Constructive Separations from Gate Elimination

Marco Carmosino

MIT-IBM Watson AI Lab, Cambridge, MA, USA

Ngu Dang

Boston University, Boston, MA, USA

Tim Jackman

Boston University, Boston, MA, USA

Abstract

Gate elimination is the primary technique for proving explicit lower bounds against general Boolean circuits, including Li and Yang's state-of-the-art $3.1n - o(n)$ bound for affine dispersers (STOC 2022). Every circuit lower bound is implicitly existential: every circuit that is too small to compute f must err on some input. This raises a natural question: are these lower bounds *constructive*? That is, can we efficiently produce such errors? Chen, Jin, Santhanam, and Williams showed that constructivity plays a central role in many longstanding open problems in complexity theory, and explicitly raised the question of which circuit lower bound techniques can be made constructive (FOCS 2021).

We show that a variety of gate elimination arguments yield refuters – efficient algorithms that, when given a circuit that is too small to compute a function f, produce an input on which the circuit errs. Our results range from elementary lower bounds for XOR and the multiplexer to more sophisticated arguments for affine dispersers. Underlying these results is a shift in perspective: gate elimination arguments *are* algorithms. Each step either simplifies the circuit or reveals a violation of some structural or functional property, from which, with a little additional work, explicit counterexamples can be extracted.

We further strengthen the XOR result to handle circuits that *match* the lower bound: given any DeMorgan circuit of size $3(n-1)$ that fails to compute XOR_n, we can efficiently produce a counterexample. While refuters follow from the gate elimination arguments themselves, this refinement requires a complete characterization of the set of optimal circuits computing XOR – a requirement rarely met by other explicit functions.

2012 ACM Subject Classification Theory of computation → Circuit complexity

Keywords and phrases Circuit Complexity, Constructivity

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material

© Marco Carmosino, Ngu Dang, and Tim Jackman;
licensed under Creative Commons License CC-BY 4.0
42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

One of the most important open problems in complexity theory is to characterize the functions computed by polynomial-size Boolean circuits. By a straightforward counting argument, most functions require huge circuits of size roughly $2^n/n^2$. But, after decades of research, it remains open to identify an *explicit* Boolean function (in P or even NP) that requires even *super-linear* circuit size.

The state of the art is a $3.1n - o(n)$ lower bound for *affine dispersers* (which can be computed in P), proved by Li and Yang in 2021 [12]. They used *gate elimination,* the best known technique for proving explicit and unconditional lower bounds against general Boolean circuits. Proving a lower bound for f via gate elimination works by setting the inputs of a circuit $\mathcal{C}$ to carefully-chosen values and simplifying C until we either (1) derive a contradiction to the assertion "$\mathcal{C}$ computes f" or (2) shrink a *complexity measure* of $\mathcal{C}$.

Gate elimination has been used to push the research frontier in circuit lower bounds since 1974 [17]. Yet we remain unable to identify an explicit function in NP that requires circuits of size $10n$. Indeed, a certain formalization of gate elimination cannot prove lower bounds better than $5n$ [7]. However, the barrier does not cover every argument by gate elimination, including those that depend on the optimal structure of circuits computing f or rely on special properties of the hard function f to successful carry out induction.

Gate elimination arguments that evade this barrier have been used to obtain breakthrough circuit lower bounds [2, 6], establish ETH-hardness of the partial Minimum Circuit Size Problem ($\mathsf{MCSP}^\star$) [9], design algorithms for the XOR-Simple Extension[1] Problem [3], and characterize the set of optimal circuits for a variety of key Boolean functions, including addition [16]. What can be said about this family of arguments, if anything? Towards a better understanding of gate elimination in general, this work shows that selected lower bounds for XOR, the multiplexer, and affine dispersers can be made *constructive.*

The statement "f does not have size $s(n)$ circuits" is implicitly existential; every circuit that fails to compute f must err on at least one input x. Thus, every circuit lower bound for a specific function f induces a total search problem: given as input a circuit $\mathcal{C}$ that is too small (and therefore *must* fail to compute f) print a bitstring on which $\mathcal{C}$ and f disagree. Constructive lower bounds play a surprisingly central role in complexity theory [4].

On the one hand, extremely difficult but widely-conjectured separations like $\mathsf{P} \neq \mathsf{NP}$ are *automatically* constructive if they hold at all. On the other hand, if certain *weak* and *simple* complexity lower bounds like "Palindromes require super-linear time to decide on one-tape Turing machines" can be made even P^{NP}-constructive, then $\mathsf{P} \neq \mathsf{NP}$! Paraphrasing Chen, Jin, Santhanm and Williams: the intuition that "easy to prove" lower bounds can be made constructive is "wildly inaccurate" [4]. Their work studied complexity separations for uniform classes of algorithms, and concluded by suggesting that "it would be interesting to examine which proof methods for circuit lower bounds can be made constructive".

Because gate elimination is the best tool we have to prove lower bounds for *unrestricted* Boolean circuits, it is a natural starting point. Furthermore, arguments via gate elimination seem difficult to categorize as "easy" or "hard". Globally, state-of-the-art lower bounds use deeply nested case analyses and carefully tailored complexity measures. Locally, each individual case of these arguments employs totally elementary manipulation of easily-identified subcircuits. We do not resolve the "proof complexity" of gate elimination here, but are able to show that selected arguments *hide efficient circuit-analysis algorithms.*

[1] The f-Simple Extension Problem was an important part in ETH-hardness of $\mathsf{MCSP}^\star$.

78 ## 1.1 Our Results & Contributions

79 We consider circuit size over two different bases. The *DeMorgan* basis contains binary AND
80 and OR gates and NOT gates. The $\mathbb{B}_2$ basis contains every binary Boolean function. The
81 *size* of a circuit C is the number of binary gates. Recall that the *multiplexer* (MUX_n, also
82 known as the storage access function) is a Boolean function on $n + 2^n$ bits, where the first n
83 *address bits* determine which of the 2^n *data bits* to output (Definition 10). Towards extracting
84 a constructive separation, in Section 3, we give an elementary proof of

85 ▶ **Lemma 1.** *Let $N = 2^n$. MUX_n requires DeMorgan-size at least $2N + \log N - 2$ to compute.*

86 Lemma 1 is not tight; the best *upper bound* for MUX_n is a DeMorgan circuit of size
87 $2N + O(\sqrt{N})$ [11]. Even so, it demonstrates constructivity in gate elimination. We recall
88 the definition of a constructive separation for non-uniform algorithms.

89 ▶ **Definition 2** (Refuters against circuits [4])**.** *Let $\mathsf{f} : \{0,1\}^\star \to \{0,1\}$ be a Boolean function,*
90 *and let Γ denote a standard complexity class. A* Γ-refuter *for f against size $s(n)$ circuits*
91 *is a Γ-algorithm R that, given input 1^n and any circuit $\mathcal{C}$ of size strictly less than $s(n)$,*
92 *prints a string $x \in \{0,1\}^n$ such that $R(1^n, \mathcal{C}) \neq \mathsf{f}(x)$ for almost every n. We say there is a*
93 Γ-constructive separation *of f from size $s(n)$ if there is a Γ-refuter for f against $s(n)$-size*
94 *circuits. When $\Gamma = \mathsf{P}$, we drop the prefix and just say* refuter *and* constructive separation*.*

95 Inspecting the proof of Lemma 1, in Section 4.1, we obtain

96 ▶ **Theorem 3.** *There is a constructive separation of MUX from DeMorgan-size $2N + \log N - 2$.*

97 The most advanced circuit lower bounds proved via gate elimination are for *affine*
98 *dispersers:* Boolean functions that are non-constant on every d-dimensional affine subspace of
99 $\mathbb{F}_2^n$. So we capture their constructivity by introducing the following class of search problems.

100 ▶ **Definition 4.** *Let $d : \mathbb{N} \to \mathbb{N}$ be an arbitrary function. An* affine refuter *for dimension*
101 *d against size $s(n)$ circuits is a P-algorithm that, given input 1^n and any circuit C of size*
102 *strictly less than $s(n)$, prints the description of an affine subspace of dimension $d(n)$ on*
103 *which C is constant.*

104 Inspecting the elementary proof of $3n - o(n)$ lower bounds for affine dispersers [5], in
105 Section 4.2, we obtain

106 ▶ **Theorem 5.** *There is an affine refuter for dimension d against $\mathbb{B}_2$-size $3n - 4d(n)$.*

107 ▶ **Corollary 6.** *Fix any explicit affine disperser $\mathsf{f} \in \mathsf{P}$ of dimension $o(n)$. There is a*
108 *P^{NP}-constructive separation of f from circuits of size $3n - o(n)$.*

109 Denote by XOR_n the sum modulo 2 of n bits. There is a generalization of Schnorr's lower
110 bound that totally characterizes the set of optimal DeMorgan-circuits computing XOR_n [3].
111 In Section A.1, we first inspect Schnorr's classic argument to obtain

112 ▶ **Theorem 7.** *There is a constructive separation of XOR from DeMorgan-size $3(n - 1)$.*

113 But an argument that characterizes the set of optimal circuits should contain more
114 powerful circuit analysis algorithms. Indeed, we refine the refuter for XOR to solve the
115 problem of efficiently *checking* correctness of claimed optimal circuits for XOR. In particular,
116 given a circuit $\mathcal{C}$ of size exactly $3(n - 1)$ that purports to compute XOR_n but *does not*, can
117 we efficiently produce bitstrings x such that $\mathcal{C}(x) \neq \mathsf{XOR}_n(x)$? Note that this is no longer a
118 total search problem. Even so, the task is feasible. In Section A.3, we obtain

119 ▶ **Theorem 8.** *Let $\mathcal{C}$ be any DeMorgan circuit on n inputs with exactly $3(n-1)$ binary gates*
120 *but does not compute XOR_n. There is an efficient and deterministic algorithm that, given*
121 *input $\mathcal{C}$, prints an n-bit string x such that $\mathcal{C}(x) \neq \mathsf{XOR}_n(x)$.*

1.2 Related Work

Feasible Mathematics and Bounded Arithmetic. One way to obtain refuters is to prove complexity separations in *bounded arithmetics:* weak fragments of Peano Arithmetic that enjoy tight connections to efficient algorithms. In particular, these theories have *efficient witnessing* where proofs of existential statement can be automatically converted into efficient algorithms that print the objects asserted to exist. Some of the lower bounds for *restricted* circuit classes, like AC^0, have been formalized in bounded arithmetic and so have efficient refuters [14]. To the best of our knowledge, no circuit lower bound via gate elimination has been formalized in bounded arithmetic.

Grosser and Carmosino showed that many of the dramatic implications that come from making simple and weak lower bounds constructive in the *algorithmic* sense still hold (in slightly weaker forms) when we consider proofs in *weak fragments of Peano Arithmetic* closely associated with computational complexity classes, like Cook's PV (which is connected to deterministic polynomial time) [8].

Lower Bounds for MUX. An asymptotically tight circuit lower bound for MUX over the DeMorgan basis is stated by [13], but we cannot locate a published proof. Tight lower bounds are required to characterize the set of optimal MUX circuits and derive an efficient checker.

Checking. Our checker for XOR is like a white-box and deterministic version of a *program checker* introduced by Blum and Kannan in the 90s [1]. In particular, a program checker is a *probabilistic* algorithm that— given a *deterministic* program P, an input instance I— certifies with high probability whether the output produced by $P(I)$ is correct or instead declares the program P to be buggy. A central conceptual contribution of this work is the insight that the structural properties of the underlying problem can be leveraged to make checking efficient: by querying the program on carefully chosen related instances, one can test consistency conditions that any correct computation must satisfy.

