Snakes and Ladders: Adapting the Surface Code to Defects

Catherine Leroux,[1,*] Sophia F. Lin,[1] Przemyslaw Bienias,[1] Krishanu R. Sankar,[1] Asmae Benhemou,[1,2] Aleksander Kubica,[1,3] and Joseph K. Iverson[1]

[1] *AWS Center for Quantum Computing, Pasadena, California 91125, USA*

[2] *Department of Physics and Astronomy, University College London, London WC1E 6BT, United Kingdom*

[3] *Yale Quantum Institute & Department of Applied Physics, Yale University, New Haven, Connecticut 06520, USA*

(Received 21 February 2025; revised 7 August 2025; accepted 28 August 2025; published 2 October 2025)

One of the critical challenges solid-state quantum processors face is the presence of fabrication imperfections and two-level systems, which render certain qubits and gates either inoperable or much noisier than tolerable by quantum error-correction protocols. To address this challenge, we develop a suite of novel and highly performant methods for adapting surface code patches in the presence of defective qubits and gates, which we call *Snakes and Ladders*. We explain how our algorithm generates and compares several strategies in order to find the optimal one for any given configuration of defective components, as well as introduce heuristics to improve runtime and minimize computing resources required by our algorithm. In addition to memory storage we also show how to apply our methods to lattice surgery protocols. Compared to prior works, our methods significantly improve the code distance of the adapted surface code patches for realistic defect rates, resulting in a logical performance similar to that of the defect-free patches.

DOI: 10.1103/815q-xjrb

I. INTRODUCTION

One of the leading architectures for error-corrected quantum computation is based on the surface code [1–4] and can be realized with qubits on a square grid and nearest-neighbor entangling gates. This approach is particularly attractive for superconducting qubits, which are typically arranged on a two-dimensional plane and strongly coupled to their nearest neighbors. However, hardware implementation of the surface code is susceptible to defective qubits or gates, which can arise due to mistargeted frequencies, fabrication defects [5,6], or two-level systems (TLSs) [7–9]. Effectively adapting the surface code to accommodate these imperfections has emerged as a critical challenge in the effort to scale up superconducting qubit devices.

Prior work [10–13] has focused on one particular strategy of handling defective qubits and gates, which we collectively refer to as data-qubit disabling (DQD). On top of defective data qubits, DQD also disables data qubits that are either involved in defective two-qubit gates or next to defective ancilla qubits. DQD converts the stabilizer checks around defective or disabled data qubits to gauge checks, which then comprise super-stabilizers. Recent refinements of DQD [14,15] optimized super-stabilizers to minimize reduction of the surface code distance for certain configurations of defects. Nonetheless, reduction of the distance remains significant and therefore appreciably diminishes the logical performance of the adapted surface code.

In this article, we introduce a new framework, called "Snakes and Ladders" (SnL), to adapt the surface code in the presence of defective qubits and gates. Compared to prior works, SnL greatly improves the code distance of the adapted surface code while ensuring that all stabilizers and super-stabilizers are measured over two error-correction rounds, the minimum number of rounds required to measure all stabilizers given that we are measuring checks that anti-commute [10]. Consequently, SnL reduces the qubit overhead of memory storage and fault-tolerant computation via lattice surgery with the surface code [16,17] in the presence of defective components.

SnL makes use of a novel primitive operation for defective ancilla qubits and two-qubit gates, called "ancilla repurposing." Ancilla repurposing uses two of the neighboring ancilla qubits to measure weight-2 gauge checks that, together, allow to recover the value of the missing stabilizer check. This primitive generally preserves the code distance, offering a significant gain over DQD (which

*Contact author: catilero@amazon.com

Published by the American Physical Society under the terms of the Creative Commons Attribution 4.0 International license. Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

reduces the distance by two for a defective ancilla qubit and by one for a defective two-qubit gate).

By ingeniously composing the ancilla-repurposing primitive and DQD, SnL constructs many global strategies for any configuration of defective components, resulting in the adapted surface code with the optimal code distance. SnL not only handles the defective components in the bulk; it also optimally deforms the boundary of the surface code. To reduce the computational overhead of constructing and analyzing global strategies, SnL may use heuristic methods to reduce the search space.

We call our algorithm "Snakes and Ladders" because repurposing ancillas (which helps us preserve code distance) is analogous to climbing ladders whereas disabling data qubits (which reduces the code distance) is akin to sliding down snakes in the eponymous game [18]. As explained in Sec. III B, several repurposed checks in a row resemble the rungs of a ladder, and the strings of disabled data qubits resemble snakes.

The article is organized as follows. First, we provide motivation in Sec. II and describe the key steps of SnL in Sec. III. Next, in Sec. IV we report numerical simulations for the surface code adapted to random configurations of defective components. In particular, we analyze memory storage, provide yield estimates and benchmark a logical CNOT gate implemented via lattice surgery. We conclude by discussing the impact of our work and some future directions in Sec. V.

II. MOTIVATION

Suppose that we have a processor consisting of a square lattice of data and ancilla qubits with nearest-neighbor gates. We wish to realize one or more patches of surface code within certain regions of the processor to enable quantum information processing, e.g., with lattice surgery. We restrict ourselves to the rotated surface code, which realizes the same code distance as the unrotated surface code with fewer physical qubits. We choose an orientation with X-type boundaries on top and bottom and Z-type boundaries on left and right. The vertical (horizontal) distance of the code is denoted $d_X\,(d_Z)$.

In what follows, we call a rectangular region of our processor in which we wish to realize a surface code patch a "target window." A target window is said to have dimensions $w \times h$ with width w and height h if it contains a grid of $w \cdot h$ data qubits and $(w+1)\cdot(h+1)$ ancilla qubits such that only ancilla qubits are on the perimeter of the window. Figure 1(a) shows a target window (yellow square). In the absence of defective components, this window supports a rotated surface code with distance up to $(d_X, d_Z) = (h, w)$. We use the terms "defect" or "defective component" to refer to a qubit or two-qubit gate in our surface code lattice that we cannot use in our code, either because it is nonoperational or because it is so much noisier than the other qubits or two-qubit gates that we prefer not to use it. We use "data defect," "ancilla defect," and "link defect" to refer to defective data qubits, ancilla qubits, and two-qubit gates, respectively.

Suppose that our processor has defective components and that we have characterized the defects, indicated in Fig. 1(a) with red marks on qubits and links, within the target window (yellow square). Then, we must determine how to construct a surface code patch within this target window. One approach would be to search for the surface code patch with the highest distance that is not compromised by defects within the target window. In general, defects will be distributed across the entire target window, and this approach will produce a small patch, like in Fig. 1(b). In this example, the resulting surface code patch has distance $(d_X, d_Z) = (3, 3)$ in a 7×7 target window. More generally, we would be unable to implement lattice surgery operations between this patch and patches in other windows unless we can find a defect-free routing space between them. To get larger code distance and to enable lattice surgery, we require a strategy for adapting our surface code patches to the presence of defects.

We place three constraints on our defect strategies. First, we insist on preserving the number of encoded qubits and the topology of the boundaries. Second, we require that the full set of stabilizer generators can be measured in two error-correction rounds, just like in prior approaches [10–15]. Finally, we do not use weight-1 gauge checks [14,15]. We find that incorporating weight-1 checks in SnL often leads to more noise and no improvement in code distance (see Appendix A for more detail). Subject to these constraints, we aim to maximize the code distance of the adapted surface code.

Two such strategies are shown in Figs. 1(c) and 1(d). Panel (c) is the strategy we find using DQD. This surface code patch has distance $(d_X, d_Z) = (5, 3)$, improving slightly over the code patch in panel (b). Panel (d) shows the optimal strategy we find with SnL. Here the patch has distance $(d_X, d_Z) = (6, 6)$. This patch is expected to result in smaller logical error rates than the patch generated with DQD since the performance is primarily limited by the minimum of the X- and Z-distance of the code. Equivalently, if we require a certain minimum logical performance for our patch, SnL allows us to work with a smaller target window than DQD. In turn, this enables more efficient layout of logical qubits in our processor, reducing the overhead of fault-tolerant quantum computation.

III. SNAKES AND LADDERS (SNL) FRAMEWORK

In this section, we detail the key steps of our framework for maximizing surface code distance in the presence of defects, which is fully implemented and available in our repository [19].

