Verifying Quantum Error Correction Codes with SAT Solvers

Pengyu Liu ✉
Carnegie Mellon University, USA

Mengdi Wu ✉
Carnegie Mellon University, USA

Abstract

Quantum error correction is essential for executing quantum algorithms under realistic noise. However, verifying the correctness of quantum error correction code implementations remains challenging due to the exponential size of the possible error patterns. In this paper, we present a SAT-based approach to formally verify quantum error correction codes by encoding the verification problem as a SAT problem. We apply our method to analyze surface code implementations and successfully identify bugs in a recently published paper, where codes claimed to correct k errors actually fail to do so for larger distances. Our approach demonstrates that SAT solvers can efficiently find counterexamples (bugs) in quantum error correction implementations, though verifying correctness (proving no bugs exist) remains computationally challenging due to the inherent difficulty of UNSAT problems combined with XOR constraints.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases SAT solver, quantum error correction, surface code, formal verification

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Quantum computing promises to solve certain problems exponentially faster than classical computers, with potential applications ranging from quantum chemistry [2], cryptography [20], machine learning [23], to finance [19]. However, quantum systems are inherently fragile: quantum bits (qubits) are susceptible to errors from decoherence, environmental noise, and imperfect gate operations [14]. Unlike classical systems where errors mainly occur during data transmission or storage, quantum errors occur *continuously during computation itself*, which intertwines quantum algorithms with quantum error correction, making the correctness of a code not only a static property but also a dynamic one [8].

Quantum error correction (QEC) codes address this challenge by spreading logical information across multiple physical qubits, ensuring that local errors cannot easily affect the logical information. The *distance* d of a code determines its error-correcting capability: a distance-d code can correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors. Verifying that a code implementation achieves its claimed distance is crucial for ensuring fault tolerance, but exhaustively testing all possible error combinations is computationally infeasible for practical code sizes.

There is prior work using SMT solvers to verify the correctness of quantum error correction codes, but the performance is not satisfactory. For example, it takes 70 hours to verify a distance-7 code [6].

1.1 Contributions

In this paper, we make the following contributions:

1. We formulate quantum error correction verification as a SAT problem, enabling the use of highly optimized SAT solvers.

2. We develop efficient encodings for XOR constraints arising from detector definitions using Tseitin transformation with both chain and tree structures.
3. We apply our method to verify surface code implementations and discover bugs in a recently published Nature paper [5], where codes claimed to achieve certain distances actually fail.
4. We analyze the performance characteristics of our approach, identifying the computational challenges that make verification (UNSAT problems) significantly harder than bug finding (SAT problems).

2 Background

2.1 Quantum Computing Basics

A *qubit* (quantum bit) is the fundamental unit of quantum information. Unlike a classical bit that exists in state 0 or 1, a qubit can exist in a *superposition* $\alpha|0\rangle + \beta|1\rangle$, where α and β are complex amplitudes satisfying $|\alpha|^2 + |\beta|^2 = 1$ [18]. When measured, the qubit collapses to $|0\rangle$ with probability $|\alpha|^2$ or $|1\rangle$ with probability $|\beta|^2$.

For n qubits, the system state lives in a 2^n-dimensional Hilbert space spanned by computational basis states $|x_1 x_2 \cdots x_n\rangle$ where each $x_i \in \{0, 1\}$. A general n-qubit state is $|\psi\rangle = \sum_{x\in\{0,1\}^n} \alpha_x |x\rangle$ with $\sum_x |\alpha_x|^2 = 1$. This exponential growth in state space makes quantum systems both powerful and fragile.

2.2 The Surface Code

The surface code [11, 9] is one of the most promising quantum error correction codes due to: (1) local nearest-neighbor interactions compatible with many hardware platforms, (2) high error threshold ($\sim 1\%$, below which error correction becomes beneficial), and (3) efficient decoding via near-linear-time algorithms [16] that are also near optimal. The surface code has been successfully demonstrated on multiple experimental platforms, including superconducting qubits [1] and neutral atoms [5].

2.3 Stabilizer Formalism

The stabilizer formalism [13] enables error detection without measuring the encoded quantum state directly. A stabilizer code is defined by a set of commuting n-qubit Pauli operators $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$. Valid codewords $|\psi\rangle$ satisfy $S_i|\psi\rangle = |\psi\rangle$ for all $S_i \in \mathcal{S}$.