Blum and Kannan provided concrete checkers for some explicit polynomial time computable problems such as sorting, computing the matrix rank, the greatest common divisor, and for several group-theoretic problems. Our results extends this line of research to the Boolean circuits' regime by showing how one can utilize the known characterization of optimal XOR-circuits to efficiently and *deterministically* check whether a given circuit $\mathcal{C}$ (of size $3n-3$) correctly computes XOR_n.

1.3 Proof Techniques

1.3.1 Elementary MUX Lower Bound

Paul's classical argument for a $2(N-1)$ $\mathbb{B}_2$ lower bound for MUX_n only restricts individual *data bits* [15]. As a result, the restricted circuit must still multiplex the remaining data bits. To ensure that at least two gates are always eliminated, Paul's substitutes *functions* of variables, rather than constants, in order to fix $\oplus$-type gates. As these substitutions may make certain address bits degenerate in the restricted function, the proof cannot manipulate the address bits to eliminate more gates. In Section 3 however, we show that the argument becomes significantly simpler in the DeMorgan basis. In $\mathbb{D}$, constant substitutions suffice to eliminate two gates per data bit. After substituting all of the data bits which agree on a certain address bit a_i, the underlying function is guaranteed to still depend on a_i. We can therefore substitute a_i to eliminate even more gates. This accounts for an extra $\log N$ factor in the $2N + \log N - 2$ lower bound established in Lemma 1.

166 ### 1.3.2 Efficient Refuters

167 Gate elimination often involves arguments that enough specific simplifications can be applied
168 after certain substitutions. If not, the circuit must violate a structural or functional property
169 of its underlying function. After simplification, the circuit computes a smaller instance of the
170 function (or a smaller instance in the function class) being considered and the lower bound
171 follows inductively.
172 In Section 4, view proofs via gate elimination as *structured algorithms*. We run the
173 arguments on circuits which *are* to small to compute the function by interpreting the case
174 analysis as a branching algorithm. As the circuits do not meet the lower bound, this
175 process *must* either fail at some step or efficiently reduce the circuit to constant size. If
176 the algorithm fails, then we can extract a counterexample from the violated structural or
177 functional property. If the algorithm reaches a constant-size restriction, the refuter can brute
178 force over all remaining inputs and find one on which it errs. The algorithms for MUX and
179 XOR in Sections 4.1 and Appendix A.1 respectively follow this structure exactly and are
180 able to extract counterexamples efficiently. The affine refuter of Section 4.2 also mirrors this
181 structure and produce affine subspaces on which the circuits are constant.

182 ### 1.3.3 Efficient Optimal XOR-circuits Checker

183 While gate elimination arguments naturally extend to refuters they do not as readily produce
184 checkers – algorithms which find errors for incorrect circuits whose size *meets* the lower
185 bound. Generally, the issue with naively running a refuter as checker is that a "successful"
186 substitution by the checker (i.e., one that succeeds at reducing the complexity of the circuit by
187 the minimal amount) may produce a circuit which *does* correctly compute the corresponding
188 restricted function. The original circuit may have only erred on inputs where the variable
189 was set according to a different value and therefore a checker will then never be able to find
190 an error.
191 In Section A, we show that it is sometimes possible to circumvent this by extending a
192 refuter for XOR into a checker. To do so, we efficiently *detect* when the reduced circuit is
193 correct, so that the algorithm can instead make the opposite constant substitution. We
194 leverage the characterization of optimal DeMorgan XOR circuits as binary trees of XOR_2
195 and $\neg\mathsf{XOR}_2$ widgets from [3]. As the converse of this theorem also holds (i.e., binary trees of
196 XOR_2 and $\neg\mathsf{XOR}_2$ widgets compute XOR (or ¬XOR)), the checker (Algorithm 4) can detect
197 whether the simplified circuit computes XOR in polynomial time and backtrack one step if
198 necessary. Afterwords, all that is required is to argue that the opposite substitution also
199 eliminates enough gates or otherwise violates some property of XOR.

200 ## 1.4 Future Directions

201 These constructivity results for XOR, MUX, and an *elementary* argument about affine
202 dispersers immediately raise further questions: what about state-of-the-art circuit lower
203 bounds? Do the more sophisticated complexity measures impede refutation by efficient
204 algorithms? Are there dramatic consequences of making those arguments constructive? We
205 also ask whether our refuter for MUX and the affine refuter can be extended to a checker
206 and improved to a constructive separation respectively. Are there any consequences if there
207 constructive separations of affine functions f of dimension d from size $3n - 4d(n)$?
208 Finally, it would be interesting to determine if gate elimination can be formalized in
209 bounded arithmetic. This would establish that not only is the refuter efficient, but also that
210 the proof of correctness for the refuter is feasible. We conjecture that all the arguments from

this paper can be formalized in PV, which seems to contain enough linear algebra to argue about affine dispersers.

2 Preliminaries

2.1 General Notation

We write $[n]$ to represent the set $\{1, 2, \ldots, n\}$. We write addition modulo 2 over the field $\mathbb{F}_2 = \{0, 1\}$ explicitly using $\oplus$. Vectors are written with an arrow e.g., $\vec{v} \in \mathbb{F}_2^n$, and addition between vectors $\vec{v} + \vec{u}$ denotes component-wise addition modulo 2. We write $\vec{e_i}$ for the ith standard basis vector, i.e, the vector whose i-th coordinate is 1 and whose other coordinates are 0. The all zero vector is $\vec{0}$, the $n \times n$ identity matrix is $\mathbf{I}_n$, and the $n \times n$ all-zeroes matrix is $\mathbf{0}_n$. For a set $I \subseteq [n]$, we write $\mathbb{1}_I \in \mathbb{F}_2^{1\times n}$ for the row indicator vector of I, i.e., the row vector whose i-th coordinate is 1 if and only if $i \in I$.

2.2 Boolean Circuits

We assume familiarity with the basics of Boolean circuit complexity; see [10] for a standard reference. We consider unrestricted circuits over the *DeMorgan* basis, $\mathbb{D} = \{\wedge, \vee, \neg\}$, and the $\mathbb{B}_2$ basis consisting of all Boolean functions of fan-in at most 2. As in [5], we divide the 16 binary Boolean functions $\mathsf{f}(x, y)$ of $\mathbb{B}_2$ into three categories: six degenerate functions (i.e., $0, 1, x, \neg x,\ y,\ \neg y$), eight $\wedge$-type (i.e., $x \wedge y, \neg x \wedge y, \ldots, \neg(\neg x \wedge \neg y)$), and two $\oplus$-type (i.e., $x \oplus y$ and $\neg(x \oplus y)$). Observe that an optimal circuit computing a function over $\mathbb{B}_2$ never contains degenerate gates unless they're computing the degenerate functions themselves.

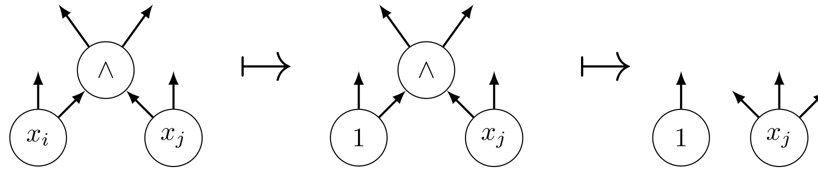
The size of a circuit $\mathcal{C}$, denoted $\sigma(\mathcal{C})$, is the number of binary gates in $\mathcal{C}$. We denote by $\eta(\mathcal{C})$ the number of input variables in $\mathcal{C}$ whose fanout is at least one. The circuit complexity of a function $\mathsf{f} : \mathbb{F}_2^n \to \mathbb{F}_2$ over a basis $\mathbb{B}$, denoted $\mathsf{CC}_{\mathbb{B}}$, is the minimum size of any $\mathbb{B}$-circuit computing f. When working with circuits over the DeMorgan basis, we will often write $(\neg)\alpha$ to refer to either a binary gate α or a $\neg$ gate reading α.

2.3 Gate Elimination

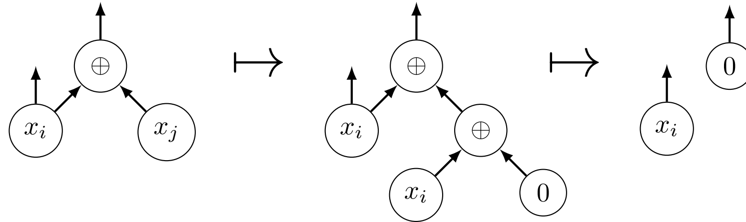
Lower bounds for general Boolean circuits use the *gate elimination* method: first, fix some inputs to some value, then repeatedly simplify the resulting circuit. Simple proofs via gate elimination substitute constants and then simplify according to basic Boolean identities. Each simplification either replaces a gate by a constant or merges it into one of its inputs. We categorize these identities into the following three types:

- *Fixing* rules: directly turn a gate into the constant 0 or 1 (e.g. $0 \wedge \gamma \to 0$).
- *Passing* rules: replace a gate by one of its inputs (e.g. $1 \wedge \gamma \to \gamma$), letting that input inherit the outgoing wires of the gate.
- *Trivial simplifications*: eliminate a literal and its negation (e.g. $\gamma \wedge \neg\gamma \to 0$) or remove duplicate inputs (e.g. $\gamma \wedge \gamma \to \gamma$) and double negations.

We apply these simplifications repeatedly until we can no longer. We call such a circuit, in which no further identities can be applied, *normalized* or *in normal form*. Note that when working in $\mathbb{B}_2$, no single constant substitution can fix an $\oplus$-type gate. Hence, more advanced gate elimination arguments substitute functions. Examples of these substitutions are given in Figure 1. While simplifying, gate elimination arguments track certain complexity measures; basic arguments simply track circuit size, thereby concluding that the original circuit had at



(a) A constant substitution and accompanying passing rule



(b) A linear substitution and accompanying fixing rule

Figure 1 Two examples of simplifications in $\mathbb{D}$ and $\mathbb{B}_2$. In (a), x_i is substituted for a constant which simplifies $\wedge$, passing its out wires to x_j. In (b), x_j is substituted with the linear equation $x_j \leftarrow x_i \oplus 0$. The top $\oplus$ gate therefore computes $x_i \oplus (x_i \oplus 0) = 0$ and thus becomes fixed.

252 least as many gates as were removed. More advanced gate elimination arguments, like [5],
253 track additional quantities, such as $\eta(\mathcal{C})$.