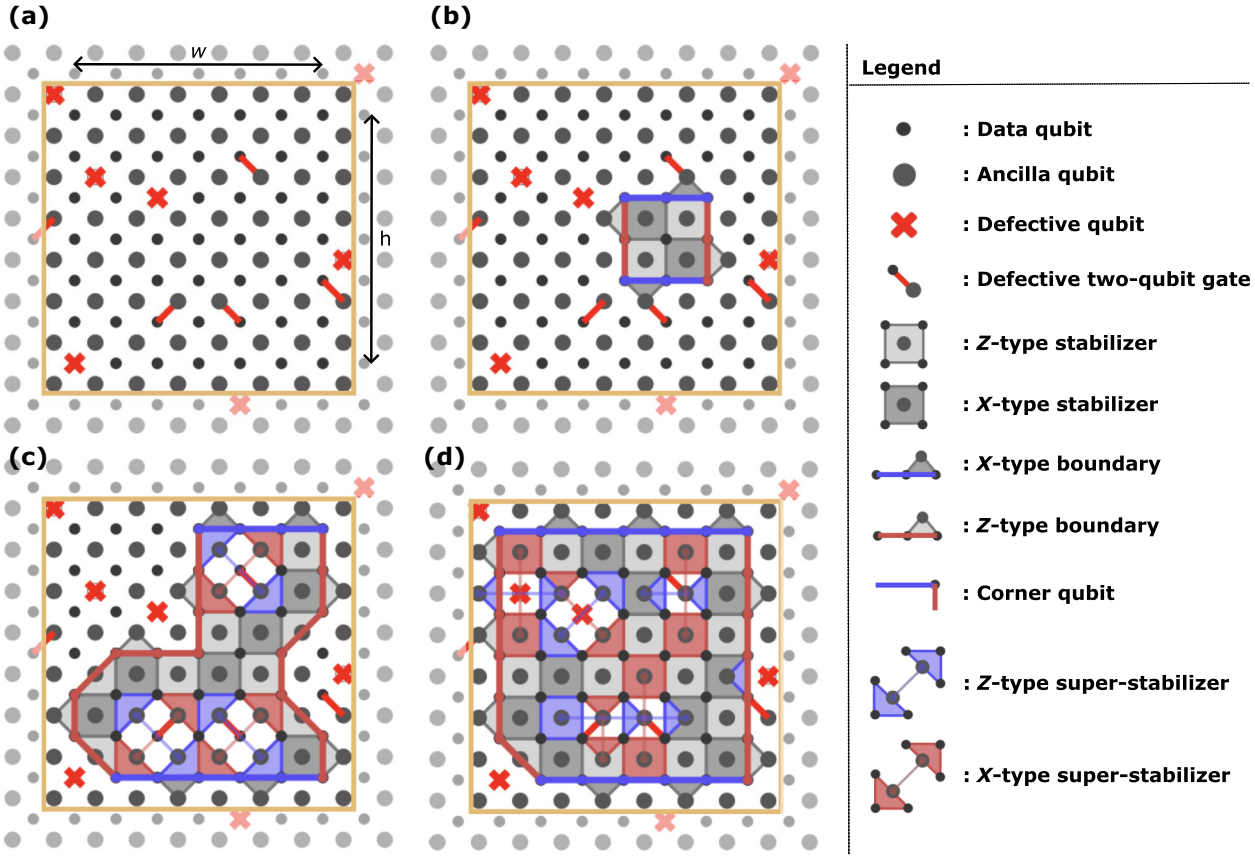


FIG. 1. (a) A processor consisting of a square lattice of qubits with nearest-neighbor gates. Several defective components marked in red. The question we study is how to realize the most performant surface code within a target window (yellow square) of dimensions $w \times h$. In the absence of defects, this window supports a surface code with distance up to $(d_X, d_Z) = (h, w)$. In this example $w = h = 7$. (b) A surface code patch with distance $(d_X, d_Z) = (3, 3)$ featuring no defective components. This is the surface code patch with the largest distance, free of defective components, that we can make in the current window. (c) A surface code patch adapted to defects using DQD. This code has distance $(d_X, d_Z) = (5, 3)$. (d) A surface code patch adapted to defects using SnL, resulting in distance $(d_X, d_Z) = (6, 6)$, which is closer to the maximum distance possible in the target window. In (b)–(d), gauge checks of the same color are combined together into super-stabilizers and are connected together with lines.

A. Strategies for isolated defects

For a given configuration of defects, our framework constructs many possible strategies to adapt the surface code, and selects the one with the best code distance. In this section, we present our strategies supposing we have only a single defect in our target window (see Fig. 2). For conciseness, we focus on defective components in the bulk, but the approach can be straightforwardly generalized to defects on the boundary (see Appendix B 1 for details). Our strategies are designed such that X-type and Z-type super-stabilizers are measured every other round. The unmodified stabilizer checks of the surface code are still measured every round.

1. Data defect

The primitive for isolated data defects is illustrated in Fig. 2(a). The prescription is to remove the defective data qubit from the four adjacent stabilizer checks, converting them to four weight-3 checks. After this change, neighboring checks of different Pauli type anti-commute, and therefore can no longer be measured simultaneously. This necessitates the use of an alternating schedule where the X-type checks and Z-types checks are measured in separate rounds. The measurement outcomes from individual gauge checks are random, but the product of the two X-type (Z-type) checks is deterministic and forms a single X-type (Z-type) stabilizer of weight 6. This approach "patches" the hole in the code and preserves the number of encoded logical qubits. However, the distance of the code is reduced by 1 in both directions because the lengths of X and Z logical strings passing through the hole in the code are both reduced by 1. Here the term "hole" denotes a set of inter-dependent super-stabilizers whose gauge checks commute with the checks outside the set but not with all the gauge checks within the set.

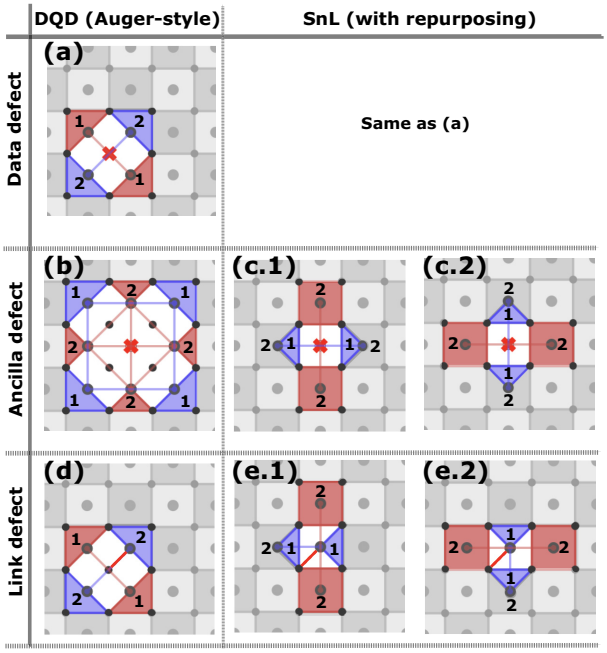


FIG. 2. Primitive operations used in our framework, for isolated defects. The highlighted checks are measured in alternating rounds, and the numbers indicate which checks are measured in odd and even rounds. The DQD strategies are shown in the left column for (a) a data defect, (b) an ancilla defect, (d) a link defect (mapped to a data defect). The SnL strategies for the same defects are shown in the right column.

2. Ancilla defect

Ancilla repurposing is a new primitive we introduce to handle isolated defective ancilla qubits. Two possible ways of using ancilla repurposing for an isolated ancilla defect are shown in Fig. 2(c). Here, the weight-4 stabilizer of the defective ancilla is split into two weight-2 gauge checks, and neighboring ancilla qubits on the left and right (top and bottom) are "repurposed" to measure these gauge checks in alternate rounds (in addition to their own stabilizer checks). The two weight-4 checks directly above and below (left and right) anti-commute with these weight-2 gauge checks and are thus themselves gauge checks which multiply to form one weight-8 stabilizer. Both left and right and top and bottom ways of repurposing around an ancilla defect are valid. However, depending on the Pauli type of the check the defective ancilla is supposed to measure, one orientation preserves the code distance, whereas the other reduces it by two along the weight-8 stabilizer (see Appendix B 2). We refer to one orientation as "distance-preserving" and the other as "lossy." In contrast, DQD disables all four data qubits in the neighborhood of the ancilla defect [see Fig. 2(b)], reducing d_X and d_Z by 2.

We find that ancilla repurposing benefits from what we call ancilla padding. Instead of using a grid with physical qubits at every other lattice position around the boundary (locations of the weight-2 checks in the defect-free surface code), we work with a target window that has a physical qubit at every position around the boundary as in Fig. 1. This allows us to repurpose these ancillas to recover the full code distance for ancilla defects near the boundary that we otherwise could not handle without loss of distance. See Appendix B 1 for an illustration.

3. Link defect

In the case of an isolated defective two-qubit gate, the ancilla qubit involved in the gate can still measure a weight-2 check that is part of its native stabilizer while one of the neighboring ancilla qubits can measure the other weight-2 check needed for recovering the original weight-4 stabilizer. This is illustrated in Fig. 2(e). As with ancilla defects, we have two orientations for the repurposing, one distance-preserving and one lossy. Similarly to ancilla defects we can recover the code distance, whereas DQD disables the data qubit in the defective gate [see Fig. 2(d)], leading to a distance loss of 1 in both directions.

B. Strategies for clustered defects

Naively, we would apply the distance-preserving repurposing for every ancilla and link defect in the patch as if they were isolated. However, this is not always possible when multiple defects are close together. Neighboring defects that cannot be considered separately using the strategies in Fig. 2 must be grouped into "defect clusters" and considered collectively. Within a defect cluster, repurposing a given ancilla might not be possible for two reasons. First, we do not allow an ancilla qubit to be repurposed more than once, since doing so would make it impossible to measure all stabilizers over two measurement rounds. Second, we do not allow repurposed weight-1 checks (see Appendix A for a detailed explanation), implying that any defective component (data, ancilla or gate) in a repurposed weight-2 check makes this check invalid. To solve this issue, we disable all data qubits involved in tentative repurposed checks that would be compromised by defects. This process results in SnL strategies that mix ancilla repurposing and the data defect primitive in Fig. 2(a). For example, if no ancilla repurposing is possible then SnL reduces to DQD.

Therefore, in an attempt to minimize the number of disabled data qubits, our framework ultimately considers both orientations of repurposing for each ancilla [Fig. 2(c)] and link defect [Fig. 2(e)]; in the presence of other defects, we might choose the lossy repurposing to optimize the overall code distance. We remark that if the ancilla in a link defect cannot be used, either because it is itself defective or due to another defective gate, we revert back to the same strategies as for an ancilla defect. Moreover, for every link defect, we also consider the possibility of disabling the data qubit instead of repurposing [Fig. 2(d)]. This can

