix when it causes conflict or unit propagation. The shadow matrix can be stored and manipulated with the same bit-vector representation used for the parity matrix.

Example. Let us take as initial parity constraints $\{P_1 := x_1 \oplus x_2 = 1, P_2 := x_1 \oplus x_3 = 0, P_3 := x_1 \oplus x_2 \oplus x_3 = 1\}$. Then the initial matrix M_0 and corresponding shadow matrix S_0 is as follows. Notice that the last column of M_0 contains the phase values. Since M_0 has 3 rows, S_0 is the 3×3 identity matrix:

$$M_0 = \left[\begin{array}{ccc|c} x_1 & x_2 & x_3 & \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{array}\right] \qquad S_0 = \left[\begin{array}{ccc} P_1 & P_2 & P_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]$$

The G-J elimination algorithm now takes the 1st row of M_0, and sums it into the 2nd and 3rd rows of M_0 in order to eliminate the non-zero values in the first column of M_0 for all but the first row. Let's call the resulting matrices M_1 and S_1:

$$M_1 = \left[\begin{array}{ccc|c} x_1 & x_2 & x_3 & \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right] \qquad S_1 = \left[\begin{array}{ccc} P_1 & P_2 & P_3 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{array}\right]$$

Notice that S_1's 2nd row now reads 110, i.e. M_1's 1st row can be reconstructed via $P_1 \oplus P_2$. Similarly, S_1's 3rd row reads 101, i.e. M_1's 3rd row can be reconstructed via $P_1 \oplus P_3$. In this way, by mirroring row swap and row sum operations, S_n can always be used to read out which original parity constraints need to be summed obtain the corresponding constraint in M_n.

2.4 Justifying Reason Clauses

Each reason clause C must be justified based on a subset S of the initial parity constraints. This involves forming a TBDD representation $\dot{\mathbf{v}}$ of the sum constraint $\Sigma(S)$ and then using $\dot{\mathbf{v}}$ to justify C. Given TBDDs $\dot{\mathbf{u}}_a$ and $\dot{\mathbf{u}}_b$ representing parity constraints P_a and P_b, TBUDDY can generate a TBDD representation $\dot{\mathbf{u}}_c$ of parity constraint $P_c = P_a \oplus P_b$. It does so by the following steps. First, it conjuncts $\dot{\mathbf{u}}_a$ and $\dot{\mathbf{u}}_b$ to form TBDD $\dot{\mathbf{w}}$. Then, it generates the BDD representation $\mathbf{u}_c$ for P_c, having the same form as shown in Figure 1. Finally, it upgrades $\mathbf{u}_c$ to TBDD $\dot{\mathbf{u}}_c$ via a proof of the implication $w \rightarrow u_c$.

The BDD representation $\mathbf{w}$ of the conjunction of constraints P_a and P_b can be of size $\Theta(k_a \cdot k_b)$, where k_a and k_b are the number of nonzero coefficients in P_a and P_b, respectively. On the other hand, the BDD representation $\mathbf{u}_c$ of their sum will be of size $O(k_a + k_b)$. Since the sizes keep growing over multiple elimination steps, the resulting savings in BDD operations (and therefore proof steps), can be considerable.

The TBDD representation for the sum of constraints S is generated by performing pairwise sums. Since the BDD representation of a constraint with k variables has $2k - 1$ nodes, it is important to sum the constraints in a way that preserves sparseness (i.e., keeping the number of variables in the constraints low) among the intermediate results. TBUDDY does so with a greedy selection policy that at each step sums the pair of constraints P_a and P_b that minimizes the size of $\mathsf{Var}(P_a) \triangle \mathsf{Var}(P_b)$, with ties broken randomly. This can be done efficiently using a priority queue and incremental updating.

Finally, the clause justification operation adds the reason clause C to the proof based on TBDD $\dot{\mathbf{v}}$, representing the sum of the constraints. A single RUP step suffices for a reason clause arising from a parity constraint.

Once the TBDD representation of a sum of constraints has been used to justify a reason clause, it can be dereferenced, allowing the TBUDDY garbage collector to reclaim nodes and to delete some of the proof clauses. This dereferencing is done only when the row of the parity matrix is modified. As a result, a given row can be used to justify multiple reason clauses.