254 2.4 Specific Functions and Function Classes

255 We consider the following functions and function class.

256 ▶ **Definition 9** (XOR). *For $\vec{x} \in F_2^n$, we define the* XOR *function on n bits (*XOR_n*) as*
257 $\mathsf{XOR}_n(\vec{x}) = \bigoplus_{i \in [n]} x_i$*. For a set $I \subseteq [n]$, we define* $\mathsf{XOR}_I(\vec{x}) = \bigoplus_{i \in I} x_i$*.*

258 We use $(\neg)$XOR to denote the XOR function itself or its negation $\neg$XOR.

259 ▶ **Definition 10** (MUX). *We define the multiplexer* $\mathsf{MUX}_n : \mathbb{F}_2^n \times \mathbb{F}_2^{2^n} \to \mathbb{F}_2$ *as follows. For*
260 *all $(\vec{a}, \vec{x})$ where $\vec{a} \in \mathbb{F}_2^n, \vec{x} \in \mathbb{F}_2^{2^n}$, we have* $\mathsf{MUX}_n(\vec{a}, \vec{x}) = \vec{x}_{(\vec{a})}$*, where $\vec{x}_{(\vec{a})}$ where $(\vec{a})$ is $\vec{a}$*
261 *interpreted as a base-2 number plus 1, i.e., the $((\vec{a}) + 1)$-th bit of $\vec{x}$.*

262 ▶ **Definition 11** (Affine Subspace and Dispersers). *A set $S \subseteq \mathbb{F}_2^n$ is an* affine subspace of
263 dimension *d if there exists a full column rank matrix $A \in \mathbb{F}_2^{n \times d}$ and vector $\vec{a} \in \mathbb{F}_2^d$ such that*
264 *$S = \{A\vec{x} + \vec{a} \mid \vec{x} \in \mathbb{F}_2^n\}$. A function $\mathsf{f} : \mathbb{F}_2^n \to \mathbb{F}_2$ is an* affine disperser of dimension *d if f is*
265 *not constant on any affine subspace of dimension at least d.*

266 3 An Elementary Lower Bound for MUX

267 Paul's $2N - 2$ lower bound for MUX_n in the $\mathbb{B}_2$ basis [15] propagates to the DeMorgan basis.
268 However, in the DeMorgan basis, the argument becomes simpler (and tighter) as constant
269 substitutions suffice to restrict data bits and therefore the argument can take advantage of
270 address bits. Substituting for an address bit makes half of the data bits irrelevant, allowing
271 us to substitute irrelevant data bits to eliminate as many gates as possible.
272 To streamline the proof of correctness for our refuter (Theorem 13), we present the proof
273 of Theorem 12 in a labeled, case-based format. The structure of the proof aligns closely with
274 the structure of the refuter; labeling allows us to reference corresponding steps of the proof

when analyzing the refuter. Cases will correspond to branches of our program. When Cases are proofs by contradiction, we label their conclusion as Assertions.

▶ **Theorem 12.** *In the DeMorgan basis,* $\mathsf{CC}(\mathsf{MUX}_n) \geq 2N + \log N - 2$ *where* $N = 2^n$*.*

Proof. We will show that $\mathsf{CC}_{\mathbb{D}}(\mathsf{MUX}_n) \geq 2 \cdot 2^n + n - 2$. We exploit the following observation about MUX_n: restricting a_1 removes dependence on the half of the data bits. Hence, we can restrict those degenerate data bits to remove as many possible gates as possible. Afterwards, by downward self-reducibility, the circuit is a multiplexer on the remaining data bits and we can apply induction.

Base Case. Observe that MUX_1 requires at least $3 = 2 \cdot 2^1 + 1 - 2$ costly gates. MUX_1 depends on all of its inputs: a_1, x_1, x_2 and thus there are at least two binary gates which read them. Furthermore, none of these inputs can feed the output as otherwise, substituting them to fix the output makes the circuit constant, but $\mathsf{MUX}_1\restriction_{v \leftarrow b}$ is not constant for any variable v and constant b. Therefore there must be at least one more gate which reads the gates that read a_1, x_1, x_2.

Inductive Case. Let $\mathcal{C}$ be a circuit computing MUX_n for $n > 1$. Let $D = \{x_i \mid i > 2^{n-1}\}$, i.e., the data inputs whose first address bit is 1. We will find a substitution for each x_i in D (i.e., exactly half of the data inputs of $\mathcal{C}$) so that two binary gates are eliminated.

Loop over D. Fix $x_i \in D$. We will substitute for x_i to eliminate at least 2 binary gates.

- **Case 1: Data degeneracy.** Suppose $\mathcal{C}$ does not read x_i. Then, $\mathcal{C}$ does not depend on x_i while MUX_n does. Hence, $\mathcal{C}$ cannot compute MUX_n.
- **Assertion 1.** $\mathcal{C}$ reads $x_i \in D$ at least once.
- **Work: Capture Minimal Gate.** Let α_i be any gate reading x_i.
- **Case 2: Short-Circuit at α_i.** Suppose $(\neg)\alpha_i$ is the output gate. Setting $x_i \leftarrow b$ which eliminates α_i via a fixing rule makes the circuit constant. However, MUX, restricted with any set of substitutions to D is not constant (as the complement of D is not empty, and those data bits could be selected by the address).
- **Assertion 2.** The output of the circuit is not $(\neg)\alpha_i$.
- **Work: Eliminate Gates via Data Bit Restrictions.** Let β_i be another binary gate reading α_i. Setting x_i to fix α_i eliminates at least α_i and β_i. Since $|D| = 2^{n-1}$, processing all data bits in D eliminates at least $2 \cdot 2^{n-1} = 2^n$ gates.

Case 3: Address degeneracy. After substituting all 2^{n-1} variables in D, suppose that the resulting circuit does not read a_1. Then the circuit output is independent of a_1, whereas the restricted multiplexer still depends on a_1. In particular, it determines whether the selected data bit lies in the free lower or fixed upper half. Hence, $\mathcal{C}$ cannot compute MUX_n.

Assertion 3. After substitutions of the data variables in D, the circuit reads a_1 at least once.

Work: Eliminate One More Gate. Substitute $a_1 \leftarrow 0$ and simplify. The resulting circuit has lost at least one more binary gate.

Inductive Hypothesis. The simplified circuit now is a multiplexer on the remaining 2^{n-1} inputs which are addressed by $\{a_2, \ldots a_n\}$. Thus, we obtain

$$\begin{aligned}
\sigma(\mathcal{C}) &\geq \sigma(\mathsf{MUX}_{n-1}) + 2 \cdot 2^{n-1} + 1 \\
&\geq (2 \cdot 2^{n-1} + (n-1) - 2) + 2 \cdot 2^{n-1} + 1 \\
&= 2 \cdot 2^n + n - 2 \\
&= 2N + \log N - 2.
\end{aligned}$$

◀

Algorithm 1 Refuter for MUX_n for circuits of size less than $2N + \log N - 2$

Input: $\mathcal{C}$ is a normalized circuit, $m \in [n]$, $\vec{d} \in \mathbb{F}_2^N$ such that $\mu(\mathcal{C}) < 2 \cdot 2^m + m - 2$ and $\mathcal{C}$ only reads from $\{a_{n-m+1}, \dots a_n\}$ and $\{x_i \mid \forall i \in [n], i < 2^m\}$

Output: $\vec{w}^A \in \mathbb{F}_2^{\log N}, \vec{w}^X \in \mathbb{F}_2^N$ such that $\mathcal{C}(\vec{w}^A, \vec{w}^X) \neq \vec{w}^X_{\text{INT}(\vec{w}^A)}$ and $\forall i \geq m, \vec{w}^A_i = 0$ and $\vec{w}^X_j = \vec{d}_j$ for all $j > 2^m$

1: **procedure** MUX-REFUTER($\mathcal{C}, m, \vec{d}$)
2: **if** $m = 1$ **then**
3: ▷ *Base Case: $\mathcal{C}$ errs on at least one of these 4 assignments as it is too small* ◁
4: **compute** $\mathcal{C}$ on $(\text{BIN}(1), \vec{d}^X), (\text{BIN}(1), \vec{d}^X + \vec{e}_1), (\text{BIN}(2), \vec{d}^X)$, and $(\text{BIN}(2), \vec{d}^X + \vec{e}_2)$
5: **return** an input on which $\mathcal{C}$ does not agree with MUX
6: $D \leftarrow \{x_i \mid 2^{m-1} < i \leq 2^m\}$
7: **for** each $x_i \in D$ **do**
8: **if** $\mathcal{C}$ does not read x_i **then**
9: ▷ *Case 1: $\mathcal{C}$ is degenerate with respect to x_i but the restricted* MUX *is not* ◁
10: **compute** $\mathcal{C}$ on $(\text{BIN}(i), \vec{d})$ and $(\text{BIN}(i), \vec{d} + \vec{e}_i)$
11: **return** whichever input $\mathcal{C}$ does not agree with MUX on
12: $\alpha \leftarrow$ topological minimal gate reading x_i
13: $\vec{d}_i \leftarrow$ constant whose substitution for x_i fixes α
14: **if** $(\neg)\alpha$ is the output gate of $\mathcal{C}$ **then**
15: ▷ *Case 2: The restricted circuit with $x_i \leftarrow \vec{d}_i$ is constant but* MUX *is not* ◁
16: **compute** $\mathcal{C}$ on $(\text{BIN}(2^{m-1}), \vec{d})$ and $(\text{BIN}(2^{m-1}), \vec{d} + \vec{e}_{2^{m-1}})$
17: **return** whichever input $\mathcal{C}$ does not agree with MUX on
18: $\mathcal{C} \leftarrow \mathcal{C}$ with $\vec{d}_i$ substituted for x_i and then simplified
19: **if** $\mathcal{C}$ does not read a_{n-m+1} **then**
20: ▷ *Case 3: $\mathcal{C}$ is degenerate with respect to a_{n-m+1} after setting all $x \in D$* ◁
21: $d_1 \leftarrow d_{2^{m-1}+1} + 1$
22: **compute** $\mathcal{C}$ on $(\text{BIN}(1), \vec{d})$ and $(\text{BIN}(2^{m-1} + 1), \vec{d})$
23: **return** whichever input $\mathcal{C}$ does not agree with MUX on
24: $\mathcal{C} \leftarrow \mathcal{C}$ with a_{n-m+1} substituted by 0 and simplified
25: **return** MUX-REFUTER($\mathcal{C}, m-1, \vec{d}$)

4 Efficient Refuters from Gate Elimination

4.1 A Refuter for MUX

We now show that our elementary lower bound for MUX is constructive by exhibiting an efficient refuter against circuits that violate our lower bound (Algorithm 1).

▶ **Theorem 13.** *Let C be a DeMorgan circuit on $n + N$ inputs, where $N = 2^n$, with size $s < 2N + \log N - 2$ that claims compute MUX_n. We can construct an input (α, ρ) with $\alpha \in \{0,1\}^n$, $\rho \in \{0,1\}^N$, and $N = 2^n$, such that $C(\alpha, \rho) \neq \mathsf{MUX}_n(\alpha, \rho)$ in polynomial time with respect to N.*

Proof. To begin, we clarify the notation used in the algorithm and its inputs. The procedure MUX-REFUTER($\mathcal{C}, m, \vec{d}$) operates on a restricted instance of the multiplexer where the least significant m address bits remain unfixed. Concretely, the live address variables are $\{a_{n-m+1}, \dots, a_n\}$, while the remaining address bits have already been fixed to 0. The vector $\vec{d} \in \mathbb{F}_2^N$ represents an assignment to the data inputs, where the entries $\vec{d}_i$ for $i \leq 2^m$

correspond to the currently live data inputs, while entries for $i > 2^m$ are fixed and will not be modified by the algorithm.

For an index $i \in [2^m]$, $\textsc{bin}(i)$ denotes the binary encoding of i on the m live address bits, i.e., the assignment to $\{a_{n-m+1}, \ldots, a_n\}$ that selects the i-th live data input. Conversely, for an address vector $\vec{w}^A$, $\textsc{int}(\vec{w}^A)$ denotes the integer index selected by that address, so that the multiplexer outputs the data bit $\vec{w}^X_{\textsc{int}(\vec{w}^A)}$. For any $i \leq 2^m$, we have $\mathsf{MUX}(\textsc{bin}(i), \vec{d}) = \vec{d}_i$.