When an error E (a Pauli operator) occurs, the corrupted state $E|\psi\rangle$ may no longer be a $+1$ eigenstate of all stabilizers. The syndrome is determined by the commutation relations: measuring S_i yields $+1$ if $ES_i = S_iE$ (commute), and -1 if $ES_i = -S_iE$ (anti-commute). Mathematically, if we define syndrome bit $s_i \in \{0, 1\}$ where $S_iE = (-1)^{s_i} ES_i$, then the syndrome vector $\mathbf{s} = (s_1, \ldots, s_m)$ can be used to determine the error.

A quantum code with parameters $[[n, k, d]]$ uses n physical qubits to encode k logical qubits with distance d, meaning any error affecting fewer than d qubits produces a non-trivial syndrome and can be detected.

2.4 Detectors and Decoders

A *detector* is a linear combination of measurement outcomes that is deterministic in the absence of errors. When errors occur, detectors may produce unexpected values, providing classical information about which errors likely occurred.

The *decoder* is a classical algorithm that uses detector information to infer the error pattern and apply corrections.

2.5 Detector Error Model (DEM)

We use Stim's Detector Error Model (DEM) format [12] to represent error mechanisms and their effects. A DEM file describes each error mechanism with its probability, affected detectors (D#), and affected logical observables (L#). For example:

```
error(0.027) D0 D1
error(0.101) D0 L0
```

The first line triggers detectors D0 and D1 with probability 0.027; the second line triggers D0 and flips logical observable L0 with probability 0.101. In this work, we only focus on the number of errors that occur and ignore the probability values.

2.6 Zero-Detector Verification

Most quantum error correction codes are *linear codes*: if error patterns E_1 and E_2 each produce syndromes $\mathbf{s}_1$ and $\mathbf{s}_2$, then $E_1 \oplus E_2$ produces syndrome $\mathbf{s}_1 \oplus \mathbf{s}_2$. This linearity has an important consequence for code verification.

A *zero-detector logical error* is an error pattern that triggers no detectors (zero syndrome) but flips at least one logical observable. Such errors are undetectable and cause logical failures. Due to linearity, if E_1 and E_2 produce the same syndrome $\mathbf{s}_1 = \mathbf{s}_2$ but different logical outcomes, then $E_1 \oplus E_2$ triggers no detectors yet flips a logical observable—a zero-detector logical error.

The *code distance* d is defined as the minimum weight of any zero-detector logical error:

$$d = \min\{|E| : E \text{ triggers no detectors and flips a logical observable}\}$$

A code with distance d can reliably correct up to $t = \lfloor (d-1)/2 \rfloor$ errors. This is because any two correctable error patterns E_1 and E_2 with $|E_1|, |E_2| \leq t$ must have distinct syndromes; otherwise $E_1 \oplus E_2$ would be a zero-detector logical error with weight at most $2t < d$, contradicting the definition of distance. This guarantee assumes an *optimal decoder* that, given a syndrome, selects the minimum-weight error pattern consistent with that syndrome, thereby ensuring correct decoding for all errors up to weight t.

3 SAT Encoding Methodology

Given n error mechanisms, m detectors, and ℓ logical observables, we create a boolean variable e_i for each error mechanism and add the following constraints:

1. *Detector*: $\bigoplus_{i \in \text{affects}(D_j)} e_i = 0$ for each detector D_j;
2. *Observable*: $\bigvee_k (\bigoplus_{i \in \text{affects}(L_k)} e_i = 1)$ for each logical observable L_k;
3. *Cardinality*: $\sum_i e_i \leq k$.

If the SAT solver finds a solution, the solution represents an undetectable logical error with at most k errors, demonstrating that the distance of the code is at most k. Conversely, if the solver proves UNSAT, then no such error pattern exists, certifying that the code distance is at least $k+1$.

3.1 XOR Encoding with Tseitin Transformation

XOR constraints must be converted to CNF using the Tseitin transformation. For a base-b XOR gate $c = e_1 \oplus e_2 \oplus \cdots \oplus e_b$, we enumerate all 2^b input combinations and generate 2^{b-1} clauses enforcing $c = 1$ when an odd number of inputs are true. For the simplest case $b = 2$, the constraint $c = a \oplus b$ requires 4 clauses: $(\neg a \vee \neg b \vee \neg c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee c)$.