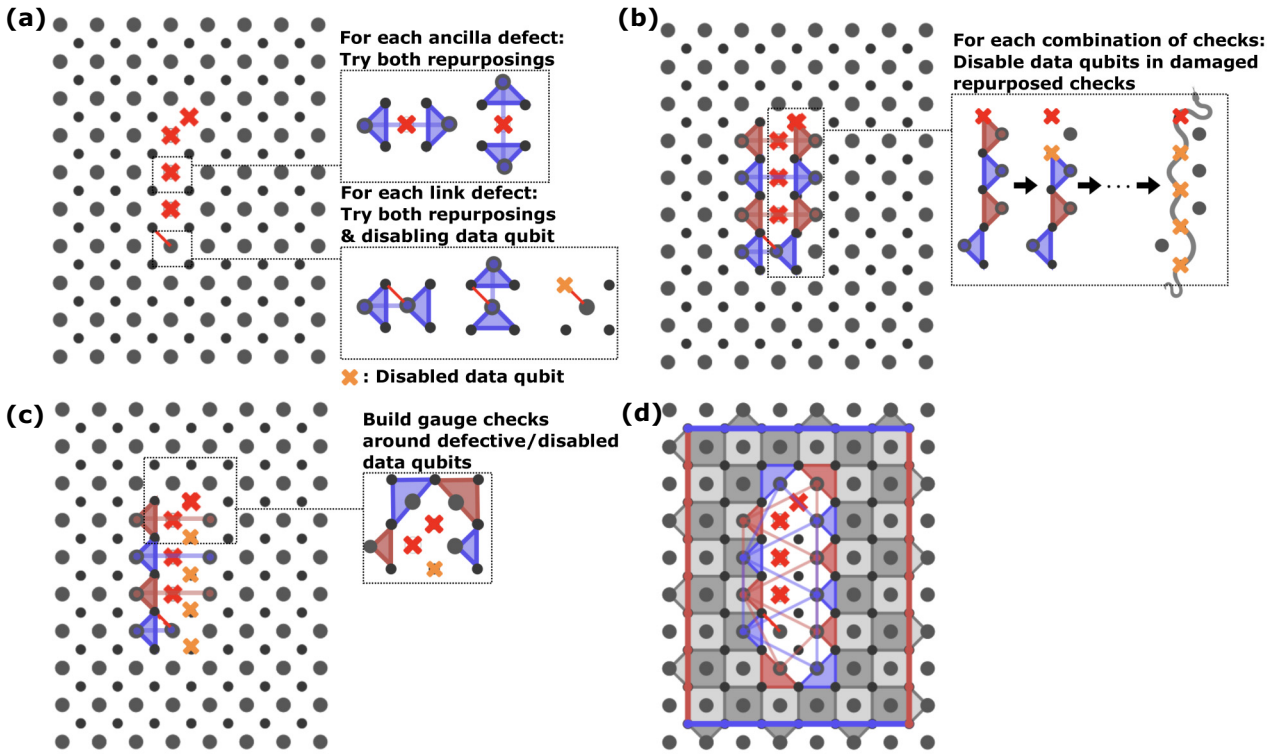


FIG. 3. Applying SnL to a cluster of defects. (a) We identify ancilla and link defects and combine all possible ways of repurposing. (b) For each combination of primitive operations found in (a) we determine which repurposed checks are invalid and disable data qubits in them. This process is simplified by identifying connected components of repurposed checks. We disable all data qubits in a connected component if there is a defect anywhere in that component or if an ancilla qubit is repurposed more than once. These disabled data qubits form a "snake." (c) We apply the data defect primitive around each defective and disabled data qubit, converting stabilizers around them to gauge checks. (d) We identify the super-stabilizers based on the commutation relations. For more details, see Appendix C.

sometimes result in fewer data qubits being disabled, for example, when the data qubit in a defective gate is also defective. This also implies that SnL reduces to DQD if no ancilla can be repurposed. SnL otherwise always outperforms DQD by having smaller weight stabilizers.

Defect clusters grow quickly in number and in size with respect to the defect rate [15]. Consequently, we need a way to maximize the number of repurposed checks, disabling a minimum of data qubits. Figure 3 illustrates how SnL finds valid strategies by combining both primitives for any defect configuration. The first step is to combine all possible strategies for the isolated ancilla and link defects as shown in panel (a) for an example defective configuration. Given a specific combination of repurposed weight-2 checks, we have a systematic way to determine which repurposed checks are compromised and which data qubits to disable. We define a graph $\mathcal{G}$ whose vertices are all the qubits (both ancilla and data) involved in these weight-2 checks. Each edge in $\mathcal{G}$ is a two-qubit gate in one of those checks. Any defect (data, ancilla or gate) or ancilla repurposed more than once in a connected component of $\mathcal{G}$ results in all the data qubits in that same component to be disabled. This ensures that we have no weight-1 check left after repurposing and no ancilla repurposed more than once. We call those strings of disabled data qubits "snakes."

This process is illustrated in panel (b) where a single data defect results in all data qubits in a chain of repurposed checks to be disabled. After having defined which repurposed checks are feasible, we then apply the data defect primitive around each defective and disabled data qubit, i.e., we convert each damaged stabilizer around them to gauge checks. This step is shown in panel (c). We then define the super-stabilizers by combining the remaining repurposed checks with the gauge checks around the disabled data qubits. One of the many possible strategies for the example defect configuration in (a) is shown in (d).

Two additional examples are shown in Fig. 4. Panel (b) shows two of the possible strategies SnL returns for adjacent ancilla and data defects. The data defect renders one of the two repurposed checks invalid: we are therefore left with a single repurposed check and a disabled data qubit. These two strategies still result in a better code distance than DQD in panel (a), where all four data qubits are disabled. Another example is shown in Fig. 4(d). There, in order to avoid repurposing twice, we need to either

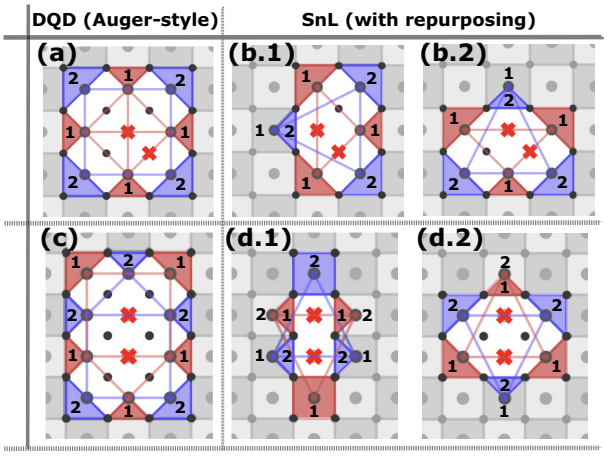


FIG. 4. Application of our strategies for two examples of defect clusters: (a),(b) adjacent ancilla and data defects, and (c),(d) two adjacent ancilla defects. (b) and (d) show the best two strategies that we find for each defect configuration with SnL.

consider a lossy repurposing (opposite to the orientation that preserves code distance) for one of the two ancilla qubits in panel (d.1) or disable two data qubits in panel (d.2). DQD, on the other hand, leads to six data qubits being disabled.

C. Defect clusters on the boundary

Up to this point, we have ignored the impact of the target window's boundary on SnL. We now discuss how we deform the boundary of the code patch when the defect cluster touches the boundary of the target window, in a way that maximizes the code distance.

SnL first entails embedding the original target window in an infinite grid we call the "extended window," such that there are no defects outside the target window. This simplifies SnL by allowing it to treat bulk and boundary defects equally. Figure 5(a) shows an example where we have defined a target window that contains a cluster of two data defects and one ancilla defect. Moreover, one of the data defects is a corner qubit, which we define as a data qubit where an X-type boundary meets with a Z-type boundary. In this example, one possible strategy for this defect cluster includes some checks outside the target window. These checks are invalid and need to be removed. The remaining checks around the cluster must be converted to valid boundary checks, unless they become part of a reduced super-stabilizer in the bulk.

Before adapting the checks in the example defect cluster, we need to move the corner from the defective data qubit to a functional one. The position of the new corner determines what type of boundary check is valid, at each point of the new boundary. The next step of this routine is therefore to identify all new possible corner positions for each strategy we found for the cluster. This step is

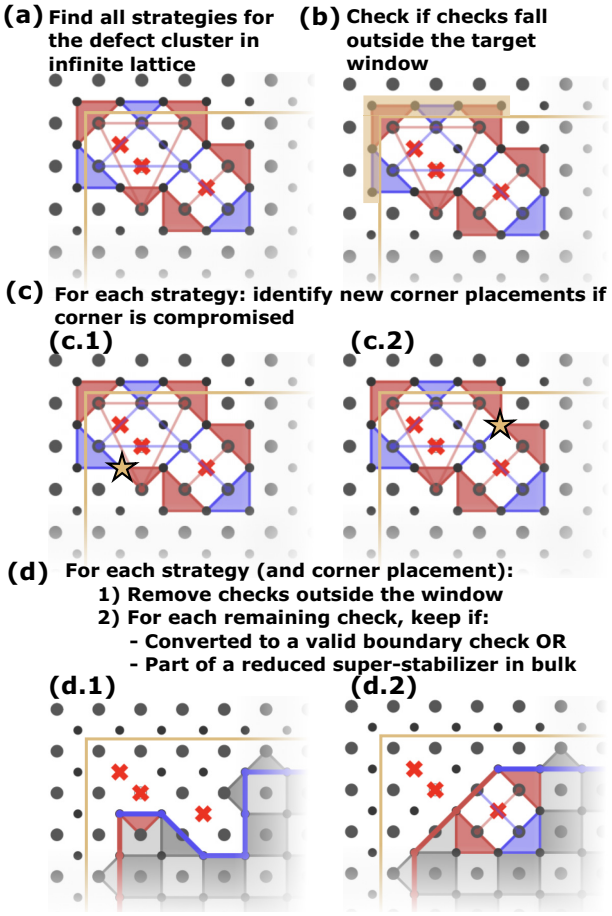


FIG. 5. Routine for the boundary deformation. (a) We find all strategies for the defect cluster. We only show the top left corner of the target window (in yellow) for compactness. (b) For each strategy, we determine if the gauge checks require qubits outside of the target window (highlighted in yellow). (c) For each strategy, we determine if one or more corners are part of the gauge checks or disabled. Here we show (using the yellow star) two possible corner placements for the strategy in (b). (d) If a corner is compromised, we repeat this step for every possible corner placement found in (c), otherwise we only do this step once. We remove all gauge checks that fall outside the target window. For each remaining gauge check we determine if it can (1) be converted to a valid boundary check or (2) be part of a smaller super-stabilizer within the target window such that the commutation relations are satisfied. If not, then we eliminate the check. For example, the two red gauge checks in (c.1) became gray boundary checks in (d.1). On the other hand, the red gauge checks in (d.2) form a new super-stabilizer in the bulk.

illustrated in panel (c). To the best of our knowledge, the optimization of the corner placement has not been considered before, but it helps both SnL and DQD achieve better code distance.

For each of these new corner positions, we must determine which checks need to be removed from the strategy and which remaining checks need to be combined into smaller super-stabilizers. This step is depicted in panel (d).

We first remove all checks (repurposed and gauge) that fall outside the target window. For each remaining check we must determine whether it can be (1) converted to a valid boundary check (a check of the Pauli type corresponding to that of the boundary) or (2) combined with other checks to form a reduced super-stabilizer inside the target window, such that the stabilizers all commute. If none of these two options is possible, we remove the check. In Appendix D we provide a systematic way that maximizes the number of data qubits inside the target window.

We remark that the extended window need not be an infinite lattice, but rather it is enough if it contains two additional layers of data qubits and ancilla qubits on each side of the target window. That way, the data defect primitive in Fig. 2(a) is supported for an ancilla defect on the boundary of the target window. It follows that all other primitives in Fig. 2 are supported for any possible defect in the target window.

Finally, we emphasize that our implementation for the boundary deformation minimizes the number of data qubits that are removed from the code for each strategy. However, in certain cases it is possible to construct a larger-distance code in a smaller area of the target window, for example, by excluding a region with a high density of defects. However, this defers the problem of adapting to all the defects in the target window until we attempt to do lattice surgery with patches outside the target window.