We argue that Algorithm 1 correctly outputs an error by induction on $m \in [n]$, the number of current unfixed address bits. In the base case, when $m = 1$, the algorithm brute forces over the four relevant assignments and returns one on which $\mathcal{C}$ disagrees with MUX. This is correct by the **Base Case** of Theorem 12, since any circuit with fewer than $2 \cdot 2^1 + 1 - 2 = 3$ costly gates cannot compute MUX_1. For the inductive case, fix some $1 < m \leq n$ and assume that MUX-REFUTER correctly outputs an error on all inputs corresponding to addresses of length $m - 1$. Let $D = \{x_i \mid 2^{m-1} < i \leq 2^m\}$. The subsequent branches of the algorithm correspond to the cases identified in the proof of Theorem 12. If the algorithm does not enter one of these branches, the corresponding assertions from the lower bound proof hold for $\mathcal{C}$.

Branch at step 8 corresponds to Case 1 of Theorem 12. Suppose the algorithm finds some $x_i \in D$ that $\mathcal{C}$ does not read. The circuit $\mathcal{C}$ does not depend on x_i, but the current restricted multiplexer does. The algorithm considers the two inputs $(\textsc{bin}(i), \vec{d})$ and $(\textsc{bin}(i), \vec{d} + \vec{e}_i)$. The circuit outputs the same value on both, while MUX and is not. Hence one of them is an error, and the algorithm returns it.

Branch at step 14 corresponds to Case 2 of Theorem 12. The algorithm chose the topological minimal gate α reading x_i and found that $(\neg)\alpha$ is the output gate. After setting x_i to a value that fixes α, the restricted circuit becomes constant, while the restricted multiplexer is not. The algorithm considers inputs that differ on a data bit still relevant to the multiplexer and returns the one on which $\mathcal{C}$ disagrees with MUX.

Branch at step 19 corresponds to Case 3 of Theorem 12. Suppose that, after all variables in D have been processed, the algorithm finds that $\mathcal{C}$ does not read a_{n-m+1}. The circuit is independent of a_{n-m+1}, but the restricted multiplexer still depends on this bit, since it decides whether the selected data input lies in the lower or upper half. The addresses $\textsc{bin}(1)$ and $\textsc{bin}(2^{m-1} + 1)$ differ only in a_{n-m+1}, so $\mathcal{C}$ outputs the same value on them. The algorithm sets $\vec{d}_1 = \vec{d}_{2^{m-1}+1} + 1$, forcing MUX to evaluate differently on these two addresses. Thus one of the two must be a valid error.

Otherwise, the algorithm follows the recursive reduction from the **Inductive Hypothesis** of Theorem 12. Since the preceding branches were not entered, Assertions 1–3 from the lower bound proof hold. Thus each substitution for $x_i \in D$ eliminates at least two gates, and substituting $a_{n-m+1} \leftarrow 0$ eliminates at least one additional gate. Therefore the circuit $\mathcal{C}'$ passed to the recursive call satisfies

$$\mu(\mathcal{C}') < \big(2 \cdot 2^m + m - 2\big) - (2^m + 1) = 2 \cdot 2^{m-1} + (m - 1) - 2.$$

Moreover, after setting $a_{n-m+1} \leftarrow 0$, the remaining function is the restricted multiplexer on the remaining $m - 1$ live address bits and 2^{m-1} live data bits. Hence, the recursive call is valid. By the induction hypothesis, it returns an input on which the simplified circuit errs. Since the error only differs on active bits, the same input is also an error for the current circuit $\mathcal{C}$. Thus, Algorithm 1 correctly returns a valid witness $(\vec{w}^A, \vec{w}^X)$ such that $\mathcal{C}(\vec{w}^A, \vec{w}^X) \neq \vec{w}^X_{\textsc{int}(\vec{w}^A)}$ for all $m \in [n]$ by the principle of mathematical induction as desired.

Finally, each recursive level performs only polynomial-time operations: checking whether a variable is read, finding a topological minimal gate, evaluating the circuit on a constant

378 number of inputs, substituting, and simplifying. The recursion depth is at most $n = \log N$,
379 so the total running time is polynomial in N. ◀

380 4.2 An Affine Refuter for Dimension d

381 The $\mathbb{B}_2$ $3n - o(n)$ lower bound of [5] for affine dispersers is constructive. For any $d : \mathbb{N} \to \mathbb{N}$,

382 ▶ **Theorem 14.** *There is an affine refuter for dimension d against $\mathbb{B}_2$ $3n - 4d(n)$ size circuits.*

383 This demonstrates that even gate elimination arguments which use (1) complex measures
384 other than circuit size and (2) substitutions other than simple constant substitutions are
385 constructive. To show this, we prove a stronger property: given any affine space of high
386 enough dimension, we can find a subset of that space on which the circuit is constant. Recall
387 from [5] that $\mu(\mathcal{C}) = \sigma(\mathcal{C}) + \eta(\mathcal{C})$. We will abuse notation and write d rather than $d(n)$.

388 ▶ **Lemma 15.** *Fix $d : \mathbb{N} \to \mathbb{N}$. Let $A \in \mathbb{F}_2^{n\times d'}$ and $\vec{a} \in \mathbb{F}_2^{d'}$ define an affine space $S = \{A\vec{y} + \vec{a}\}$*
389 *of dimension $d' > d$. For any circuit $\mathcal{C}$ such that $\mu(\mathcal{C}) < 4(d' - d)$, Algorithm 2 outputs*
390 *$\{Wx + \vec{w}\} \subseteq S$ of dimension at least d on which $\mathcal{C}$ is constant in polynomial times.*

391 To obtain an affine refuter, simply run Algorithm 2 with $A = \mathbf{I}_n$ and $\vec{a} = \vec{0}$. Algorithm 2
392 outputs an affine subspace of $\mathbb{F}_2^n$ of degree at least d on which $\mathcal{C}$ is constant. The algorithm
393 mirrors the proof of the following theorem from [5].

394 ▶ **Theorem 16** (from [5])**.** *Let $\mathsf{f} : \{0,1\}^n \to \{0,1\}$ be an affine disperser of dimension d and*
395 *S be a affine space of dimension $d' > d$. If $\mathcal{C}$ computes f on S then $\mu(\mathcal{C}) \geq 4(d' - d)$.*

396 We begin by labeling proof of this theorem the proof as in the proof of Lemma 1.

397 **Proof of Theorem 16 from [5].** Let $\mathcal{C}$ be a circuit computing f on an affine space $S =$
398 $\{A\vec{y} + \vec{a}\}$ of dimension at least d. If the dimension is exactly d then the lower bound trivially
399 holds. Otherwise we proceed by induction on the dimension of S. Fix $d' \geq d + 1$.

400 **Case 1: Computes (¬)XOR.** Suppose for the sake of contradiction, $\mathcal{C}$ only contains fanout
401 1 $\oplus$-type gates. Then $\mathcal{C}$ computes $\bigoplus_{i\in I} x_i \oplus b$ for some $I \subset [n]$ and $c \in \{0,1\}$. However
402 then $\mathcal{C}$ outputs constant 0 on $S \cap \{\vec{x} \in \mathbb{F}_2^n \mid \bigoplus_{i\in I} x_i = b\}$, an affine subspace whose
403 dimension is at least $d' - 1 \geq d$
404 **Assertion 1.** $\mathcal{C}$ contains a gate which is not a fanout 1 $\oplus$-type gate.
405 **Work.** Let α be the topologically minimal gate which is not an $\oplus$-type gate with fanout 1.
406 **Cases over α.** In each case, we find a substitution that reduces μ by at least 4 and the
407 resulting circuit computes f on an affine space of dimension at least $d' - 1$. Note if it
408 computes a circuit on computes f on an affine space of dimension at least $d' - 1$, it
409 computes f on all subsets of that space with dimension exactly $d' - 1$.
410 **Case 2: α is $\oplus$-type with fanout at least 2.** $\mathcal{C}$ computes compute $\bigoplus_{i\in I} x_i \oplus b$ for some
411 $I \subseteq [n]$ and $b \in \{0,1\}$. Substituting $x_j \leftarrow \bigoplus_{i\in I\setminus\{j\}} x_i \oplus b$ for some $j \in I$ fixes α to
412 output 0 and reduces η by 1. Replacing α by 0, eliminates at least the gates reading α
413 reducing σ by at least 3. Let $\mathcal{C}'$ be the resulting circuit.
414 **Case 2.1: Reduce Measure.** The circuit $\mathcal{C}'$ satisfies $\mu(\mathcal{C}') \leq \mu(\mathcal{C}) - 4$ and $\mathcal{C}'$ computes
415 f on $S \cap \{\vec{x} \in F_2^n \mid \bigoplus_{i\in I} x_i = b\}$, an affine space of dimension at least $d' - 1$.
416 **Case 3: α is a $\wedge$-type gate fed by an $\oplus$-type gate β.** The gate β computes $\bigoplus_{i\in I} x_i \oplus$
417 c for some I and c. Let b be the value that fixes α. Substituting $x_j \leftarrow \bigoplus_{i\in I\setminus\{j\}} x_i \oplus$
418 $(b \oplus c)$ fixes β to output b which fixes α. Let $\mathcal{C}'$ be the resulting circuit.

Case 3.1: α is the output. Suppose for the sake of contradiction α is the output. The function f would be constant on $S \cap \{\vec{x} \in \mathbb{F}_2^n \mid \bigoplus_{i \in I} x_i = b \oplus c\}$ which is an affine space of dimension at least d.

Assertion 3.1. α is not the output.

Case 3.2: Reduce Measure The substitution eliminates at least 3 gates. $\mu(\mathcal{C}')$ for the resulting circuit $\mathcal{C}'$ is at least 4 less than $\mu(\mathcal{C})$, and $\mathcal{C}'$ agrees with f on the affine space $S \cap \{\vec{x} \in \mathbb{F}_2^n \mid \bigoplus_{i \in I} x_i = b \oplus c\}$.

Case 4: α is fed by two variables x_p and x_q Assume the fanout of x_p is at least x_q. Let b be the value that fixes α. Substituting $x_p \leftarrow \beta$ and simplifying will produce a circuit $\mathcal{C}'$ where $\mu(\mathcal{C}') \leq \mu(\mathcal{C}) - 4$. We elide the complete case analysis here, see [5] for more details. Each subcase (which depends on the fanout of x_p, the fanout of α, and which specific gates x_p feeds) reduces the complexity by at least four unless α is the output of the circuit.

Case 4.1 α is the output. Suppose for the sake of contradiction α is the output. Then f would be constant on $S \cap \{\vec{x} \in \mathbb{F}_2^n \mid x_p = c\}$

Assertion 4.1 α is the not the output of the circuit.

Case 4.2: Reduce Measure Lastly, $\mathcal{C}'$ computes f on $S \cap \{\vec{x} \in \mathbb{F}_2^n \mid x_p = c\}$, an affine space of dimension at least $d' - 1$ and $\mu(\mathcal{C}') \leq \mu(\mathcal{C}) - 4$.