To encode $e_1 \oplus \cdots \oplus e_n = 0$, we recursively decompose it using base-b XOR gates as building blocks:

Chain Structure: Introduce auxiliary variables sequentially: $a_1 = e_1 \oplus \cdots \oplus e_b$, $a_2 = a_1 \oplus e_{b+1} \oplus \cdots \oplus e_{2b-1}$, etc. This produces a linear chain with depth $O(n/b)$.

Tree Structure: Reduce XORs in a balanced tree: first compute $a_i = e_{(i-1)b+1} \oplus \cdots \oplus e_{ib}$ for each group of b variables, then recursively combine a_i's using the same method. This achieves depth $O(\log_b n)$ for better unit propagation.

Higher base values reduce the number of auxiliary variables but increase clause complexity exponentially (2^{b-1} clauses per gate).

3.2 Cardinality Constraints

To encode "at most k of n variables are true," the naive approach adds a clause for each $(k+1)$-subset, yielding $\binom{n}{k+1}$ clauses—exponential in k.

We use the *totalizer encoding* [3], which constructs a unary counting circuit via a binary tree. Each leaf represents an input variable e_i. Each internal node merges two sorted unary counters from its children: if the left child outputs $(l_1, \ldots, l_a)$ and the right outputs $(r_1, \ldots, r_b)$, the merged output $(o_1, \ldots, o_{a+b})$ satisfies $o_i = 1$ iff at least i inputs below are true. The merge operation uses clauses of the form $l_i \wedge r_j \Rightarrow o_{i+j}$. At the root, it is enforced that $o_{k+1} = 0$ to guarantee at most k variables are true. This encoding requires $O(n \log n)$ auxiliary variables and $O(nk)$ clauses, and provides strong unit propagation.

4 Evaluation

Our implementation uses Python with PySAT and CaDiCaL [4], a state-of-the-art CDCL solver. We use Stim [12] to generate detector error models from quantum circuits.

4.1 Bug Discovery in Nature Paper

We applied our method to surface code implementations from a Nature paper [5], where the authors claimed to have implemented a variant of the surface code that can correct $\frac{d-3}{2}$ errors for distance d.

Table 1 shows the problem scales for different code distances in the buggy version of the surface code.

Table 2 shows our findings: **distances 11 and 13 fail to correct the claimed number of errors**. The distance-11 code corrects only 3 errors (not 4), and the distance-13 code corrects only 4 (not 5).

Our SAT solver not only proves the existence of error patterns that trigger no detectors while flipping a logical observable but also provides explicit counterexamples. For the distance-11 code, an 8-error pattern (versus the expected minimum of 11) demonstrates a "shortcut" through the code. These counterexamples provide valuable debugging information, pinpointing exactly which error mechanisms combine to defeat error correction.

The root cause of this bug appears to be incorrect extrapolation from smaller code distances. While the observed sequence (0, 1, 2, 3) for distances 3, 5, 7, 9 naturally suggests

Table 1 Problem sizes for different code distances

Distance	Errors	Detectors	CNF Vars
3	1	16	438
5	3	72	3,392
7	4	192	11,824
9	6	400	29,058
11	7	720	58,704
13	9	1,176	104,856

Table 2 Verification Results: Claimed vs Actual Correctable Errors

Distance	Actual	Claimed
3	0	0
5	1	1
7	2	2
9	3	3
11	**3**	4
13	**4**	5

the pattern continues with 4 for distance 11, our formal verification reveals this intuition is incorrect. This highlights the importance of formal verification in quantum error correction: properties that hold for small instances do not necessarily generalize to larger systems.

4.2 Performance Analysis: SAT vs UNSAT

Figure 1 compares SAT (bug finding) vs UNSAT (verification) performance. Finding counterexamples is fast, but the time required for proving correctness grows rapidly with problem size.

We notice that for small instances, SAT is sometimes slower than UNSAT. This might be because when the code is small, the relative difference between d and $d-1$ is large, making UNSAT easier to prove. In an extreme case, when $d=3$, we are trying to prove that 0 errors can trigger a logical observable, which is obviously impossible.

4.3 Performance Analysis: Different XOR Encoding Strategies

Figure 2 compares XOR encoding strategies (chain vs tree, base-2 vs base-3). Tree-based encodings provide better propagation and thus are faster in both SAT and UNSAT problems. However, we do not see a significant difference in performance between base-2 and base-3 encodings.