3 Experimental Results

We have performed experimental evaluations for several families of unsatisfiable formulas that are especially challenging for CDCL solvers. We compared different operating modes of CryptoMiniSat to TBSAT, a BDD-based solver built on top of TBUDDY. Tbsat has its own Gauss-Jordan elimination solver, and it can combine parity reasoning with *bucket elimination* [10, 15, 20], a systematic way to reduce a set of BDDs to one of the two possible terminal nodes via a sequence of conjunctions and quantifications [4, 7].

We found that the standard DRAT-TRIM proof checker performs poorly on proofs generated by TBUDDY, due to the large number of RAT lemmas used to define extension variables. Instead, we modified CryptoMiniSat to generate proofs in FRAT format [1], with over 50% of the proof clauses generated by CryptoMiniSat and 100% of those generated by TBUDDY having hints. The associated FRAT-RS verifier can make use of these hints, while also generating hints for the other clauses.

3.1 Urquhart Formulas

Urquhart's formulas [27] consist of sets of parity constraints based on undirected graphs in a manner similar to a construction of Tseitin [26]. There is a variable associated with each edge, and the formula contains an even or odd parity constraint for each node over its incident edges. Each edge occurs in two constraints, and so the formula will be unsatisfiable as long as the sum of the node phases is odd. We used a benchmark generator written by Li [18] that is parameterized by both the graph size m and a value p with $25 \leq p \leq 75$ indicating the percentage of nodes that are assigned odd parity.

As mentioned earlier, even a minimum instance of these formulas, having $m = 3$, is beyond the reach of today's CDCL solvers. Applying Gaussian elimination to these constraints, on the other hand, yields the infeasible constraint $0 = 1$, and so they are readily handled just using parity constraint reasoning. We have also shown that a BDD-based solver has polynomial scaling on these formulas when using bucket elimination [4, 7].

We generated Urquhart Formulas for hardness parameter m ranging from 3 to 15 and with phase percentage parameter p ranging from 26 to 70. Tests were performed on AMD Ryzen 9 5950X processors with a time limit of 3600 seconds per run. The solving and verification times for the two solvers are shown in Table 1. The rightmost column shows the time required by FRAT-RS to check the proofs generated by CryptoMiniSat plus TBUDDY. As can be seen, the Urquhart formulas pose no challenge for either solver, and the time to check the proofs are comparable to their generation times. Tbsat's performance advantage over CryptoMiniSat on these benchmarks can be attributed to its use of a sparse representation for parity constraints and that the problem can be solved purely by parity reasoning.

m	Tbsat lrat enabled Gauss-J	Tbsat lrat enabled Bucket-Elim	lrat-check of tbsat Gauss-J	lrat-check of tbsat Bucket-Elim	CMS Gauss-J with FRAT	frat-rs Verify CMS Proof
3	0.00	0.00	0.69	0.69	0.03	0.06
4	0.01	0.01	0.69	0.69	0.10	0.17
5	0.03	0.03	0.69	0.69	0.24	0.40
6	0.05	0.05	0.69	0.69	0.46	0.84
7	0.07	0.07	0.69	0.69	0.87	1.66
8	0.10	0.10	0.69	0.69	1.44	2.96
9	0.13	0.13	0.69	0.69	2.08	4.34
10	0.16	0.17	0.69	0.69	3.33	7.30
11	0.20	0.21	0.69	0.69	5.37	12.26
12	0.25	0.26	0.69	0.69	5.00	11.73
13	0.31	0.30	0.69	0.69	7.25	16.61
14	0.37	0.36	0.69	0.69	10.36	23.96
15	0.43	0.43	0.69	0.69	14.10	34.47

Table 1: Average times to solve and to verify Urquhart Formulas with Tbsat and CryptoMiniSat+tbuddy

3.2 Learning Parity with Noise

Learning Parity with Noise (LPN), also known as the minimal disagreement parity problem, is a well known hard problem for CDCL SAT solvers. Crawford contributed several instances of formulas for this problem as part of the original SATLIB benchmark suite [8]. We wrote a new generator for this problem,[3] since Crawford's generator is no longer available. From a satisfiability solving perspective, these formulas combine a set of parity constraint with a cardinality constraint. As a consequence, Gauss-Jordan elimination does not lead directly to an infeasible constraint, but it can serve a useful role in reasoning about the parity constraints, while the clausal reasoning component deals with the cardinality constraint.