Case 5: Invoke Inductive Hypothesis Each case resolves by reducing μ by at least 4 and the simplified $\mathcal{C}'$ computes f on an affine subspace of dimension $d' - 1 \geq d$. By induction, $\mu(\mathcal{C}) \geq \mu(\mathcal{C}') + 4 \geq 4((d' - 1) - d) = 4(d' - d)$. ◀

We now prove that Algorithm 2 correctly outputs an affine subspace of sufficient size on which its input circuit is constant.

Proof of Lemma 15. Fix $d : \mathbb{N} \to \mathbb{N}$. Let $A \in \mathbb{F}_2^{d'}$ and $\vec{a} \in \mathbb{F}_2^{d'}$ define an affine space S of dimension $d' > d$. Let $\mathcal{C}$ be a circuit where $\mu(\mathcal{C}) < 4(d' - d)$. We will show that Algorithm 2, FindConstantSubspace($\mathcal{C}, A, \vec{a}$), outputs a space S' of dimension $\geq d$ such that $S' \subseteq S$ in polynomial time with respect to n and $\mathcal{C}$ is constant on S'. FindConstantSubspace uses four polynomial time subroutines defined in Appendix B. We summarize these in Table 1.

Table 1 The four polynomial subroutines used by FindConstantSubspace

Procedure	Description
ParitySupport	Given $\mathcal{C}$ with only $\oplus$-type gates of fanout 1 Returns I, c such that $\mathcal{C}$ computes $\bigoplus_{i \in I} x_i \oplus c$
AffineIntersect	Given two affine spaces S and S' of dimension d' and $n - 1$ Returns a description of their dimension $\geq d' - 1$ intersection.
ConstraintToAffine	Given (I, c) where $I \subseteq [n]$ and $c \in \{0, 1\}$ Returns the description of the affine space $\{x_i \in \mathbb{F}_2^n \mid \bigoplus_{i \in I} x_i = c\}$
FindSubstitution	Given ($\mathcal{C}$, b), a circuit with only $\oplus$-type gates of fanout 1 Returns the substitution $x_j \leftarrow \mathcal{S}$ that makes $\mathcal{C}$ output b

It is clear that if the algorithm terminates then the output is a subset of S, as in each call we only take the *intersection* of affine spaces, which are subsets of our starting space. We argue that each iteration, if it does not return in branches 2 or 5, reduces $\mu(\mathcal{C})$ by at least 4 or makes the circuit constant. This follows as the algorithm mirrors the structure of the proof of theorem 16; subsequent branches correspond to the cases over α.

Algorithm 2 Searches for an affine subspace on which $\mathcal{C}$ is constant

Input: $\mathcal{C}$ is a normalized circuit, $A \in \mathbb{F}_2^{n \times d'}$, $\vec{a} \in \mathbb{F}_2^n$, such that A is full column rank, $d' > d$, and $\mu(\mathcal{C}) < 4(d' - d)$ or $\mu(\mathcal{C}) = 1$.
Output: $W, \vec{w}$ such that $\mathcal{C}$ is constant on $\{W\vec{x} + \vec{w}\} \subseteq \{A\vec{y} + \vec{a}\}$ and $\dim W \geq d$

1: **procedure** FindConstantSubspace($\mathcal{C}$, A, $\vec{a}$)
2: **if** $\mathcal{C}$ is constant **then** ▷ *Case 3.1 and 4.1*
3: **return** A, $\vec{a}$
4: **if** $\mathcal{C}$ is a circuit of $\oplus$-type gates with fan-out 1 **then** ▷ *Case 1*
5: $I, c \leftarrow$ ParitySupport($\mathcal{C}$) ▷ *$\mathcal{C}$ computes $\bigoplus_{i \in I} x_i \oplus c$*
6: $R, \vec{r} \leftarrow$ ConstraintToAffine(I, c) ▷ $\{Rz + \vec{r}\} = \{x \in \mathbb{F}_2^n \mid \bigoplus_{i \in I} x_i = c\}$
7: **return** AffineIntersect($A, \vec{a}, R, \vec{r}$) ▷ *C is constantly 0 on this intersection*
8: $\alpha \leftarrow$ topologically minimal non-fanout 1 $\oplus$-type gate of $\mathcal{C}$
9: **if** α is an $\oplus$-type gate **then** ▷ *Case 2*
10: $\mathcal{P} \leftarrow$ the induced sub-circuit of $\mathcal{C}$ rooted at α
11: $I, c \leftarrow$ ParitySupport($\mathcal{P}$) ▷ *$\mathcal{P}$ computes $\bigoplus_{i \in I} x_i \oplus c$*
12: $\mathcal{S}, j \leftarrow$ FindSubstitution($\mathcal{P}, 0$) ▷ $\mathcal{S} = \bigoplus_{i \in I \setminus \{j\}} x_i \oplus c$
13: $\mathcal{C} \leftarrow \mathcal{C}$ where x_j is substituted with $\mathcal{S}$, α is replaced by 0, and then simplified
14: $R, \vec{r} \leftarrow$ ConstraintToAffine(I, c)
15: **else if** a $\oplus$-type gate β feeds α **then** ▷ *Case 3*
16: $\mathcal{Q} \leftarrow$ induced sub-circuit of $\mathcal{C}$ rooted at β
17: $c' \leftarrow$ the constant value that fixes α
18: $I, c \leftarrow$ ParitySupport($\mathcal{Q}$) ▷ *$\mathcal{Q}$ computes $\bigoplus_{i \in I} x_i \oplus c$*
19: $\mathcal{S}, j \leftarrow$ FindSubstitution($\mathcal{Q}, c'$) ▷ $\mathcal{S} = \bigoplus_{i \in I \setminus \{j\}} x_i \oplus (c + c')$
20: $\mathcal{C} \leftarrow \mathcal{C}$ where x_j is substituted with $\mathcal{S}$, β is replaced by c', and then simplified
21: $R, \vec{r} \leftarrow$ ConstraintToAffine($I, c \oplus c'$)
22: **else** ▷ *Case 4: α is fed by two variables*
23: $x_j \leftarrow$ the variable feeding α with maximal fanout
24: $c' \leftarrow$ the constant value that fixes α
25: $\mathcal{C} \leftarrow \mathcal{C}$ where x_j is substituted with c' and then simplified
26: $R, \vec{r} \leftarrow$ ConstraintToAffine($\{j\}, c'$)
27: $A, \vec{a} \leftarrow$ AffineIntersect($A, \vec{a}, R, \vec{r}$)
28: **return** FindConstantSubspace($\mathcal{C}, A, \vec{a}$)

Branch at step 13 corresponds to Case 2. The substitution from the proof removes at least three gates and reduces $\eta(\mathcal{C})$ by at least 1.
Branch at step 21 corresponds to Case 3. The substitution from the proof either reduces μ by four or makes the circuit constant.
Branch at step 30 corresponds to Case 4. The substitution from the proof either reduces μ by four or makes the circuit constant.

If, $\mathcal{C}$ becomes constant as in Cases 3.1 or 4.1, the next recursive call enters branch 2 and returns the intersection on which $\mathcal{C}$ is constant described in the proof. Otherwise, $\mu(\mathcal{C})$ is reduced by 4. Notice that we can make at most $(d' - d)$ iterations before $\mathcal{C}$ must become constant. Hence the algorithm terminates and returns an affine space on which $\mathcal{C}$ is constant. During each iteration, ConstraintToAffine, reduces the dimension by at most 1, hence the returned space has dimension at least d.

Finally, as each subroutine runs in polynomial time and we make at most $d' - d$ iterations, FindConstantSubspace runs in polynomial time. ◀

466 ## References

467 **1** Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of*
468 *the ACM (JACM)*, 42(1):269–291, January 1995. doi:10.1145/200836.200880.
469 **2** Norbert Blum. A boolean function requiring 3n network size. *Theor. Comput. Sci.*, 28:337–345,
470 1984. doi:10.1016/0304-3975(83)90029-4.
471 **3** Marco Carmosino, Ngu Dang, and Tim Jackman. Simple Circuit Extensions for XOR in
472 PTIME. In Meena Mahajan, Florin Manea, Annabelle McIver, and Nguyen Kim Thang, editors,
473 *43rd International Symposium on Theoretical Aspects of Computer Science (STACS 2026)*,
474 volume 364 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:20,
475 Dagstuhl, Germany, 2026. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https:
476 //drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2026.23, doi:10.4230/
477 LIPIcs.STACS.2026.23.
478 **4** Lijie Chen, Ce Jin, Rahul Santhanam, and Ryan Williams. Constructive separations and their
479 consequences. *TheoretiCS*, 3, 2024. URL: https://doi.org/10.46298/theoretics.24.3,
480 doi:10.46298/THEORETICS.24.3.
481 **5** Evgeny Demenkov and Alexander S. Kulikov. An elementary proof of a 3n - o(n) lower bound
482 on the circuit complexity of affine dispersers. In Filip Murlak and Piotr Sankowski, editors,
483 *Mathematical Foundations of Computer Science 2011 - 36th International Symposium, MFCS*
484 *2011, Warsaw, Poland, August 22-26, 2011. Proceedings*, volume 6907 of *Lecture Notes in*
485 *Computer Science*, pages 256–265. Springer, 2011. doi:10.1007/978-3-642-22993-0_25.
486 **6** Magnus Gausdal Find, Alexander Golovnev, Edward A. Hirsch, and Alexander S. Kulikov. A
487 better-than-3n lower bound for the circuit complexity of an explicit function. In *2016 IEEE*
488 *57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 89–98, 2016.
489 doi:10.1109/FOCS.2016.19.
490 **7** Alexander Golovnev, Edward A. Hirsch, Alexander Knop, and Alexander S. Kulikov. On the
491 limits of gate elimination. *J. Comput. Syst. Sci.*, 96:107–119, 2018. URL: https://doi.org/
492 10.1016/j.jcss.2018.04.005, doi:10.1016/J.JCSS.2018.04.005.
493 **8** Stefan Grosser and Marco Carmosino. Student-teacher constructive separations and
494 (un)provability in bounded arithmetic: Witnessing the gap. In Michal Koucký and Nikhil
495 Bansal, editors, *Proceedings of the 57th Annual ACM Symposium on Theory of Com-*
496 *puting, STOC 2025, Prague, Czechia, June 23-27, 2025*, pages 1341–1347. ACM, 2025.
497 doi:10.1145/3717823.3718216.
498 **9** Rahul Ilango. Constant depth formula and partial function versions of MCSP are hard.
499 *SIAM J. Comput.*, 53(6):S20–317, 2024. URL: https://doi.org/10.1137/20m1383562, doi:
500 10.1137/20M1383562.
501 **10** Stasys Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms*
502 *and combinatorics*. Springer, 2012. doi:10.1007/978-3-642-24508-4.
503 **11** Klein and Paterson. Asymptotically optimal circuit for a storage access function. *IEEE*
504 *Transactions on Computers*, 100(8):737–738, 1980.
505 **12** Jiatu Li and Tianqi Yang. $3.1n - o(n)$ circuit lower bounds for explicit functions. In Stefano
506 Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium*
507 *on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1180–1193. ACM, 2022.
508 doi:10.1145/3519935.3519976.
509 **13** SA Lozhkin and DE Khzmalyan. The complexity of the standard multiplexer function in a
510 class of switching circuits. *Computational Mathematics and Modeling*, 32(4):478–489, 2021.
511 **14** Moritz Müller and Ján Pich. Feasibly constructive proofs of succinct weak circuit lower bounds.
512 *Ann. Pure Appl. Log.*, 171(2), 2020. URL: https://doi.org/10.1016/j.apal.2019.102735,
513 doi:10.1016/J.APAL.2019.102735.
514 **15** Wolfgang J Paul. A 2.5 n-lower bound on the combinational complexity of boolean functions.
515 In *Proceedings of the seventh annual ACM symposium on Theory of computing*, pages 27–36,
516 1975.