4.4 Performance Analysis: Different Solvers

We also tested the performance of solving the SAT and UNSAT problems using other solvers, including CryptoMiniSat [21], MaxSAT(RC2) [17], and Z3 [7].

We chose these solvers because CryptoMiniSat has native support for XOR constraints, MaxSAT(RC2) is optimized for cardinality constraints, and Z3 has the potential to leverage SMT reasoning for better performance.

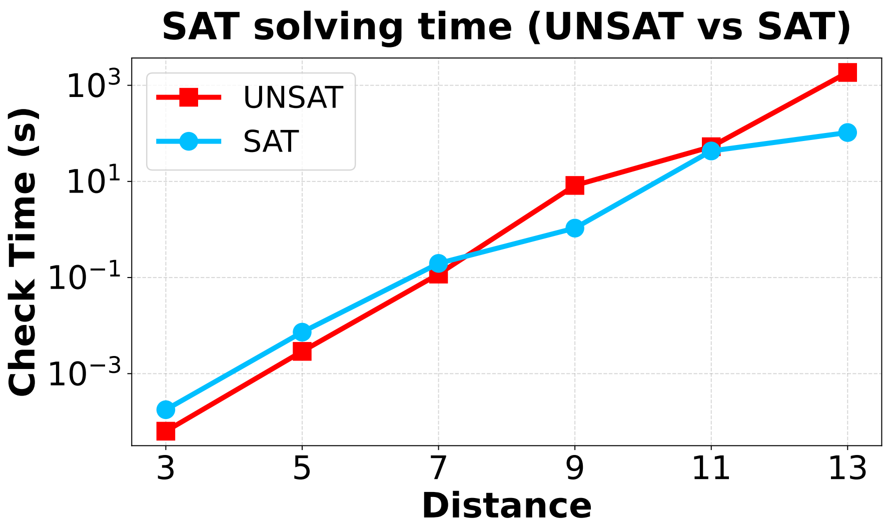

Figure 1 Solving time for the buggy surface code. The UNSAT problem is much harder to solve than the SAT problem when the code distance is large.

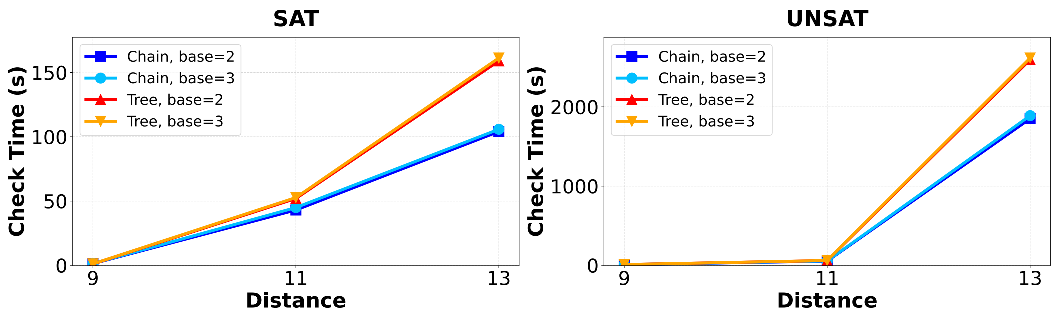

Figure 2 Comparison of XOR encoding strategies

Throughout this subsection, we use the “correct” implementation of the surface code and base-2 chain XOR encoding when applicable.

4.4.1 Results using CaDiCaL

Table 3 shows the results using CaDiCaL. CaDiCaL is able to solve the UNSAT problem up to distance 9 but fails to solve the UNSAT problem at distance 11. This is already the best performance among all solvers. The build time shows how long it takes to build the model, which only grows polynomially with the number of variables.

Table 3 Results using CaDiCaL

Distance	Errors	Result	Num vars	Build Time	Solve Time
3	2	UNSAT	679	0.002	0.0001
3	3	SAT	679	0.002	0.0002
5	4	UNSAT	4479	0.063	0.024
5	5	SAT	4479	0.066	0.011
7	6	UNSAT	14461	0.218	1.623
7	7	SAT	14461	0.213	0.119
9	8	UNSAT	34160	0.986	72.82
9	9	SAT	34160	0.991	1.067
11	10	UNSAT	67004	4.748	>3600
11	11	SAT	67004	4.748	0.007

4.4.2 Results using CryptoMiniSat

Table 4 shows the results using CryptoMiniSat [21]. We use the native XOR encoding support of CryptoMiniSat instead of using the Tseitin transformation. CryptoMiniSat is able to solve the UNSAT problem up to distance 7 but fails to solve the UNSAT problem at distance 9. We attribute this to the fact that CryptoMiniSat is not optimized for cardinality constraints.