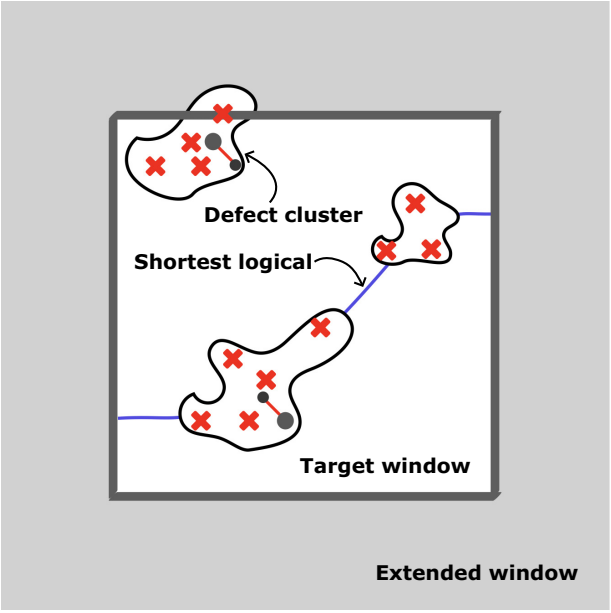


FIG. 6. Cartoon of a processor with multiple defect clusters. For the purpose of the algorithm, we embed a target window in an extended window, which is simply a trick to facilitate our routine for clusters near the boundary. We define defect clusters in the extended window and find all strategies for each cluster individually. We then combine these strategies to form global strategies that maximize the code distance that can be realized within the target window.

D. Recap of SnL on a defective patch

In this section, we describe how we find the best global strategies for a target window of a defective processor (see Fig. 6). Here, we imagine having a defective processor where a random defect configuration can be decomposed into several defect clusters. The goal here is to find a global strategy that maximizes the distance of the surface code patch.

In Secs. III A and III B, we explained how to find local strategies for an isolated defect or a defect cluster. For each local strategy, we can compute the "local distance loss," which is the distance of the surface code patch adapted to this single defect or defect cluster, relative to the maximum distance achievable in the defect-free target window. It is computed with respect to a single defect cluster, without accounting for the other defects. A strategy that minimizes the local distance loss for a given cluster is called "locally optimal." In Sec. III C, we discussed how the strategies for each defect cluster are affected by the target window's boundary (see Fig. 5). A missing puzzle piece is the interdependence of the defect clusters when we evaluate the global code distance, accounting for all defect clusters and the deformation of the boundary.

Finding a combination of local strategies that maximizes the global distance is more complex than independently minimizing the local distance loss at each cluster. Local strategies at two different clusters can sometimes have cumulative effects on the code distance, e.g., if there is a minimal logical string passing through both clusters (see Fig. 6). This interaction between local strategies is complex and depends on the relative positions of all clusters. Therefore, to find the best global strategy, we need to consider multiple local strategies at each cluster.

Using this observation we summarize the steps of our algorithm in Algorithm 1. First we identify the defect clusters in the extended window. We then make a list of local strategies for each defect cluster as in Sec. III B. We define a global strategy by composing the local strategies and optimize boundary deformation for each as in Sec. III C. We return the global strategy resulting in the a surface code patch with the best distance (see Appendix E for how it is computed). For simplicity we omit the heuristics (Sec. III E 2) from the pseudocode.

E. Reducing the runtime and memory costs

Below, we discuss two key steps that improve the scalability of SnL.

1. Defect clustering

A key step of our algorithm is to define the smallest possible defect clusters, which ensure the strategies

ALGORITHM 1. SnL achieves the optimal surface code distance in a target window of the defective processor.

```
procedure SnL(W, [p_i])
    input: target window W, defect locations [p_i]
    output: optimized defect strategy
    W̄ ← extended window given W
    [D_i] ← defect clusters in W̄ given [p_i]
    for all defect clusters D_i do
        [R_j] ← combinations of ancilla repurposings
        s_i ← [ ]          // local strategies for D_i
        for all combinations R_j do
            s ← initialize strategy with R_j
            Disable data qubits in invalid checks
            Build gauge checks
            if valid stabilizers then add s to s_i
    [S_k] ← Cartesian product s_1 × s_2 × ···
    for all global strategies S_k do
        Remove checks outside W
        Move corners as needed
        Adapt remaining checks
        if valid boundary then keep strategy
    return global strategy with best distance
```

for different clusters do not have overlapping checks. The smaller the cluster, the fewer the ways of repurposing, thus facilitating the search for local optimal solutions for each cluster based on the code distance loss. The method we use to define the smallest defect clusters is detailed in Appendix F. Put simply, we use DQD (with the link defects mapped to ancilla defects instead) to determine which qubits to group together since the super-stabilizers obtained with SnL are always strictly contained inside those defined by DQD (see Fig. 2).

2. Heuristics

SnL relies on trying two choices of ancilla repurposing for each ancilla and link defect (depicted in Fig. 3). We also consider treating each link defect as a data defect and applying the data defect primitive. We therefore find up to $2^a 3^l$ unique strategies for a cluster with a ancilla defects and l link defects. While enumerating all possible strategies is feasible for small defect clusters, the runtime and memory overhead (which scales proportionally with the number of strategies) become prohibitive for large defect clusters. To reduce the computational cost of SnL, we choose to consider only a subset of the possible combinations of repurposings in each cluster.

We first sort all combinations of repurposing strategies so that the most promising ones are considered first, i.e., those that minimize the number of disabled data qubits and use the distance-preserving repurposings (see Appendix G 1). We then iterate through them to remove the "snakes" and evaluate the local distance loss. In this process we skip some combinations or stop early based on conditions defined by several parameters (see Appendix G 2), most of which are related to the distance loss. Finally, we filter the strategies we found such that only a subset will be used for the global strategies. For example, only the locally optimal strategies for each cluster could be combined into a global strategy.

Our heuristics allow us to consider fewer strategies while guaranteeing that we evaluate those that most likely locally minimize distance loss, resulting a runtime that is subexponential. Depending on the choice of parameters, we ultimately find that the runtime and memory of SnL can be made comparable to what is achieved with DQD while simultaneously achieving significantly larger distances in general.

IV. NUMERICAL SIMULATIONS

In this section, we demonstrate how SnL leads to a significant improvement at all defect rates with respect to both the realized code distance and logical performance for arbitrary defect configurations relative to DQD, which includes the primitive from Ref. [10] along with the bandage-like super-stabilizers in Ref. [15] and our optimized boundary deformation (i.e., we consider the best known implementation of DQD without weight-1 checks). Then, we also present numerical simulations for the logical performance, for the yield of preserving full distance, and for the logical CNOT gate based on lattice surgery.

The setting we consider for the numerical results in Secs. IV A to IV C is a target window as in Fig. 1(a) with dimensions $d_{\text{targ}} \times d_{\text{targ}}$, where d_{targ} is the target code distance. We sample random configurations of defects and apply SnL or DQD to construct optimized code patches within the target window to carry out memory experiments.

A. Effective code distance

In this section, we report the effective code distances achieved with SnL for random defect configurations and compare them to those obtained with DQD. In what follows, we denote the code distance of an adapted surface code by d_{out} (output distance), the relative code distance by $\widetilde{d} = d_{\text{out}}/d_{\text{targ}}$, and their mean values over multiple samples by $\langle d_{\text{out}} \rangle$ and $\langle \widetilde{d} \rangle$. Unless otherwise specified, the value of d_{out} we present will be the minimum of d_X and d_Z.

We randomly sample 1000 defect configurations for each combination of defect rate ($q = 0.001$ and $q = 0.01$) and target distance d_{targ} (using a $d_{\text{targ}} \times d_{\text{targ}}$ target window) ranging from 3 to 69, and generate adapted patches for each of them. Each qubit and link on the processor, including the padding ancillas, is independently assigned defective at the same probability, q, unless otherwise specified. We do not require the configurations to be unique.

For distances up to 17, we run SnL using a hierarchy of heuristics (culminating in the unconstrained global optimization) and report their performance in Appendix H. We

run the different heuristics in order of increasing runtime and stop for a given configuration when the runtime or memory usage become prohibitively large. We are able to run this on most defect configurations with 1% defect rate up to distance $d = 13$, but for larger distances we ran most configurations only with heuristics. We keep the code patch that has the largest d_{out}, and break ties using the sum of d_X and d_Z, followed by the number of active data qubits. Moreover, with ancilla padding, we also have the option to use either half of the boundary ancillas for the native weight-2 checks in our code. Keeping the orientation of the logical operators fixed, this amounts to the freedom of reversing the Pauli type of checks. In order to produce optimal code patches, we consider both choices and record the best code distances. To save computational resources, we use stronger heuristics and do not consider both choices of boundary ancillas for distances starting from 21 (see Appendix H for details on the heuristics for large distances).

We report the average relative distance $\langle \tilde{d} \rangle$ for different d_{targ} in Fig. 7(a). Each data point represents 1000 random defect samples, with shaded regions indicating 95% confidence intervals. We find that SnL significantly outperforms DQD, especially for higher defect rates. The relative distance trends to a constant as the target distance increases, indicating a stable performance trend at larger scales. For $q = 0.01$, the relative distance by SnL converges to 0.79 whereas DQD to 0.54. The drop between the $q = 0.001$ and $q = 0.01$ curves for SnL is primarily due to data defects (see Appendix I 1 for details). We also plot the histogram of output distance for $d_{\text{targ}} = 21, 61$ and for defect rate $q = 0.01$ in panel (b). The average output distance for SnL is clearly higher than for DQD, and importantly, the narrowing of the distribution indicates improved reliability of yield for patches with defects.