517 **16** NP Red'kin. Proof of minimality of circuits consisting of functional elements. *Systems Theory*
518 *Research: Problemy Kibernetiki*, pages 85–103, 1973.
519 **17** Claus-Peter Schnorr. Zwei lineare untere schranken für die komplexität boolescher funktionen.
520 *Computing*, 13(2):155–171, 1974. doi:10.1007/BF02246615.

A An Optimal XOR Checker

Refuters find errors for computational models which fall *below* a threshold established from a lower bound. It is natural to ask whether one can still find errors the model *meets* the threshold but still fails to compute f. We call such an algorithm a *checker*. We show that checkers exist for XOR. It will be useful to keep in mind the following facts about XOR.

Properties of $(\neg)$XOR**.** $(\neg)\mathsf{XOR}_n$ is fully downward self-reducible: fixing any subset of inputs leaves $(\neg)$XOR on the remaining variables. Second, $(\neg)\mathsf{XOR}_n$ strongly depends on every input: flipping any single bit always flips the output. Taken together, every partial restriction of $(\neg)\mathsf{XOR}_n$ remains non-degenerate.

▶ **Theorem 17.** *Let C be a DeMorgan circuit on n inputs with size $s = 3(n-1)$ that **does not** compute $(\neg)\mathsf{XOR}_n$. Then, an input $x \in \{0,1\}^n$ such that $C(x) \neq (\neg)\mathsf{XOR}_n(x)$ can be found in polynomial time with respect to n.*

While refuters follow straightforwardly from proofs via gate elimination, Theorem 17 does not. It leverages a strong structural characterization of the circuits which compute XOR. Section A.1 establishes a refuter subroutine for XOR which serves as the basis for the checker. In Section A.2, we show that we can *detect* optimal $(\neg)$XOR circuits due to to their rigid structure. Finally, in Section A, we leverage the refuter and the detectability of XOR to obtain our checker.

A.1 A Refuter for XOR

We begin with an XOR refuter (Algorithm 3), which our checker can both use as a subroutine, and whose procedure captures many ways in which an appropriately sized circuit can still "trivially" fail to compute XOR.

▶ **Theorem 18.** *Let C be a DeMorgan circuit on $n \geq 2$ inputs with size $s < 3(n-1)$. We can find an input $x \in \{0,1\}^n$ such that $C(x) \neq \mathsf{XOR}_n(x)$ in polynomial time with respect to n.*

Proof. Let $\mathcal{C}$ be a DeMorgan circuit on n inputs with size $s < 3(n-1)$ and let f be the function it computes. We will show Algorithm 3, when run on $\mathcal{C}$, $I = [n], b = 0(1$ for $(\neg)\mathsf{XOR}), \vec{a} = \vec{0}$ outputs an input on which $\mathcal{C}$ errs in polynomial time. Termination and correctness of the algorithm follow from the fact that it follows the structure of Schnorr's $3(n-1)$ lower bound, modulo the proof that $\mathcal{C}$ actually errs on one of the inputs at each **compute** step. We recall Schnorr's argument, structuring it so that the underlying algorithm is clear.

$(\neg)$XOR **Lower Bound from [17].** As a proof via gate elimination, the argument proceeds via induction. The base case is trivial as $(\neg)\mathsf{XOR}_1(x) \equiv (\neg)x$ and $3(1-1) = 0$. Let C be a normalized circuit with $n \geq 2$ inputs.

Case 1: Obvious Degeneracy. Suppose C does not read some x_i. Then C cannot depend on x_i, and therefore does not compute XOR_n, which depends on all n variables.

Assertion 1. C reads every input variable at least once.

Work: Capture the First Gate. Let α be the topologically minimal binary gate in C. Because C is in normal form, α reads $(\neg)x_i$ and $(\neg)x_j$ for some *distinct* $i, j \in [n]$.

Case 2: Insufficient Fanout. Without loss of generality, fix i and suppose α is the *only* costly gate reading from x_i, so the fanout of x_i is 1. If we substitute x_j to fix α and simplify, x_i becomes disconnected from C. The resulting circuit cannot compute $(\neg)\mathsf{XOR}_{n-1}$, which still depends on x_i. Therefore C cannot compute XOR_n.

Assertion 2. The fanout of x_i is strictly greater than 1.

Case 3: Short-Circuit at α. Suppose α is the output gate. Without loss of generality, fix i and substitue x_j to eliminate α via a fixing rule. This trivializes the entire circuit C by simplifying it to a constant. Therefore, C cannot compute XOR_n as a single bit restriction of XOR_n is $(\neg)\mathsf{XOR}_{n-1}$. This is the same contradiction obtained above.

Assertion 3. Gate α is not the output.

Work: Capture a Second Gate. Let β be a topologically minimal binary gate distinct from α that also reads x_i in C. β must exist because the fanout of x_i is strictly greater than 1.

Case 4: Short-Circuit at β. Suppose β is the output gate and argue exactly as in Case 3 using x_i to fix β.

Assertion 4. Gate β is not the output.

Work: Capture a Third Gate. Let γ be the topologically minimal binary gate that reads β. Fix x_i to fix β and simplify. This eliminates α, β and γ, reducing the circuit size by 3.

Invoke Inductive Hypothesis. $\mathcal{C}$ now computes $(\neg)\mathsf{XOR}_{n-1}$ and its size has reduced by at least three. Applying our inductive hypothesis, the original circuit had size at least $3 + \sigma((\neg)\mathsf{XOR}_{n-1}) = 3 + 3((n-1) - 1) = 3n - 3 = 3(n-1)$. ◀

We now argue that the algorithm outputs an error. Each recursive call of the algorithm corresponds to the inductive step for $|I| = k \leq n$. In the base case, when $k = 2$ and $\sigma(\mathcal{C}) < 3$, the algorithm in steps 2 - 6 brute forces over all four possible assignments and returns one on which it errs by construction. We proceed via induction; fix $k > 2$ to be the size of I in our input and presume XOR-Refuter correctly outputs on all inputs where $|I| = k - 1$. The algorithm does not enter the branch at step 2 as $|I| = k > 2$. Subsequent branches correspond to the Cases identified in Schnorr's proof. If we do not enter these branches then the corresponding Assertions of Schnorr's proof hold for $\mathcal{C}$. Hence any gates and variables bound in the interleaving steps exist.

Branch at step 7 corresponds to Case 1 of Schnorr's proof. $\mathcal{C}$ does not depend on x_i but $\mathsf{XOR}_I(\cdot) \oplus c$ does. In step 9, $\mathcal{C}$ evaluates the same on both inputs but $\mathsf{XOR}_I(\cdot) \oplus c$ does not. The algorithm returns whichever is incorrect.

Branch at step 13 corresponds to Case 2 and 3. Setting x_q to fix α in $\mathcal{C}$ either disconnecting x_p or makes the circuit constant. $\mathcal{C}$ evaluates the same on both inputs unlike $\mathsf{XOR}_I(\cdot) \oplus c$.

Branch at step 19 corresponds to Case 4. Setting x_p as described leaves $\mathcal{C}$ to be a constant, but $\mathsf{XOR}_I(\cdot) \oplus c$ does not become constant. In this and the previous branches, the only indices of $\vec{a}$ that are changed are in I, and hence the condition that the error agrees with the original $\vec{a}_i$ on all non-I inputs is met.

Otherwise, the algorithm in steps 25 - 28 follows the final Work and Inductive Hypothesis step of the proof. As Assertions 1-4 hold for $\mathcal{C}$, setting x_p to fix β eliminates at least three gates, and the resulting circuit is now too small to compute the appropriate parity of the $k - 1$ bits indexed by $I \setminus \{p\}$. Therefore the recursive call returns an input on which the simplified circuit errs that is consistent with $\vec{a}_p$ (and all $\vec{a}_i$ where $i \notin I$). Hence $\mathcal{C}$ also errs on this input as desired and this error is consistent with $\vec{a}$ on all non-I indices. Algorithm 3 is therefore correct.

Each iteration can clearly be computed in polynomial time, as $\sigma(\mathcal{C}) < 3(n-1)$ and the normalization of $\mathcal{C}$ ensure that evaluating, substituting, and simplifying can all be done in polynomial time. ◀

Algorithm 3 Refuter for circuits purportedly computing $(\neg)\mathsf{XOR}_n$ with fewer than $3(n-1)$ gates

Input: $\mathcal{C}$ is a normalized circuit, $I \subseteq [n]$, $c \in \{0,1\}$, $\vec{a} \in \mathbb{F}_2^n$, such that $\sigma(\mathcal{C}) < 3(|I|-1)$ and $\mathcal{C}$ only reads variables in I

Output: $\vec{w} \in \mathbb{F}_2^n$ such that $\mathcal{C}(\vec{w}) \neq \mathsf{XOR}_I(\vec{w}) \oplus c$ and $\vec{w}_i = \vec{a}_i$ for all $i \notin I$

1: **procedure** XOR-Refuter($\mathcal{C}, I, c, \vec{a}$)
2:   **if** $|I| = 2$ **then**
3:     ▷ *Base Case: only 4 assignments possible and $\mathcal{C}$ must err on one* ◁
4:     $i, j \leftarrow$ elements of I
5:     **compute** $\mathcal{C}$ on $\vec{a}, \vec{a}+\vec{e}_i, \vec{a}+\vec{e}_j, \vec{a}+\vec{e}_i+\vec{e}_j$
6:     **return** an input on which $\mathcal{C} \neq \mathsf{XOR}_I$
7:   **if** $\mathcal{C}$ does not read x_i for some $i \in I$ **then**
8:     ▷ *Case 1: $\mathcal{C}$ is degenerate with respect to x_i but XOR_I is not* ◁
9:     **compute** $\mathcal{C}$ on $\vec{a}$ and $\vec{a}+\vec{e}_i$
10:     **return** whichever input $\mathcal{C}$ does not agree with $\mathsf{XOR}_I(\cdot) \oplus c$ on
11:   $\alpha \leftarrow$ topological minimal binary gate in $\mathcal{C}$
12:   $p, q \leftarrow$ indices of I such that α reads $(\neg)x_p$ and $(\neg)x_q$
13:   **if** Fanout$(x_p) = 1$ or $(\neg)\alpha$ is the output **then**
14:     $\vec{a}_q \leftarrow$ constant whose substitution for x_q fixes α
15:     ▷ *Case 2 and 3: $\mathcal{C}\restriction_{x_q \leftarrow \vec{a}_q}$ no longer depends on x_p but $\mathsf{XOR}_{I\setminus\{q\}}$ does* ◁
16:     **compute** $\mathcal{C}$ on $\vec{a}$ and $\vec{a}+\vec{e}_p$
17:     **return** whichever input $\mathcal{C}$ does not agree with $\mathsf{XOR}_I(\cdot) \oplus c$ on
18:   $\beta \leftarrow$ topological minimal binary gate reading $(\neg)x_p$ distinct from α
19:   **if** $(\neg)\beta$ is the output gate of $\mathcal{C}$ **then**
20:     $\vec{a}_p \leftarrow$ constant whose substitution for x_p fixes β
21:     ▷ *Case 4: $\mathcal{C}\restriction_{x_p \leftarrow \vec{a}_p}$ is constant but $\mathsf{XOR}_{I\setminus\{p\}}$ is not* ◁
22:     **compute** $\mathcal{C}$ on $\vec{a}$ and $\vec{a}+\vec{e}_p$
23:     **return** whichever input $\mathcal{C}$ does not agree with $\mathsf{XOR}_I(\cdot) \oplus c$ on
24:   ▷ *We can eliminate at least three gates and recurse* ◁
25:   $\gamma \leftarrow$ topologically minimal binary gate reading $(\neg)\beta$
26:   $\vec{a}_p \leftarrow$ constant whose substitution for x_p fixes β
27:   $\mathcal{C} \leftarrow \mathcal{C}$ with $\vec{x}_p$ substituted for x_p and then simplified
28:   **return** XOR-Refuter($\mathcal{C}, I \setminus \{p\}, b \oplus \vec{a}_p, \vec{a}$)

A.2 Optimal XOR Circuits Are Detectable

Algorithm 4, the XOR checker, initially proceeds in the same way as the refuter. However, when it finds a substitution that eliminates at least three gates, it is possible that the restricted circuit *does* compute the appropriate XOR of the remaining inputs, and $\mathcal{C}$ when the input is restricted with the opposite bit. To circumvent this, the checker will need to be able to detect whether the resulting circuit is now correct. We show this can be done in polynomial time.