The problem is parameterized by values n, m, and k. There are m parity constraints defined over a set of n *solution variables* $s_1, s_2, \ldots, s_n$, plus a set of m *corruption variables* $r_1, r_2, \ldots, r_m$. Each parity constraint P_i depends on some randomly chosen subset of the solution variables, plus corruption variable r_i. An instance is generated so that for a *target solution*, consisting of a randomly generated bit sequence $a_1, a_2, \ldots, a_n$, a subset of $n-k$ of the constraints will be satisfied when the solution variables are set to the target solution and with their associated corruption variables r_i set to 0, while k of them require $r_i = 1$ for the target solution to satisfy the constraint. That is, k of the equations have been "corrupted" by flipping their phases. An at-most-k constraint is placed on the corruption variables, and thus the target solution will satisfy the formula, as can other solutions, as long as at most k constraints are corrupted. The standard LPN formulas are satisfiable, but by imposing the condition that at most $k-1$ of the corruption bits can be set to 1, the formula will generally (but not necessarily) become unsatisfiable.

For each value of n from 20 to 35, we generated 20 instances of LPN formulas, setting $m = 2n$. Each constraint in each formula was corrupted at random with probability 0.125, causing the number of corrupted constraints k in each of the formulas to vary, but having

[3] Available at https://github.com/rebryant/mdp-benchmark.

n	UNSAT	CryptoMiniSat without G-J		CryptoMiniSat with G-J	CryptoMiniSat with G-J+tbuddy		frat-rs avg (s)
		T.O.	PAR-2	no verification	T.O.	PAR-2	
20	15	0	4.59	0.03	0	1.00	0.94
21	10	0	4.29	0.03	0	1.06	1.64
22	13	0	20.67	0.05	0	2.41	3.98
23	14	0	31.96	0.05	0	2.79	2.55
24	16	0	113.09	0.05	0	3.40	4.51
25	11	0	257.20	0.14	0	9.45	20.51
26	14	0	908.71	0.25	0	17.70	28.63
27	13	5	2401.46	0.31	0	22.91	42.79
28	8	5	2184.10	0.73	0	38.39	42.95
29	8	10	3800.17	1.27	0	70.64	159.45
30	14	12	4696.38	1.70	0	101.98	289.13
31	15	13	5120.66	3.07	0	194.07	726.44
32	12	15	5519.15	3.09	0	216.26	553.69
33	10	16	6167.31	6.71	0	364.75	1996.44
34	12	18	6631.76	19.99	5	2169.01	1391.08
35	9	18	6493.04	14.79	5	2144.69	1922.08

Table 2: Applying different configurations of CryptoMiniSat to LPN formulas. There were 20 instances for each value of n. The verification times are shown only for UNSAT instances. Columns labeled "T.O." indicate the number of cases that exceeded the 3600-second time limit.

average value $n/4$. These choices match the parameters suggested by Crawford. The formulas were then passed to both CryptoMiniSat and tbsat to be solved, and the resulting proofs were respectively passed to lrat-check and frat-rs to verify.

For CryptoMiniSat, we ran these on Intel 2xE5-2690v3 CPUs with 24GB RAM allocated for each run, a time limit of 3600 seconds. For tbsat, we ran these on a Ryzen 9 5950x with 4GB allocated for each run, a time limit of 3600s, and a proof size limit of 2^{30} clauses. The proof size limit was set based on our experience with the lrat-check checker. Allowing larger proofs would only generate unwieldy files and create proofs that are too large to be checked.