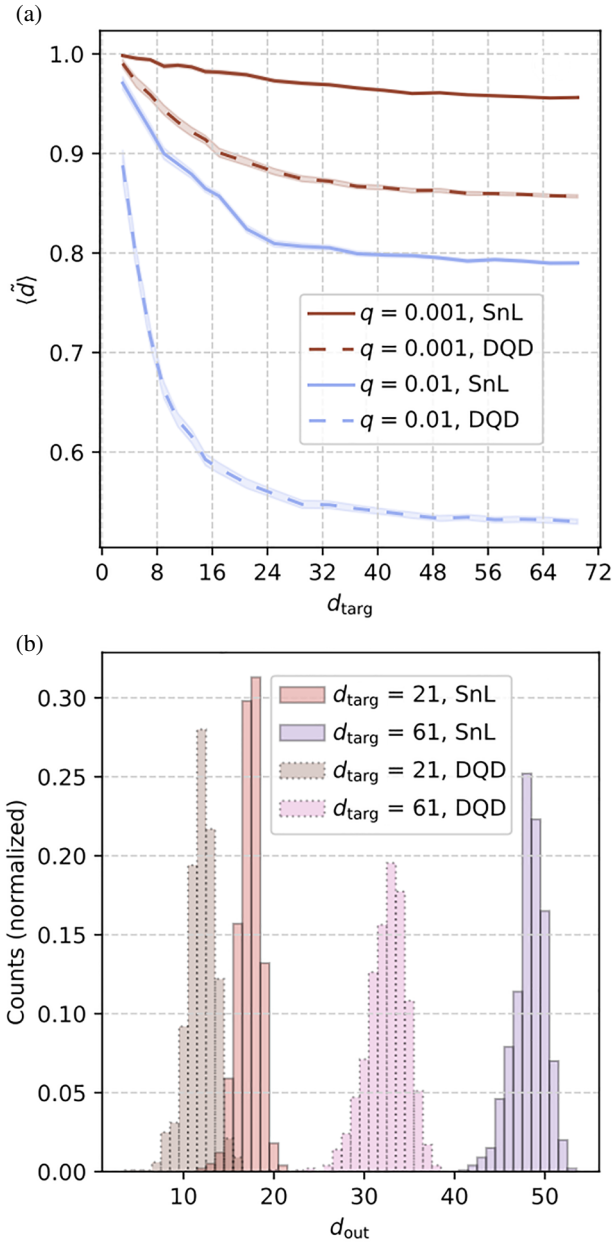


FIG. 7. (a) Average realized distance relative to the target distance, $\langle \tilde{d} \rangle = \langle d_{\text{out}} \rangle / d_{\text{targ}}$. The maximum value is 1 and indicates that we recover the full distance. (b) Histogram of realized distances for large distances and defect rate $q = 0.01$. For SnL $\langle d \rangle$ is much greater and the distribution width is smaller compared to DQD.

B. Logical performance

The effective code distance is a good indicator for the logical performance of the surface codes adapted with super-stabilizers. However, our protocol introduces a new primitive for defective ancilla qubits, so it is important to test that this primitive does not compromise logical performance.

Here we present Monte Carlo simulations of logical memory obtained with `stim` [20]. We use the `PyMatching 2` decoder [21] and the SI1000 noise model [22], which includes errors on idle qubits. Compared to the error models used by prior work, SI1000 is more realistic, especially because super-stabilizers introduce more idle locations on qubits. We run each simulation for $2d_{\text{targ}}$ rounds.

We use a standard surface code syndrome extraction schedule (see Appendix K) for all unmodified stabilizer checks. For the checks we modify as part of our defect strategy we retain the same schedule but without the gates that we remove. For example, around a data defect, the four neighboring ancilla qubits measure weight-3 gauge checks. The schedule for these weight-3 checks is the same as the ordinary schedule but with an idle round inserted where the fourth gate would ordinarily go. For repurposed checks we adopt the schedule of the defective ancilla. Therefore, from the point of view of the data qubits involved in the

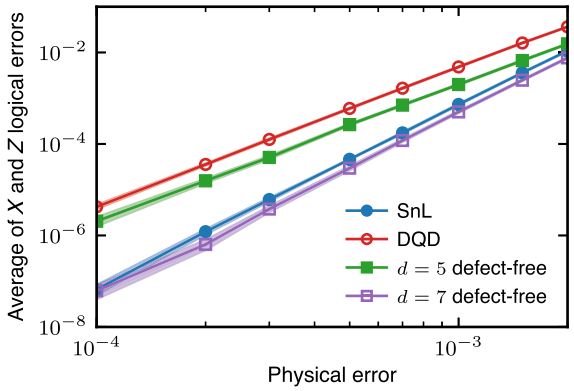


FIG. 8. Logical performance in the subthreshold regime. Different adaptations to the same distance-7 rotated surface code with a defective ancilla qubit.

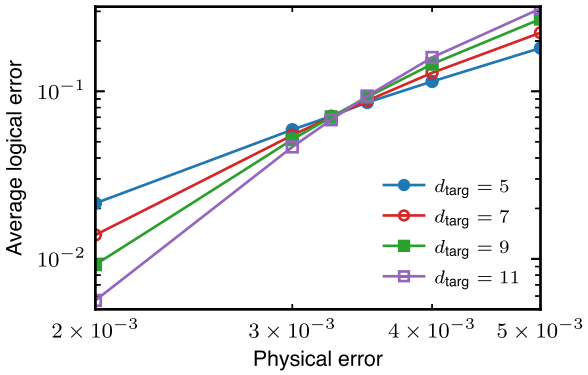


FIG. 9. Threshold plot for the surface code adapted via SnL. The simulations are run for $2d_{\text{targ}}$ rounds with the SI1000 noise model. Each point is the average logical error rate (between X and Z) on a sample of 1000 patches with 1% defect rate.

check, they are still measured in the same time steps they would ordinarily be measured. This way we prevent additional hook errors that might be introduced by modifying the check schedule.

First, we test the distance-preserving repurposing in Fig. 2(c) on a distance-7 rotated surface code that has a single defect on an ancilla qubit in the bulk. SnL retains the code distance of the patch, while DQD reduces the code distance to 5. As Fig. 8 shows, our adapted patch nearly matches the logical performance of the distance-7 defect-free code, and significantly outperforms both the distance-5 defect-free code and DQD.

We also numerically study how defects influence the threshold of the code (see Table I, Fig. 9 and Appendix I 2 for the plots used to extract those values) [23]. We run logical memory experiments on patches with target distance 5, 7, 9, and 11 adapted to random defect configurations that we sampled at a defect rate of 1%, and average the results. Here, we measure the X-type and Z-type super-stabilizers on alternating rounds for a total of $2d_{\text{targ}}$ rounds. When SnL is used, a defect rate of 1% decreases the threshold of surface code (under the SI1000 error model) from 0.42% in the defect-free case to 0.33%. However, with DQD, the threshold drops further to 0.27%. We also run a set of simulations using a standard depolarizing noise model [24] that does not include idling errors. Under this model, SnL also results in a higher threshold than DQD [25].

TABLE I. Threshold values under the standard and SI1000 noise models, for the defect-free patches and patches with 1% defect rates adapted with either SnL or DQD.

	Standard	SI1000
Defect-free	0.86%	0.42%
SnL (our approach)	0.72%	0.33%
DQD (baseline)	0.57%	0.27%

Compared to the standard surface code, our defect-adapted surface codes have larger stabilizer weights, scaling with the size of the defect clusters. The larger the stabilizer weight the more difficult it is for the decoder to determine where the errors occur. Moreover, the super-stabilizer measurements combine multiple uncertain gauge checks measurements. This is because all gauge checks that anti-commuted with the previously measured gauge checks now have random measurement outcomes. One approach to improve performance is to use shells (or gauge fixing) [11,12,26], which was shown to yield a threshold for DQD [11]. By repeatedly measuring a commuting subset of the gauge checks, our measurements are deterministic. As a result, the resolution and reliability of our syndrome information can be improved. Shells can improve the logical performance for both DQD and SnL, especially for large defect clusters. However, SnL usually results in significantly smaller super-stabilizer weights. We leave the analysis of gauge fixing with SnL for future work.

C. Yield for preserving full distance

We now shift our focus slightly to consider yield, which we define as the probability of preserving the full distance of the code ($d_{\text{out}} = d_{\text{targ}}$) in the presence of defects. Here we not only compare SnL with DQD within a target window of a larger surface code processor, but also study the impact of different padding strategies on a small chip where the target window is the entire chip. Here we compare "no padding," wherein we consider a chip with only the boundary ancillas necessary for a standard rotated surface code, "ancilla padding," wherein we have extra ancillas around the perimeter of our code as we do in the target window shown in Fig. 1 and "$d + 1$," wherein we fabricate a patch of size $d_{\text{targ}} + 1$, with only the necessary boundary ancillas, to increase the probability of realizing a distance-d_{targ} patch. Ancilla padding is more defect-tolerant than no padding because we are able to use

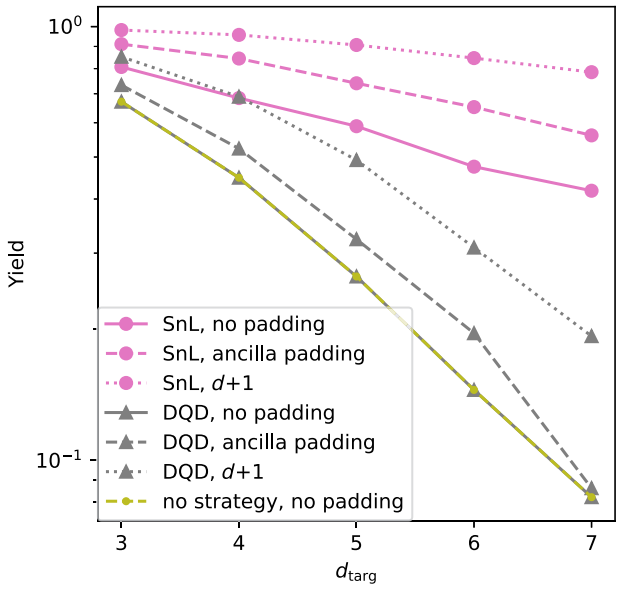


FIG. 10. Yield of full-distance ($d = d_{\text{targ}}$) surface code for three different padding strategies: no padding, ancilla padding, and $d + 1$. The defect rate is fixed at 1% for qubit and link defects. "no strategy, no padding" refers to the yield for a chip with no padding, where we discard the samples with any defective components.

the extra ancillas for ancilla repurposing in certain cases, but it requires an additional $2(d_{\text{targ}} - 1)$ ancilla qubits. The $(d + 1)$ approach enables us to tolerate any one data qubits defect and even some configurations of multiple data defects. This requires $4d_{\text{targ}} + 2$ more qubits than the no-padding approach.

In Fig. 10, we show the yields achieved by SnL and DQD with each padding strategy. For all padding strategies, SnL significantly increases the yield relative to DQD. With no padding, DQD obtains a 10% yield at $d = 7$, while SnL achieves a yield of 50%. However, ancilla padding (which adds 12 ancilla qubits for $d = 7$) increases the yield to 65% for SnL. The "$d + 1$" padding strategy, which adds 30 physical qubits compared to "no padding," further increases the yield to 80%. Moreover, DQD with no padding has the same yield as using no strategy at all, because it cannot preserve the full distance in the presence of any defect. We observe that the padding strategies do offer the expected gains relative to the no-padding baseline. The gain for DQD with ancilla padding might seem surprising because DQD does not make use of ancilla repurposing. The reason for this gain is that with ancilla padding we have two choices for boundary ancillas. We choose one of two disjoint sets of alternating ancillas around the boundary. This gives us the flexibility to avoid boundary ancilla defects in some cases. For SnL we benefit from this effect as well as the additional ancilla repurposing possibilities.