▶ **Lemma 19.** *Let $\mathcal{C}$ be a normalized DeMorgan circuit on n variables of size $3(n-1)$. Determining whether $\mathcal{C}$ computes (1)XOR_n, (2)$\neg\mathsf{XOR}_n$, or (3) neither can be computed in polynomial time with respect to n.*

Computing the truth table in exponential in n. Improving over naive brute force is possible because the structure of optimal circuits computing XOR in the DeMorgan basis (when $\neg$ gates do not contribute to circuit size) has been characterized exactly [3].

▶ **Theorem 20** (from [3]). *When $\neg$ gates do not contribute to circuit size, optimal $(\neg)$XOR circuits in the DeMorgan basis are trees of $n-1$ $(\neg)\mathsf{XOR}_2$ widgets.*

We can therefore partition $\mathcal{C}$ into $(\neg)\mathsf{XOR}_2$ widgets. If this is impossible than $\mathcal{C}$ does not compute $(\neg)\mathsf{XOR}_n$. To prevent false positives, we need the following claim which is the converse of Theorem 20.

▶ **Lemma 21.** *Let C be a circuit of size $3(n-1)$ and let $f : \{0,1\}^n \to \{0,1\}$ be the Boolean function it computes. If C can be partitioned into $(\neg)\mathsf{XOR}_2$ widgets then $f \equiv (\neg)\mathsf{XOR}_n$.*

Proof. This follows via strong induction and verifying that taking $(\neg)\mathsf{XOR}_2$ of two circuits computing $(\neg)$XOR on distinct subsets of variables computes $(\neg)$XOR of their union. ◀

The characterization and the lack of false positives together yields the lemma.

Proof. Proof of Lemma 19 To determine whether $\mathcal{C}$ computes XOR_n (or $\neg\mathsf{XOR}_n$), partition the circuit into $n-1$ blocks, each of size 3, as described in the proof of Theorem 20. As there are a finite number of normalized optimal $(\neg)\mathsf{XOR}_2$ blocks, we simply hardcode a list of them in our algorithm to ensure that each block is an $(\neg)\mathsf{XOR}_2$ widget. If any block does not computes $(\neg)\mathsf{XOR}_2$ then we **reject**. Lastly, we determine whether $\mathcal{C}$ computes XOR_n or $\neg\mathsf{XOR}_n$ we can evaluate $\mathcal{C}$ on the all zero input: if it evaluates to 0 then C computes XOR_n rather than $\neg\mathsf{XOR}_n$. Since $\mathcal{C}$ is normalized each of the above steps runs in polynomial time with respect to n. ◀

A.3 An XOR Checker

We now show Theorem 17 by giving an efficient procedure (Algorithm 4) that leverages the refuter (Algorithm 3) and the detectability of XOR.

Proof of Theorem 17. Let $\mathcal{C}$ be a DeMorgan circuit on n inputs of size $3(n-1)$ which does not compute XOR_n. We will argue that XOR-Checker, Algorithm 4, outputs an input on which $\mathcal{C}$ fails to compute XOR_n in polynomial time when run with $I = [n], \vec{a} = \vec{0}$.

XOR-Checker proceeds as the XOR-Refuter up to line 23. In the base case, $n = 2$, the algorithm brute forces over all four possible inputs and $\mathcal{C}$ must err on at least one of them else it computes XOR_2. We proceed via induction; fix $k > 2$ to be the size of I and presume that XOR-Checker is correct on any inputs where $|I| = k-1$.

Algorithm 4 Checker for circuits purportedly computing $(\neg)\mathsf{XOR}_n$

Input: $\mathcal{C}$ is a normalized circuit, $I \subseteq [n]$, $c \in \{0,1\}$, $\vec{a} \in \mathbb{F}_2^n$, such that $\sigma(\mathcal{C}) = 3(|I|-1)$ and $\mathcal{C}$ only reads variables in I and $\mathcal{C}$ does not compute $\mathsf{XOR}_I \oplus c$
Output: $\vec{w} \in \mathbb{F}_2^n$ such that $\mathcal{C}(\vec{w}) \neq \mathsf{XOR}_I(\vec{w}) \oplus c$ and $\vec{w}_i = \vec{a}_i$ for all $i \notin I$

1: **procedure** XOR-Checker($\mathcal{C}, I, c, \vec{a}$)
2: Run lines 2 - 23 of XOR-Refuter
24: ▷ *We can eliminate at least three gates and try to recurse* ◁
25: $\gamma \leftarrow$ topologically minimal binary gate reading $(\neg)\beta$
26: $\vec{a}_p \leftarrow$ constant whose substitution for x_p fixes β
27: $\mathcal{C}' \leftarrow \mathcal{C}$ with $\vec{a}_p$ substituted for x_p and then simplified
28: **if** $|\mathcal{C}'| < |\mathcal{C}| - 3$ **then**
29: ▷ $\mathcal{C}\restriction_{x_p \leftarrow \vec{a}_p}$ *is too small to compute* $(\neg)\mathsf{XOR}_{I\setminus\{p\}}$ ◁
30: **return** XOR-Refuter($\mathcal{C}', I \setminus \{p\}, c \oplus \vec{a}_p, \vec{a}$)
31: **else if** $\mathcal{C}'$ does not computes $\mathsf{XOR}_{I\setminus\{p\}}(\cdot) \oplus (c \oplus \vec{a}_p)$ **then**
32: **return** XOR-Checker($\mathcal{C}', I \setminus \{p\} \oplus \vec{a}_p, \vec{a}$)
33: ▷ $\mathcal{C}\restriction_{x_p \leftarrow \vec{a}_p}$ does *compute* $\mathsf{XOR}_{I\setminus\{p\}}(\cdot) \oplus (c \oplus \vec{a}_p)$*; must flip* $\vec{a}_p$ *to find error* ◁
34: $\vec{a}_p \leftarrow 1 + \vec{a}_p$
35: **if** α only feeds β as in Figure 2 **then**
36: $\tilde{\mathcal{C}} \leftarrow \mathcal{C}$ with α removed and β replaced by an optimal circuit computing the same function as β
37: ▷ $\tilde{\mathcal{C}}$ *computes the same function as* $\mathcal{C}$ *but* $\sigma(\tilde{\mathcal{C}}) < 3(|I|-1)$ ◁
38: **return** XOR-Refuter($\tilde{\mathcal{C}}, I, b, \vec{a}$)
39: ▷ $x_p \leftarrow 1 + \vec{a}_p$ *also eliminates at least three gates* ◁
40: $\mathcal{C} \leftarrow \mathcal{C}$ with x_p substituted by $\vec{a}_p$ and simplified
41: **return** XOR-Checker($\mathcal{C}$, $I \setminus \{p\}, c \oplus \vec{a}_p$, $\vec{a}$)

649 If the algorithm branches before line 24, then $\mathcal{C}$ must err as in the proof of correctness
650 for XOR-Refuter. Otherwise, Assertions 1 - 4 of Schnorr's proof hold for $\mathcal{C}$ and setting x_p
651 to fix β will eliminate at least three gates. However, unlike in XOR-Refuter, we cannot
652 recurse immediately unless more than three gates have been removed (as in step 28). It is
653 possible that the circuit after simplifying, $\mathcal{C}'$, correctly computes $(\neg)\mathsf{XOR}_{n-1}$; $\mathcal{C}$ might only
654 err when x_p is set the other way.

655 At step 31 XOR-Checker, checks whether this is the case and if it is not then it can
656 recurse. Since $\mathcal{C}'$ does not compute the appropriate parity on $I \setminus \{p\}$, the returned value is
657 also incorrect for $\mathcal{C}$. If the algorithm does not return, then XOR-Checker must restrict x_p
658 in the opposite way from Schnorr to find an error. However, this eliminates β via a *passing*
659 rule and does not guarantee γ is removed.

660 If $\mathcal{C}$ is structured as in Figure 2 (negations are omitted) it is possible for only two gates
661 to be removed (e.g. if β and α are both removed via a passing rule). We can however instead
662 argue that $\mathcal{C}$ is non-optimal and reduce its circuit size. Notice the subcircuit rooted at β
663 computes a binary Boolean function of x_p and x_q using two costly gates. This circuit is too
664 small to compute $(\neg)\mathsf{XOR}_2$, and thus must compute a function in $\mathbf{U}_2$. However, all such
665 functions can be computed using at most one binary gate in the DeMorgan basis. Therefore
666 XOR-Checker can use brute force to determine which function β computes, replace β
667 with an optimal circuit for that function, and remove α. This yields a circuit of size strictly
668 less than $3(k-1)$, and the algorithm can call XOR-Refuter to find an input it (and by
669 extension $\mathcal{C}$) err on.

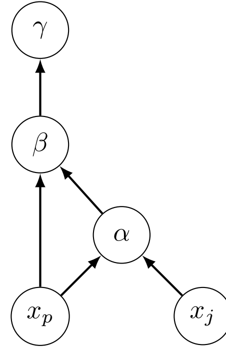


Figure 2 The local structure around x_p where $x_p \leftarrow 1 - \vec{a}_p$ might only eliminate two gates.

670 Otherwise restricting x_p the opposite way is also guaranteed to remove at least three
671 binary gates. Furthermore, the resulting circuit does not compute the appropriate parity of
672 the inputs indexed by $I \setminus \{p\}$ and hence XOR-Refuter returns an input, consistent with $\vec{a}$
673 on p and non-I indices, on which the simplified circuit, and by extension $\mathcal{C}$ err.
674 As detecting optimal XOR_n circuits can be done in polynomial time and XOR-Refuter
675 is a polynomial time refuter, Algorithm 4 also runs in polynomial time. ◀

676 B Subroutines for Affine Refuter

677 In this section we define Algorithms 5 − 8 used by Algorithm 2. For completeness, we show
678 that these algorithms are correct and run in polynomial time.