CryptoMiniSat and frat-rs performance Table 2 shows the performance of several different ways to apply CryptoMiniSat to these formulas, with the associated PAR-2 scores[4]. As is shown, the number of unsatisfiable formulas for each value of n ranges from 8 (40%) to 16 (80%). Without Gauss-Jordan elimination, CryptoMiniSat starts hitting the time limit with $n = 27$ and does so for 90% of the formulas once n reaches 34. As a result, the PAR-2 times trend toward the limiting value of 7200. On the other hand, using Gauss-Jordan elimination and without concern for proof generation, all of the benchmarks could be solved in under 20 seconds.

The measurements for CryptoMiniSat in combination with tbuddy were performed by running the solver with proof generation enabled, regardless of whether or not the formula was satisfiable. The combination enables proof generation for much larger problems than is possible with pure CDCL. The solver exceeded the time limit only for $n \geq 34$. Compared to having no proof generation, it exacts a significant performance penalty. The increased time

[4]PAR-2 scores are used in the SAT competitions to measure performance. Each benchmark contributes a score that is the number of seconds used to solve it, or in case of a timeout or memory out, twice the timeout in seconds. The average score for all benchmarks is then calculated, giving PAR-2.

N	UNSAT	TBSAT G-J		LRAT-CHECK
		P.O.	PAR-2	avg (s)
20	15	0	0.18	0.38
21	10	0	0.55	0.22
22	13	0	0.37	0.14
23	14	0	0.43	0.11
24	16	0	0.36	0.13
25	11	0	4.49	0.17
26	14	0	1.54	0.21
27	13	0	16.19	0.25
28	8	0	42.51	0.25
29	8	0	69.72	0.44
30	14	2	756.12	0.60
31	15	1	387.51	0.69
32	12	3	1104.10	0.64
33	10	2	792.41	1.15
34	12	4	1489.05	3.24
35	9	4	1565.19	2.32

Table 3: Applying TBSAT in Gauss-Jordan mode to LPN formulas. There were 20 instances for each value of n. The verification times are shown only for UNSAT instances. The column labeled "P.O." indicates the number of instances for which the generated proof exceeded 2^{30} clauses.

could be partly due to the loose integration of our implementation. TBUDDY must start fresh in summing a subset of the parity constraints every time a new parity matrix row is used to generate a reason clause, even if many elements of the subset have been summed previously. A tighter integration, however, would require major rewriting of the already-complex parity reasoning code.

The final column shows the average times for running FRAT-RS for the cases where the formula is unsatisfiable and CRYPTOMINISAT is able to complete its execution. Overall, the performance is acceptable, but FRAT-RS can take around 3× more time to check a proof than was required to generate it. In these proofs, TBUDDY provides hints for each of its proof steps, and CRYPTOMINISAT does so for over 50% of them, and so it seems like it should be possible to improve the checker performance.

Tbsat and lrat-check performance Table 3 shows the performance of TBSAT with Gauss-Jordan elimination and LRAT proof generation applied to the LPN formulas. As before, the number of unsatisfiable formulas for each value of n ranges from 8 (40%) to 16 (80%). A separate program detects parity constraints in the CNF representation and generates a *schedule* for TBSAT identifying each constraint and the set of clauses that give rise to it [5]. TBSAT lacks any variable ordering heuristics, and so the user must supply a variable ordering to achieve good performance on these formulas. Our ordering for these formulas is as follows: the variables encoding the at-most-one constraints, the corruption variables, the auxilliary variables for the parity constraints, and finally the solution variables.

Overall, TBSAT ran significantly faster than CRYPTOMINISAT on these formulas, even accounting for the performance difference of the execution platforms. On the other hand, it was less stable. Even for formulas as small as $n = 30$, it could encounter a single conjunction operation that would cause so many BDD operations that the proof generator exceeded the limit of 2^{30} clauses. By contrast, CryptoMiniSat performance was very stable until it started exceeding the time limit once n reached 34. The generated proofs, however, could reach 100 GB for both solvers. Checking the LRAT proofs generated by TBSAT ran much faster than did the checking of the FRAT proofs generated by CryptoMiniSat. All of the problems that TBSAT could solve without overflowing the proof clause counter ran in under 1200s. Indeed, the PAR-2 scores would improve if the timeout limit were reduced.