D. Lattice surgery

Compared to DQD, SnL produces patches with higher distances in the presence of defects. Consequently, the logical operations between the patches have higher fidelity. Aside from this, SnL facilitates lattice surgery for two additional reasons. First, the boundaries of the adapted patch are less likely to be significantly deformed. Second, the holes in the routing space have smaller areas.

In this section, we demonstrate that it is possible to realize a lattice-surgery-based logical CNOT gate between two surface code patches adapted using SnL. A basic example is shown in Figs. 11(a)–11(c). The three patches remain full-distance in the presence of ancilla and link defects, but it would not be possible to adapt these patches with DQD. We also show how to apply SnL to optimize for lattice surgery operations rather than individual logical qubits. In what follows, we now use d to represent the target code distance for conciseness.

First, we need to define the patches for the control, ancilla, and target qubits. To do so, we apply SnL to the entire processor, comprising the control, target and ancilla patches as well as the routing space, as if it was a single patch. Second, we choose a rectangular target window around each subpatch, preserving the super-stabilizers we found for the entire processor, and recompute the new optimal boundaries for that subpatch. This guarantees that as long as the merged patch and the separate patches have minimum distance d, then the logical information is encoded with distance d throughout the process. It is straightforward to define the boundaries of the subpatches once the rectangular target windows containing them are defined, in order to optimize their effective code distance. After having defined the subpatches for the control, target and ancilla qubits, shown in Fig. 11(a), we define the merged control-ancilla and ancilla-target subpatches, illustrated in Figs. 11(b) and 11(c), respectively. We are now ready to set up the CNOT gate numerical simulation.

We first stabilize the control, target, and ancilla patches over $r = 2d$ (d) measurement rounds, where d is the code distance, for the defective (defect-free) patches. To realize a CNOT gate between the control and target patches, we then merge the control and ancilla patches [16,17]. We do so by measuring the product of their logical operators $Z_L Z_L$: we measure all Z-type stabilizers and super-stabilizers between the bottom Z and top Z logicals of the control and target patches, respectively. We then stabilize the merged control-ancilla patch over $2d$ (d) measurement rounds. We split the merged patch back into separate ancilla and control patches by measuring the data qubits in the routing space in the X basis. We then stabilize the split patches over an additional $2d$ (d) rounds. We then merge the ancilla and target patches by measuring $X_L X_L$: this time, we measure all X-type stabilizers and super-stabilizers between the right X and left X logicals of the ancilla and target patches, respectively. We again stabilize

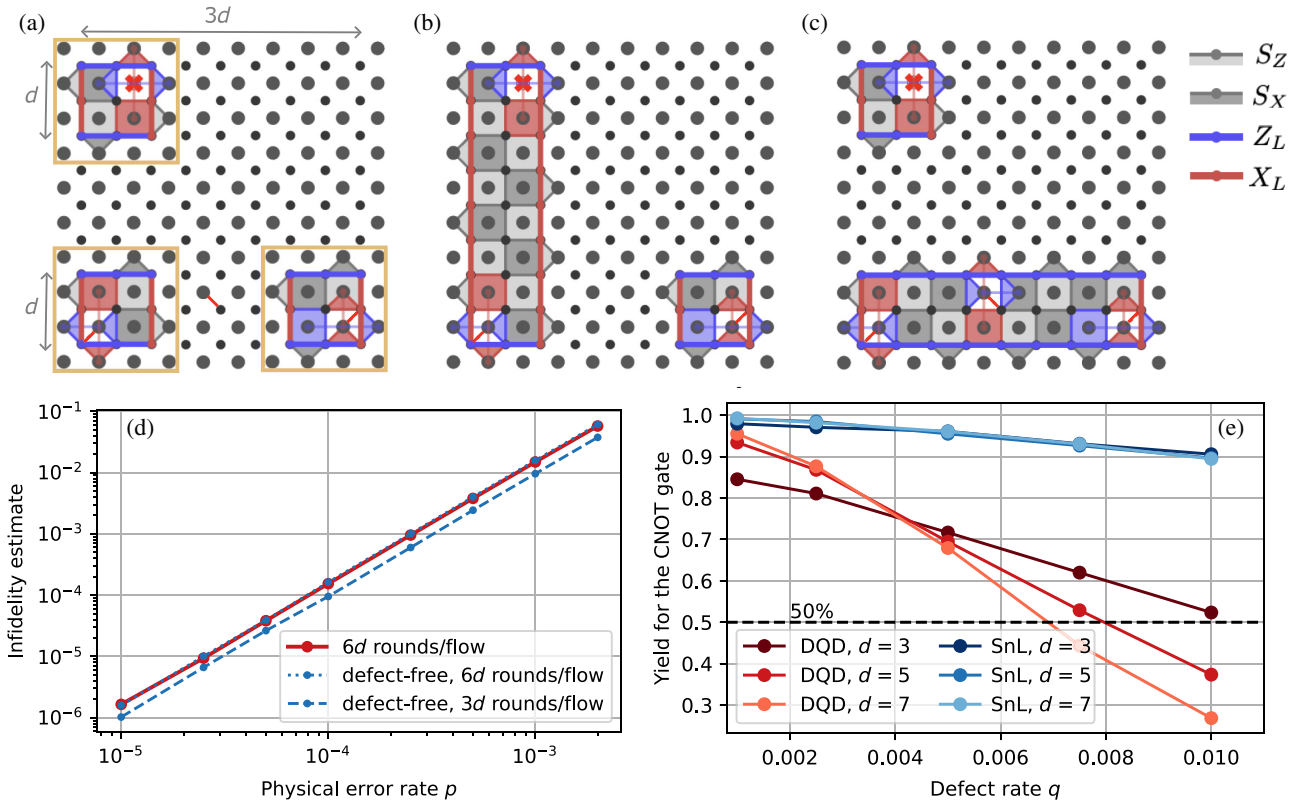


FIG. 11. Lattice-surgery-based CNOT gate with defects. (a) The control (top left), ancilla (bottom left), and target (bottom right) patches in a $3d \times 3d$ defective processor. The target window (in yellow) used for each patch has dimensions $d \times d$. Here $d = 3$. (b) The merged control-ancilla patch (left). (c) The merged target-ancilla patch (bottom). (d) Estimate of the infidelity for the CNOT gate using the patches in (a). The errors are cumulated from the four flows (XX and ZZ merge and split experiments initialized and measured in the X or Z basis) over $6d$ (defective patches) or $3d$ (defect-free patches) measurement rounds per flow, where $d = 3$. The CNOT gate itself takes place over $10d$ (defective case) or $5d$ (defect-free case) measurement rounds. (e) Yield experiment in a $3d \times 3d$ defective processor with the same layout as in (a) but where the control, ancilla and target patches are expanded into the routing space if needed to recover an effective distance d for each patch. A failure occurs if any of the patches overlap.

the merged patch over $2d$ (d) measurement rounds. We now split the merged patch by measuring in the Z basis. We then stabilize the codes one last time with $2d$ (d) measurement rounds. Overall, the gate takes $10d$ ($5d$) measurement rounds for the defective (defect-free) patches.

Since the `stim` package that we use for numerical simulations does not allow nondeterministic observables, we split the CNOT experiment into four flows [27]. We simulate lattice surgery between the control and ancilla patches, and between the ancilla and target patches, separately. We initialize and measure each pair of patches in either the X or Z basis. In each of the four flows, there are one or three deterministic observables. To estimate the infidelity of the logical CNOT as a function of the physical error, we sum up the average error rates for the observables from all flows and divide by two (averaging over the X and Z basis). See Appendix J 1 for more details. Each flow takes $6d$ ($3d$) rounds for the defective patches (defect-free patches): $2d$ (d) to stabilize the patches, $2d$ (d) during the merge and $2d$ (d) after the split.

In Fig. 11(d) we report the average error in the two lattice surgery experiments as a function of the physical error rate, for the defective patches shown in panels (a)–(c) and their defect-free counterparts, averaged over five million Monte Carlo samples. Each point is averaged over 1000 random defect configurations. We find that the logical error rates are approximately the same (when rescaled to take into account the different number of error-correction rounds) for the defective and defect-free processors. For this simulation, we used the standard noise model.

We now move to a yield analysis for the CNOT gate and compare SnL to DQD. Here we consider random defect configurations on a $3d \times 3d$ processor with different defect rates. Our goal is to realize a CNOT gate with three distance-d surface code patches. We therefore define three $d \times d$ target windows at the top left, bottom left and bottom right corners of the processor such that we have the same layout as in Fig. 11(a). For each target window, we find the surface code patch with largest output distance $d_{\text{out}} = \min(d_X, d_Z)$, and expand the window by size 1

vertically if $d_X < d$ and horizontally if $d_Z < d$. We repeat this process recursively until the algorithm yields three non-overlapping patches of effective distance $d_{\text{out}} \geq d$ and with sufficient room for routing (i.e., there must be at least one layer of data qubits separating each pair of patches). If we fail to produce such patches within the processor, we record a failure. We present the yield of the CNOT gate in Fig. 11(e) as a function of the defect rate, for different distances d. This numerical experiment is further detailed in Appendix J 2. There are two crucial observations. First, at a 1% defect rate, DQD results in a yield around 50% for $d = 3$, which plummets as we increase d. Second, the yield for SnL barely changes with d and remains above 90%. Both trends can be explained by the relative distances achieved by the two methods as d grows [Fig. 7(a)].

V. CONCLUSION

In this article, we introduced SnL, a suite of novel and highly performant strategies for adapting the surface code to defective qubits and two-qubit gates. SnL relies on repurposing ancillas (for ancilla and link defects) to implement weight-2 gauge checks, allowing for reconstruction of missing stabilizers and helping preserve the distance of the adapted surface code. Additionally, all super-stabilizers are measured over two error-correction rounds, just like in the state-of-art methods [10–15].

Using SnL, we demonstrated a large gain in the distance of the adapted surface code over existing methods for code sizes up to thousands of qubits. For realistic defect rates the adapted surface code exhibits logical error rates close to the error rates of the defect-free surface code and a few orders of magnitude better than with DQD. SnL also greatly increases the probability of realizing a full-distance surface code. At the same time, computer resources needed for SnL can be on par with resources used in previous methods.