Table 4 Results using CryptoMiniSat

Distance	Errors	Result	Num. vars	Build Time	Solve Time
3	2	UNSAT	539	0.016	0.150
3	3	SAT	539	0.012	0.0003
5	4	UNSAT	3691	0.063	272.9
5	5	SAT	3691	0.066	0.0025
7	6	UNSAT	12175	0.529	91.68
7	7	SAT	12175	0.467	0.019
9	8	UNSAT	29197	2.374	>3600
9	9	SAT	29197	2.554	0.059

4.4.3 Results using Z3

Table 5 shows the results using Z3 [7]. We use the boolean theory, linear integer arithmetic and modular arithmetic theories. Z3 is the worst among all solvers and can only solve the UNSAT problem up to distance 5.

Table 5 Results using Z3

Distance	Errors	Result	Build Time	Solve Time
3	2	UNSAT	0.016	0.150
3	3	SAT	0.012	0.066
5	4	UNSAT	0.063	272.9
5	5	SAT	0.066	0.697
7	6	UNSAT	0.187	>3600
7	7	SAT	0.006	0.007

4.4.4 Results using MaxSAT(RC2)

Table 6 shows the results using MaxSAT(RC2) [17]. In MaxSAT, we use the same XOR encoding, but instead of using cardinality constraints, we ask the solver to find the minimum number of errors that can trigger the logical observable. RC2 is comparable to CryptoMiniSat but has the advantage of not requiring any prior knowledge and can find the exact error-correcting capability in one shot.

Table 6 Results using MaxSAT(RC2)

Distance	Result	Num. hard clauses	Num. soft clauses	Build Time	Solve Time
3	3	506	74	0.001	0.002
5	5	2758	382	0.002	0.321
7	7	8034	1094	0.004	96.3
9	9	17606	2378	0.010	>3600

5 Challenges and Limitations

Through our experiments, we found that verification is significantly harder than bug finding. We believe the following factors explain this:

UNSAT Problem Difficulty: Verification requires proving UNSAT—that no satisfying assignment exists. This is inherently harder than finding satisfying assignments because the solver must exhaustively rule out all possibilities. While SAT problems can often be solved quickly by finding a single witness, UNSAT proofs require exploring (and pruning) the entire search space.

XOR Constraints: SAT solvers are known to struggle with parity constraints [22]. Our XOR constraints, while encoded into CNF via Tseitin transformation, retain their underlying parity structure that causes difficulty for resolution-based proof systems. The inability of resolution to efficiently handle XOR is a fundamental limitation.

Cardinality Constraints: The "at most k errors" constraint resembles the pigeonhole principle, which is known to require exponentially long resolution proofs [15]. Combined with XOR constraints, this creates a particularly challenging problem structure.

Together, these factors make verification substantially harder than bug finding, explaining the dramatic performance gap observed in our experiments.

5.1 A Proof Complexity Perspective

It is well known that resolution is intractable for the pigeonhole principle: any resolution refutation of PHP^n_{n+1} has size $2^{\Omega(n)}$ [15]. This classical lower bound serves as a canonical benchmark for reasoning about counting constraints in CNF.

A closely related structure arises in our setting. Consider variables $x_1, \ldots, x_{n+1}$ and the contradictory formula

$$\bigwedge_{i=1}^{n+1} x_i \wedge \sum_{i=1}^{n+1} x_i \leq k.$$

This formula is an instance of the surface code problem when all detectors only detect two errors, and captures a simple form of global counting inconsistency. The proof complexity of the resulting CNF depends on the particular encoding of the cardinality constraint into clauses.

Under the *pigeonhole encoding* of cardinality constraints [**?**], the formula above contains, as a projection, an instance of the pigeonhole principle. Consequently, every resolution refutation of this encoding has exponential size, by an immediate reduction to the lower bound of [15].

The situation for other standard encodings (such as totalizer, sorting-network, or binary-adder encodings) is less well understood. Existing lower bounds do not directly apply, and it remains open whether these alternative encodings also admit exponential resolution lower bounds or whether some of them may yield polynomial-size refutations. Establishing the precise proof complexity of these cardinality encodings is an interesting direction for further investigation.