4 Conclusion

CDCL solvers have a major weakness in their handling of parity constraints. Augmenting CDCL with Gauss-Jordan elimination is well established, but this capability has so far lacked the ability to generate proofs of unsatisfiability. Our work shows that standard clausal proof frameworks suffice, with BDDs providing a bridge between parity reasoning and proof generation. We hope that others will explore the CDCL(T) framework, while maintaining the important capability of generating unsatisfiability proofs.

References

[1] S. Baek, M. Carneiro, and M. J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I*, volume 12651 of *LNCS*, pages 59–75, 2021.

[2] L. A. Barnett and A. Biere. Non-clausal redundancy properties. In *Conference on Automated Deduction (CADE)*, volume 12699 of *LNAI*, pages 252–272, 2021.

[3] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[4] R. E. Bryant. TBUDDY:, a proof-generating BDD package. In *Formal Methods in Computer-Aided Design*, 2022.

[5] R. E. Bryant, A. Biere, and M. J. H. Heule. Clausal proofs for pseudo-Boolean reasoning. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, 2022.

[6] R. E. Bryant and M. J. H. Heule. Generating extended resolution proofs with a BDD-based SAT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I*, volume 12651 of *LNCS*, pages 76–93, 2021.

[7] R. E. Bryant and M. J. H. Heule. Generating extended resolution proofs with a BDD-based SAT solver. *CoRR*, abs/2105.00885, 2021.

[8] J. M. Crawford, M. J. Kearns, and R. E. Schapire. The minimal disagreement parity problem as a hard satisfiability problem. Mirrored at https://www.cs.cornell.edu/selman/docs/crawford-parity.pdf, 1994.

[9] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340, 2008.

[10] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.

[11] S. Gocht, C. McCreesh, and J. Nordström. VeriPB: The easy way to make your combinatorial search algorithm trustworthy. In *From Constraint Programming to Trustworthy AI*, 2020.

[12] S. Gocht and J. Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *AAAI Conference on Artificial Intelligence*, pages 3768–3777, 2021.

[13] C.-S. Han and J.-H. R. Jiang. When Boolean satisfiability meets Gaussian elimination in a simplex way. In *Computer-Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 410–426, 2012.

[14] M. J. H. Heule, W. A. Hunt, M. Kaufmann, and N. D. Wetzler. Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving*, volume 10499 of *LNCS*, pages 269–284, 2017.

[15] T. Jussila, C. Sinz, and A. Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *LNCS*, pages 54–60, 2006.

[16] O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999.

[17] T. Laitinen, T. Junttila, and I. Niemelä. Extending clause learning SAT solvers with complete parity reasoning. In *International Conference on Tools with Artificial Intelligence*, pages 65–72. IEEE, 2012.

[18] C.-M. Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003.

[19] J. Lind-Nielsen. *BuDDy: a Binary Decision Diagram Package*. Department of Information Technology, Technical University of Denmark, 1996.

[20] G. Pan and M. Y. Vardi. Search vs. symbolic techniques in satisfiability solving. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3542 of *LNCS*, pages 235–250, 2005.

[21] T. Philipp and A. Rebola-Pardo. DRAT proofs for XOR reasoning. In *Logics in Artificial Intelligence*, volume 10021 of *LNAI*, pages 415–429, 2016.

[22] C. Sinz and A. Biere. Extended resolution proofs for conjoining BDDs. In *Computer Science Symposium in Russia (CSR)*, volume 3967 of *LNCS*, pages 600–611, 2006.

[23] M. Soos, S. Gocht, and K. S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Computer Aided Verification (CAV)*, volume 12224 of *LNCS*, pages 463–484, 2020.

[24] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *LNCS*, pages 244–257, 2009.

[25] Y. K. Tan, M. J. H. Heule, and M. O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II*, volume 12652 of *LNCS*, pages 223–241, 2021.

[26] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1983.

[27] A. Urquhart. Hard examples for resolution. *J.ACM*, 34(1):209–219, 1987.

[28] N. D. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *LNCS*, pages 422–429, 2014.