To better appreciate the impact of SnL we also benchmarked the performance of a logical CNOT gate implemented via lattice surgery. We found similar performances for the adapted and defect-free surface codes with the same code distances. We also performed a yield estimation and found that SnL has a significantly larger yield for both memory and lattice surgery compared to DQD. The yield quickly drops with the code distance for DQD whereas it remains approximately constant for SnL. Our results convincingly demonstrate that in the presence of defects SnL shows a major improvement not only for memory storage, but also for logical operations.

SnL enables more hardware-efficient quantum computation with the surface code when hardware components have a non-negligible probability of failure. Between variations in targeting, fabrication defects, and time-dependent TLSs, this will be the reality of superconducting qubit devices for the foreseeable future. Therefore, SnL can make fault-tolerant quantum information processing significantly more attainable and resource-efficient. For transparency and reproducibility, the code used for the numerical simulations is readily available on Github [19].

We identify a few directions of future work. First, in our logical error rate simulations, we measured X and Z gauge checks in alternating rounds. Measuring the same type of gauge checks for multiple rounds before switching can lead to better logical performance for large super-stabilizers [12]. Second, SnL yields larger-distance surface codes than strategies involving weight-1 gauge checks used in Refs. [14,15], however using weight-1 gauge checks in conjunction with ancilla repurposing can give better distance for some defect configurations (see Appendix A). Third, we constrained our ancilla qubits to be repurposed at most once. In some cases, however, it is possible to achieve larger distance by allowing ancilla qubits to be repurposed twice, at the expense of no longer measuring all super-stabilizers within two quantum error-correction rounds. Fourth, SnL may be optimized for biased noise, for instance by considering Clifford deformations [28–30] of the adapted surface code. Further study of the logical performance under realistic noise assumptions is necessary to better understand the trade-offs involved.

Note added. While finalizing the publication release process, we became aware of an independent work by Debroy *et al.* [31], which addresses the problem of adapting the surface code in the presence of defective components.

ACKNOWLEDGMENTS

We thank Prasahnt Sivarajah, Arne Grimsmo, Colm Ryan, Will Morong, and Fernando Brandão for fruitful discussions about the algorithm and the experimental implementation of our strategies. We also thank Simone Severini, Bill Vass, James Hamilton, Nafea Bshara, Peter DeSantis, and Andy Jassy at Amazon, for their involvement and support of the research activities at the AWS Center for Quantum Computing.

DATA AVAILABILITY

The data that support the findings of this article are not publicly available. The data are available from the authors upon reasonable request.

APPENDIX A: WEIGHT-1 CHECKS

Naively, weight-1 checks contradict the idea of having a stabilizer code by measuring a data qubit directly. However, Ref. [15] suggests to enable weight-1 checks such as to prevent the "avalanche" effect that disables a large area of qubits. This avalanche effect results from the recursive search of weight-1 checks to disable, which ultimately leads to the holes to be rectangular in the rotated surface

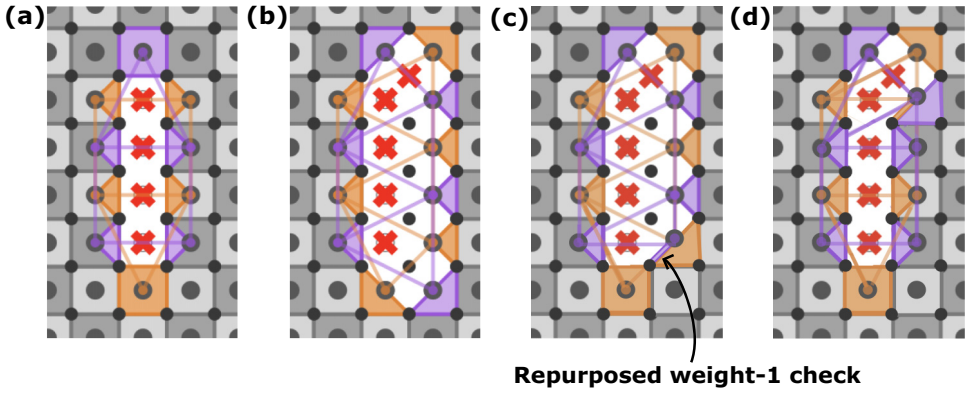


FIG. 12. (a) Chain of adjacent ancilla defects. (b) Adding a data defect to the configuration in (a). (c) and (d) are examples where a repurposed weight-1 check could help the code distance vertically. However, (d) utilizes more checks overall. The weight-1 check is at the bottom right in (c) and top right in (d).

code. It follows that allowing weight-1 checks helps keep more data qubits active. This was also studied in Ref. [14].

The use of ancilla repurposing however already strongly suppresses this avalanche effect and in fact preserves significantly more data qubits. Indeed, with DQD this effect is usually triggered by ancilla defects since DQD disables all four neighboring data qubits around an ancilla defect. There are cases where weight-1 checks could be useful for SnL too, particularly when repurposing is not available, but it appears that they generally do not help the code distance and result in a larger error rate. Rather than allowing all gauge-1 checks, one should selectively allow them in cases where they are expected to result in gains, such as when all three data qubits around an ancilla are defective. We leave this addition for future work since the choice is not trivial.

Here, we elaborate on why we decide not to allow repurposed weight-1 checks (i.e., when a repurposed weight-2 check is invalid because of a defective data qubit or a defective link). We however stress that this constraint could be relaxed given clearer rules on which repurposed weight-1 checks are beneficial for the code distance and which are not.

When using SnL, long "snakes" of disabled data qubits arise when we have a chain of at least two adjacent ancilla defects such as in Fig. 12(a). As long as there is a data defect on one side of the chain [see panel (b)], all the data qubits on that side will be disabled. On the other hand, allowing for repurposed weight-1 checks would allow us to disable only the data defect in that case [see panels (c) and (d)]. If the data defect is at one of the endpoints of the chain, this increases the vertical code distance by 1, and otherwise by 2. This is one of the rare occasions where repurposed weight-1 checks help us recover more of the code distance. Now we notice that the code distance loss is the same for both panels (c) and (d), however, panel (d) has a lot more (repurposed) checks. Although they both have the same code distance, we would prefer (c) over (d) because the additional repurposed checks and the higher-weight stabilizer checks will introduce more noise to the stabilizer readouts. In fact, if more data defects are present along the chain, we would still prefer to use repurposed weight-1 checks only at the endpoints. This shows that in general, we do want to disable most data qubits along the "snakes" except potentially at the endpoints.

In summary, we think that disabling data qubits in "snakes" is generally a better strategy but emphasize that repurposed weight-1 checks could, in principle, improve the code distance in some cases.

APPENDIX B: ANCILLA REPURPOSING

1. Repurposing on the boundary

Here we show how ancilla repurposing applies to the defects on the boundary of the code patch. A defective ancilla qubit that measures a weight-2 stabilizer can be handled by repurposing its neighbor ancilla in the bulk to measure the check in alternate rounds. Figure 13(a) illustrates this with a defect on the top left. If instead we have a defective ancilla qubit [such as the other ancilla defect

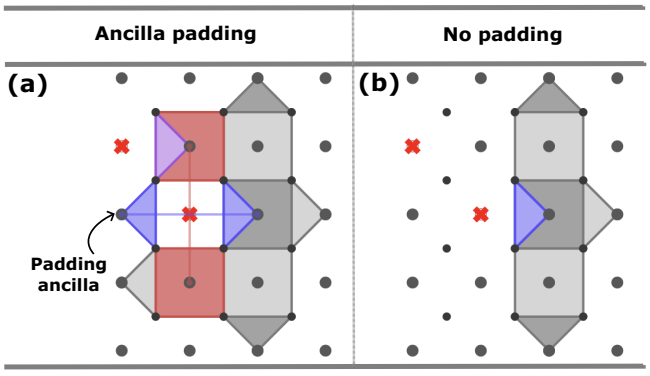


FIG. 13. Ancilla repurposing for defective ancilla qubits on the boundary.

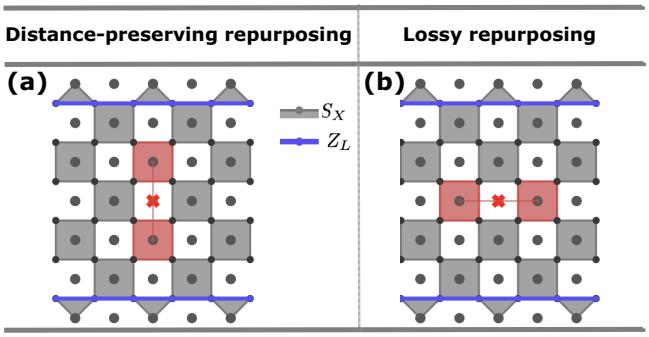


FIG. 14. Distance drops when using repurposed checks for an ancilla defect for both orientations. One preserves the distance whereas the other leads to a distance drop by two in the direction along the weight-8 stabilizer.

in Fig. 13(a)], we can use the "padding" ancilla qubit next to it for one of the repurposed weight-2 checks. Such an ancilla qubit is not part of the original surface code patch, but if it is present in our grid of qubits, we can use it to recover larger distance. The patch in Fig. 13(a) includes two defective ancilla qubits but suffers no distance loss. However, if the padding ancilla qubit was unavailable, the boundary deformation would lead to a distance loss as illustrated in panel (b).

2. Impact on the code distance

We illustrate the impact of repurposed checks around an ancilla defect on the distance in Fig. 14. The orientation for which the weight-8 stabilizer is aligned with the logical of opposite Pauli type results in a distance drop by 2 along that direction.

APPENDIX C: DEFINING THE SUPER-STABILIZERS

We define the super-stabilizers of X- (or Z-) type by building a graph whose vertices are ancilla and data qubits. We add weightless edges between every defective and disabled data qubit and its neighboring ancilla qubits that would be used to measure X- (or Z-) type stabilizers in the defect-free patch. For every repurposed check we also add an edge between the replaced ancilla and the repurposed ancilla if the replaced ancilla would measure a X- (or Z-) type stabilizer in the defect-free patch, otherwise we add an edge between the pair of ancillas that is perpendicular to the pair of repurposed checks. Super-stabilizers are then defined by the connected components in this graph, i.e., we simply multiply the gauge checks and repurposed checks associated with ancilla contained in the connected component.

We remark that we only keep strategies where the super-stabilizers satisfy all commutation relations. The graphs we build to obtain the super-stabilizers are also closely related to those used to compute the code distance (see Appendix E). When defining the super-stabilizers for a given strategy, we can therefore simultaneously compute the distance loss for the cluster and check the validity of the strategy in terms of the commutation relations.