679 ParitySupport

Algorithm 5 Identifies the affine function by a circuit of fanout 1 ⊕-type gates

Input: C, a normalized circuit of fanout 1 ⊕-type gates
Output: I, c such that C computes $\bigoplus_{i \in I} x_i \oplus c$
1: **procedure** ParitySupport(C)
2: $I \leftarrow \emptyset$
3: **for** $i \in [n]$ **do**
4: **if** fanout(x_i) is odd **then**
5: add i to I
6: **return** $I, C(\vec{0})$

680 **Proof.** Let $\mathcal{C}$ be a normalized circuit whose only gates (if any) are fanout 1 ⊕-type gates.
681 Then C computes the formula where ⊕ is the operator and the inputs are variables and
682 constants. As ⊕ commutes, it is easy to see that every variable which appears in the formula
683 an even number of times is canceled out. Thus $\mathcal{C}$ computes $\bigoplus_{i \in I} x_i \oplus c$ for some $c \in \{0, 1\}$.
684 Furthermore, $\mathcal{C}$, $\mathcal{C}(\vec{0}) = \bigoplus_{i \in I} 0 \oplus c = c$ as desired. Checking the fanout of each x_i and
685 evaluating $\mathcal{C}$ on one input can be done in quadratic time with respect to $|\mathcal{C}|$. ◀

686 AffineIntersect

687 **Proof.** Let $U \in \mathbb{F}_2^{n \times d'}, \vec{u} \in \mathbb{F}_2^n, V \in \mathbb{F}_2^{n \times (n-1)}$, and $\vec{v} \in \mathbb{F}_2^n$ where U and V are full column
688 rank and $\{Ux + \vec{u} \cap \{Vy + \vec{y}\} \neq \emptyset$. We first argue that steps 2 and 3 can be computed.

Algorithm 6 Computes the description for an affine space and linear restriction

Input: $U \in \mathbb{F}_2^{n\times d'}, \vec{u} \in \mathbb{F}_2^n, V \in \mathbb{F}_2^{n\times(n-1)}$, and $\vec{v} \in \mathbb{F}_2^n$ where U and V are full column rank and $\{U\vec{x}+\vec{u}\} \cap \{V\vec{y}+\vec{v}\} \neq \emptyset$
Output: $W, \vec{w}$ such that $W \in \mathbb{F}_2^{n\times\delta}$ for $\delta \in \{d'-1, d\}$ is full column rank and $\{W\vec{z}+\vec{w}\} = \{U\vec{x}+\vec{u}\} \cap \{V\vec{y}+\vec{v}\}$
1: **procedure** AffineIntersect$(U, \vec{u}, V, \vec{v})$
2: $\begin{pmatrix}\vec{x}_0\\ \vec{y}_0\end{pmatrix} \leftarrow$ a solution to $[U \mid V]\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix} = \vec{u}+\vec{v}$
3: $\begin{pmatrix}\vec{x}_1\\ \vec{y}_1\end{pmatrix}, \dots, \begin{pmatrix}\vec{x}_\delta\\ \vec{y}_\delta\end{pmatrix} \leftarrow$ a basis for $\ker([U \mid V])$
4: $X \leftarrow$ a $D \times \delta$ matrix whose δ columns are $x_1, x_2, \dots, x_\delta$
5: **return** $UX, U\vec{x}_0 + \vec{u}$

As the intersection is not empty, there exists some $\vec{x}', \vec{y}'$ such that $U\vec{x}' + \vec{u} = V\vec{y}' + \vec{v}$. Rearranging yields that $U\vec{x}' + V\vec{y}' = \vec{u} + \vec{v}$. We rewrite this using block matrices to get $[U \mid V]\begin{pmatrix}\vec{x}'\\ \vec{y}'\end{pmatrix} = \vec{u}+\vec{v}$. Since there is a solution to this system, we can perform Gaussian elimination to find one. Furthermore, Gaussian elimination can also be used in step 3 to find a basis for $\ker([U \mid V])$.

We show that δ is either d' or $d'-1$. Let δ be $\dim(\ker([U \mid V]))$ and observe that $\dim(\ker([U|V]) = \dim(\text{im}(U) \cap \text{im}(V)) = \dim(U) + \dim(V) - \dim(\text{im}(U) \cup \text{im}(V))$. As $\dim(U) = d'$, $\dim(V) = n-1$ and $\dim(\text{im}(V)) \le \dim(\text{im}(U) \cup \text{im}(V)) \le \dim(F_2^n) = n$, we conclude that δ is either d or $d-1$.

We verify that $W = UX$ and $\vec{w} = Ux_0 + \vec{u}$ define the intersection of the two affine spaces. Fix $\hat{z} \in \mathbb{F}_2^\delta$ and consider $W\vec{z} + \vec{w}$. Unwrapping definitions, we have $W\vec{z} + \vec{w} = UX\vec{z} + U\vec{x}_0 + \vec{u} = U(X\vec{z} + \vec{x}_0) + \vec{u}$ and thus by definition it is in $\{U\vec{x}+\vec{u}\}$. To show $W\vec{z} + \vec{w} \in \{V\vec{y}+\vec{v}\}$, we must first rewrite X.

Let K be the matrix whose columns are the computed basis for $\ker([A|B])$. Then $X = [\mathbf{I}_\delta|\mathbf{0}_{n-1}]K$ where $\mathbf{I}_\delta$ is the $\delta\times\delta$ identity matrix and $\mathbf{0}_{n-1}$ is the $(n-1)\times(n-1)$ zero matrix. We also rewrite $[U \mid V] = U[\mathbf{I}_\delta \mid \mathbf{0}_{n-1}] + V[\mathbf{0}_\delta \mid \mathbf{I}_{n-1}]$. As $[U \mid V]Kv = \vec{0}$ for any v, we have $U[\mathbf{I}_\delta \mid \mathbf{0}_{n-1}]K\vec{z} = V[\mathbf{0}_\delta \mid \mathbf{I}_{n-1}]K\vec{z}$. Furthermore, $U\vec{x}_0 + \vec{u} = V\vec{y}_0 + \vec{v}$ since $\begin{pmatrix}\vec{x}_0\\ \vec{y}_0\end{pmatrix}$ as a solution. Hence,

$$\begin{aligned}
W\vec{z} + \vec{w} &= UX\vec{z} + U\vec{x}_0 + \vec{u}\\
&= U[\mathbf{I}_\delta \mid \mathbf{0}_{n-1}]K\vec{z} + U\vec{x}_0 + \vec{u}\\
&= V[\mathbf{0}_\delta \mid \mathbf{I}_{n-1}]K\vec{z} + V\vec{y}_0 + \vec{v}\\
&= V([\mathbf{0}_\delta \mid \mathbf{I}_{n-1}]K\vec{z} + \vec{y}_0) + \vec{v}
\end{aligned}$$

We now show that every element in the intersection is also in $\{W\vec{z}+\vec{w}\}$. Let $\vec{u}$ be an element of the intersection: $u = U\vec{x} + \vec{u} = V\vec{y} + \vec{v}$ for some $\vec{x}, \vec{y}$. Rearranging yields $U\vec{x} + V\vec{y} = \vec{u} + \vec{y}$ and therefore $[U|V]\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix} = \vec{u}+\vec{v}$. As $\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix}$ is a solution to the system, we can write that $\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix} = \begin{pmatrix}\vec{x}'\\ \vec{y}'\end{pmatrix} + \begin{pmatrix}\vec{x}_0\\ \vec{y}_0\end{pmatrix}$ for some $\begin{pmatrix}\vec{x}'\\ \vec{y}'\end{pmatrix} \in \ker([U|V])$. Therefore $\begin{pmatrix}\vec{x}'\\ \vec{y}'\end{pmatrix} = K\vec{z}'$

for some $\vec{z}' \in \mathbb{F}_2^\delta$. Therefore:

$$u = U\vec{x} + \vec{u} = U[\mathbf{I}_\delta \mid \mathbf{0}_{n-1}]\left(K\vec{z}' + \begin{pmatrix}\vec{w}_0\\ \vec{z}_0\end{pmatrix}\right) + \vec{u} = UX\vec{z'} + U\vec{w}_0 + \vec{u} = W\vec{z}' + \vec{w}.$$

712 Therefore the algorithm correctly outputs the direction matrix and offset for the intersec-
713 tion of the given affine subspaces. As Gaussian elimination is efficient, the overall algorithm
714 runs in polynomial time with respect to n as desired. ◀

715 ConstraintToAffine

Algorithm 7 Computes the affine space corresponding to a parity constraint

Input: $I \subseteq [n], c \in \{0,1\}$ such that $I \neq \emptyset$
Output: $B, \vec{v}$ such $B \in \mathbb{F}_2^{n\times(n-1)}, b \in \mathbb{F}_2^n$ and $\{B\vec{z} + b\} = \{\vec{x} \in F_2^n \mid \bigoplus_{i\in I} \vec{x}_i = c\}$
1: **procedure** ConstraintToAffine(I, c)
2: $\vec{z}_0 \leftarrow$ a solution to $\mathbb{1}_I\vec{z} = c$
3: $\vec{z}^1, \vec{z}^2, \ldots z^{n-1} \leftarrow$ a basis for $\ker(\mathbb{1}_I)$
4: $Z \leftarrow$ the $n \times (n-1)$ matrix whose columns are $\vec{z}^1, \vec{z}^2, \ldots z^{n-1}$
5: **return** $Z, \vec{z}_0$

716 **Proof.** The algorithm is well-defined. As $I \neq 0$, there is a solution to $\mathbb{1}_I(\vec{z}) = c$. The
717 algorithm can solve the system and find a solution and find a basis for the kernel. As the
718 rank of $\mathbb{1}_I$ is 1, the basis has dimension $n-1$ by the rank-nullity theorem. Correctness
719 follows immediately: $\bigoplus_{i\in I} x_i = c$ is a linear equation over $\mathbb{F}_2$ and its solution set is exactly
720 the affine space $\vec{z}_0 + \ker(\mathbb{1}_I)$.
721 As Gaussian elimination is efficient and can be used to implement sets 2 and 3, the
722 algorithm runs in polynomial time with respect to n. ◀

723 FindSubstitution

Algorithm 8 Computes a substitution which makes $\mathcal{C}$ output constant b

Input: $\mathcal{C}$ a normalized non-constant circuit of fan-out 1 $\oplus$ gates, $b \in \{0,1\}$
Output: $\mathcal{S}, j$ such that $\mathcal{S} \equiv \bigoplus_{i\in I} x_i \oplus c_0$ for some I where substituting x_j in C for $\mathcal{S}$ makes C output constant b
1: **procedure** FindSubstitution($\mathcal{C}, b$)
2: $(I, c) \leftarrow$ ParitySupport($\mathcal{C}$)
3: $j \leftarrow$ any element of I
4: $\mathcal{S} \leftarrow \bigoplus_{i\in I\setminus\{j\}} x_i \oplus (c \oplus b)$ as a circuit
5: **return** $\mathcal{S}, j$

724 **Proof.** The algorithm is correct. As $\mathcal{C}$ is non-constant and paritysupport is correct, $\mathcal{C}$
725 computes $\bigoplus_{i\in I} x_i \oplus c$ and $I \neq \emptyset$. Substituting $x_j \leftarrow S$ in $\mathcal{C}$ means that $\mathcal{C}$ computes
726 $\bigoplus_{i\in I\setminus\{I\}} x_i \oplus \left(\bigoplus_{i\in I\setminus\{I\}} x_i \oplus (c \oplus b)\right) \oplus c = b$ as desired. As paritysupport runs in
727 polynomial time, and $|S| = O(|\mathcal{C}|)$, the algorithm runs in polynomial time. ◀