6 Related Work

There has been prior work on verifying quantum error correction codes using SMT solvers, for example, [10]. However, their proof relies on a specific decoder and cannot generalize to other codes. A more general approach is proposed in [6] using SMT solvers. However, their scalability is not satisfactory: it takes 70 hours to verify a distance-7 code.

7 Conclusion and Future Work

We presented a SAT-based approach to verifying quantum error correction codes, encoding the verification problem as boolean satisfiability with XOR constraints for detectors, cardinality constraints for error bounds, and disjunctive constraints for logical observables. Our method discovered bugs in a published Nature paper's surface code implementation, where distance-11 and distance-13 codes fail to achieve their claimed error correction capability.

Our experiments reveal a fundamental challenge: SAT solvers efficiently find counter-examples in faulty implementations, but proving correctness (UNSAT) is significantly harder due to the combination of XOR constraints, cardinality constraints, and the need to exhaustively rule out all possibilities.

For future work, we propose a hybrid SAT and theorem prover approach. SAT solvers excel at bug finding and search space pruning, while theorem provers (e.g., Lean) provide formal correctness guarantees. A hybrid approach could use SAT for rapid counterexample detection and pruning and then employ theorem provers to formally verify correctness.

References

1 Quantum error correction below the surface code threshold. *Nature*, 638(8052):920–926, 2025.

2 Ryan Babbush, Jarrod McClean, Dave Wecker, Alán Aspuru-Guzik, and Nathan Wiebe. Chemical basis of trotter-suzuki errors in quantum chemistry simulation. *Physical Review A*, 91(2):022311, 2015.

3 Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *International conference on principles and practice of constraint programming*, pages 108–122. Springer, 2003.

4 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

5 Dolev Bluvstein, Alexandra A Geim, Sophie H Li, Simon J Evered, J Pablo Bonilla Ataides, Gefen Baranes, Andi Gu, Tom Manovitz, Muqing Xu, Marcin Kalinowski, et al. A fault-tolerant neutral-atom architecture for universal quantum computation. *Nature*, pages 1–3, 2025.

6 Kean Chen, Yuhao Liu, Wang Fang, Jennifer Paykin, Xin-Chuan Wu, Albert Schmitz, Steve Zdancewic, and Gushu Li. Verifying fault-tolerance of quantum error correction codes. In *International Conference on Computer Aided Verification*, pages 3–27. Springer, 2025.

7 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

8 Nicolas Delfosse and Adam Paetznick. Spacetime codes of clifford circuits. *arXiv preprint arXiv:2304.05943*, 2023.

9 Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. Topological quantum memory. *Journal of Mathematical Physics*, 43(9):4452–4505, 2002.

10 Wang Fang and Mingsheng Ying. Symbolic execution for quantum error correction programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1040–1065, 2024.

11 Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A—Atomic, Molecular, and Optical Physics*, 86(3):032324, 2012.

12 Craig Gidney. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, 2021.

13 Daniel Gottesman. *Stabilizer codes and quantum error correction.* California Institute of Technology, 1997.

14 Daniel Gottesman. Surviving as a quantum computer in a classical world. *Textbook manuscript preprint*, 8(8.1):8–2, 2024.

15 Armin Haken. The intractability of resolution. *Theoretical computer science*, 39:297–308, 1985.

16 Oscar Higgott. Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching. *ACM Transactions on Quantum Computing*, 3(3):1–16, 2022.

17 Alexey Ignatiev, António Morgado, and Joao Marques-Silva. Rc2: an efficient maxsat solver. *Journal on Satisfiability, Boolean Modelling and Computation*, 11(1):53–64, 2019.

18 Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information.* Cambridge university press, 2010.

19 Román Orús, Samuel Mugel, and Enrique Lizaso. Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 4:100028, 2019.

20 Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

21 Mate Soos. The cryptominisat 5 set of solvers at sat competition 2016. *Proceedings of SAT Competition*, 28, 2016.

22 Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM (JACM)*, 34(1):209–219, 1987.

23 Xin-Ding Zhang, Xiao-Ming Zhang, and Zheng-Yuan Xue. Quantum hyperparallel algorithm for matrix multiplication. *Scientific reports*, 6(1):24910, 2016.