APPENDIX D: BOUNDARY DEFORMATION

To facilitate the boundary deformation, we introduce the term *hole* [see Fig. 15(a)] to denote a set of inter-dependent super-stabilizers whose gauges commute with the checks outside the set but not with all the gauge checks within the set. Usually, a hole has a pair of X and Z super-stabilizers, but it can also have more super-stabilizers in complex cases.

We identify the *holes* by grouping the super-stabilizers we found in the cluster based on commutation relations. We then need to determine which ones require boundary deformation because they contain gauge checks that are outside the target window. Such holes are denoted *open holes*.

Next we define the term *cycle* [see Fig. 15(a)] which is a closed and chordless contour with data qubits as

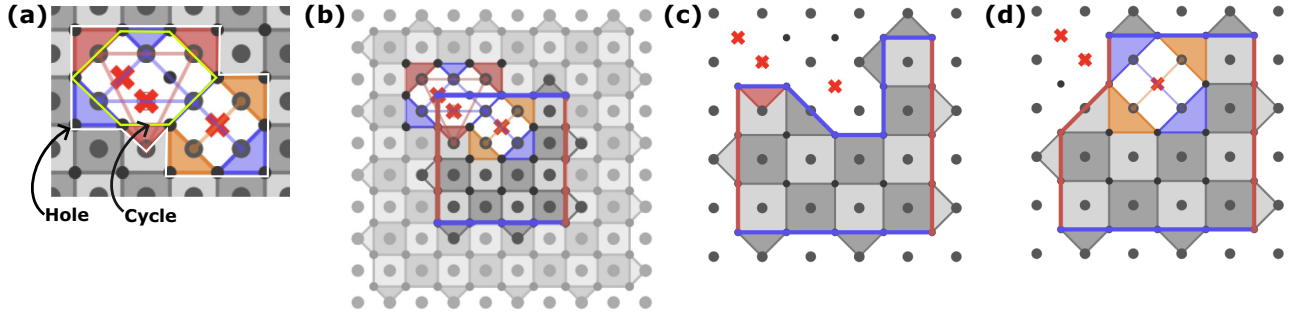


FIG. 15. Definition of a hole and a cycle. (a) A hole (contoured in white) with two cycles. One of them is highlighted in green. (b) Hole enclosing the corner of the initial patch. The hole is *open* because some of its gauges fall outside the target window, and removing them breaks the commutation relation between the super-stabilizers. (c),(d) Two of the possible boundary deformations for the patch. (c) Gauges of the wrong type along both cycles are removed because they share a gauge of this type. (d) Only the gauges along the cycle enclosing the corner are removed.

vertices, that encloses only defective components. The gauge checks along each cycle are of alternating Pauli types and overlap at only a single data qubit. We clean the gauges around an open hole following one of the two possible cases.

1. Edge hole

The first case is when the open hole does not involve a corner qubit, in which case we call it an edge hole. This is the simpler case and always leads to a single solution for boundary deformation.

First we identify the *open* cycles. Similar to open holes, the open cycles are those with some gauge checks outside the target window. Along those cycles, we remove the gauge checks of the type opposite to that of the boundary. We then examine the remaining cycles in the hole to check if they had a gauge check removed; if so, they have become open and we also need to remove their gauge checks as well.

To simplify the process we therefore create a graph whose vertices are the cycles, and add an edge between every pair of vertices corresponding to cycles that share one (or more) gauge checks of the Pauli type to remove. All the connected components that include an open cycle (at the boundary) will tell us which cycles will become open. We now simply have to remove the gauge checks of the wrong type. The remaining gauge checks in those open cycles will become boundary checks in the deformed patch.

Finally, the gauges along the closed cycles will be combined into smaller super-stabilizers in the bulk.

2. Corner hole

The second and more complex case is when the open hole contains a corner qubit, in which case we call it a corner hole. We need to evaluate multiple possible boundary deformations, because we now need to define a new corner in the most optimal way possible.

We consider all the data qubits along open cycles and contained inside the target window as potential new corners. For each potential corner, we then remove all the gauge checks of the type opposite to that of the boundary. The approach is similar to the "edge hole" case.

Our software does not handle the very rare case that the hole contains more than two corners. In such a case, the user can define a smaller target window to avoid the problematic defect cluster.

APPENDIX E: COMPUTING THE CODE DISTANCE

In order to calculate the code distances for both types of logical errors, we need to find the shortest horizontal and vertical logical operators.

Our method starts by building a graph whose vertices are the ancilla qubits used to measure super-stabilizers of the Pauli type P opposite to the Pauli type Q of the horizontal and vertical logical operators, and whose edges have weights equal to the Chebyshev distance between the nodes. We also add the virtual boundary nodes (which will be the endpoints of all logical strings in the graph sitting on the boundaries of the ideal defect-free patch) and the data qubits along the open holes on the two boundaries of Pauli type Q to the graph (to compute the distance loss due to the boundary deformation). We then introduce a weight-0 edge between every pair of ancillas in every super-stabilizer as well as a weight-0 edge between every pair of repurposed and replaced ancillas in repurposed weight-2 checks. Finally, we add an edge between each virtual boundary node and every ancilla whose native stabilizer type is P (i.e., we do not include repurposed ancillas) and every data qubit in the graph (due to boundary deformation) with its weight equal to the Chebyshev distance. The shortest logical length between the two virtual boundary nodes is then obtained using the Dijkstra algorithm.

Further details can be found in our repository [19]. We use this approach both to (1) compute the code distance of the output patches and (2) compute the distance loss locally in a given defect cluster for a fixed combination of ancilla repurposings. The latter allows us to determine which local strategies are optimal when using the heuristics in Appendix G.

Our method was validated with `stim` using the length of shortest graphlike error (i.e., using `shortest_graphlike_error()`) but requires less computing resources for large distances and high defect rates (resulting in a higher number of super-stabilizers) since we do not need to build a quantum circuit.

APPENDIX F: DEFECT CLUSTERING

First we virtually extend the defective processor (with two additional rows and columns of data qubits and ancilla qubits on each side) such that we treat boundary and bulk defects equally.

Although at this step we do not distinguish between defects in the bulk and near the boundary, we do handle two types of defects specially. One is the defective padding ancillas that are on the processor but not part of the original code (see Fig. 13), and the other is the defective links that connect the original code to the padding ancillas. These components are not part of the original code. In both cases we ignore these defective components when defining the defect clusters in the extended patch, but will also avoid using these defective or disconnected padding ancillas for any repurposing strategy.

Second we define the defect clusters. To guarantee that each hole is contained in one cluster, we consider the worst-case scenario where no repurposed check is

implemented. This means that we consider only the DQD strategies in Fig. 2. We build a graph where all the defective ancillas are connected to their neighboring data qubits, and all these data qubits as well as the defective data qubits are connected to their neighboring ancilla qubits. Finally, we connect all the ancilla qubits in the graph to their neighboring data qubits such as to reproduce the DQD checks in Figs. 2(a) and 2(b). The defect clusters are defined as connected components of this graph.

These clusters however often need to be recursively grown due to weight-1 checks (whose data qubits need to be disabled), an effect also known as the "avalanche effect" [15]. We remark that DQD holes are always rectangular in the rotated surface code since any concavity would result in weight-1 checks. This implies that each defect cluster can potentially cover a large area of the processor and we could ultimately find multiple small-size holes with our repurposing strategies within that area. Nonetheless, this is the only way to guarantee that strategies in individual clusters will never overlap.

To limit the growth of the defect clusters, we remove nodes on the perimeter of the graph that we are certain will never be part of gauge checks because there is not any first or second neighbor defect around those nodes. We refer the reader to our repository [19] for further details.

APPENDIX G: REDUCING THE COMPUTATIONAL COSTS

Here we discuss how to reduce the requirement of computational resources when we find optimal strategies for each defect cluster, without significantly lowering the quality of the algorithm's output patches. The desired balance between solution quality and search space size varies by problem, so we introduce multiple tunable parameters to enable a flexible trade-off.

1. Sorting all the combinations of repurposings

For a given defect cluster, the search for local strategies is implemented with a nested `for` loop. First we loop over each possible link assignment: whether we use ancilla repurposing or the data defect primitive for each link defect. Second, we loop over all possible combinations of repurposing strategies for ancilla and link defects. In order to avoid evaluating all these combinations, we want to break early from the nested loops. Then to avoid missing the locally optimal strategies, we need to sort the trial combinations so that the promising ones come first.

First, we sort the link assignments that we iterate through in the outer loop. Since ancilla repurposing is usually the better approach for handling link defects than the data defect primitive, we start by applying ancilla repurposing on every link defect. After each iteration of the outer loop, we try the data defect primitive on one of the link defects and so on. Eventually, we try the data defect primitive on all link defects.

Then we sort the combinations of repurposings for a fixed link assignment, which we go over in the inner loop. This is achieved by inspecting the two possible repurposings for each ancilla defect and link defect (that is not mapped to a disabled data qubit), and deciding which of them is more promising. Generally, we prefer the repurposing that is oriented to preserve the distance. However, not all repurposing checks survive after we remove the "snakes," so our decision also accounts for the likelihood that the repurposing checks survive. For each of the two possible repurposings around an ancilla or link defect, we check for defects on the qubits and links in the repurposing checks. This gives us an upper bound on the number of feasible weight-2 checks (more could be disabled later due to the chain effect). If one of the repurposings has fewer checks that are marked invalid, we mark it as the more promising one. If tied, we prefer the distance-preserving one.

As a result of the sorting in both layers, the strategy that we first evaluate likely implements the highest number of repurposed checks and the most distance-preserving checks. The strategies that immediately follow implement a repurposing with flipped orientation for one ancilla or link defect, and far down the list are the strategies with some link defects adapted with the data defect primitive. This sorting guarantees that we encounter the optimal strategies for the defect cluster early in the nested loops.

2. Parameters for the heuristics

Here we introduce what we call "heuristics," which are a set of conditions that allow us to either stop the search or skip some possibilities that are suboptimal. We detail the different parameters for the heuristics that we use for reducing the computational costs. Some of them define stopping conditions for the nested `for` loop that we discussed in the previous section, and others define filters that further reduce the number of local strategies that we keep for each cluster. This rarely compromises the code distance due to the sorting we perform.

After executing the nested loops, we can filter out some local strategies before moving on to the global stage. This reduces the number of global strategies that we build from the local ones, and hence reduces the memory and runtime of the algorithm at the cost of global optimization opportunities. This filter is defined by `n_sum`, `n_max`, and `n_sol_max_per_cluster`. The parameters `n_sum` and `n_max` set up a threshold for keeping a strategy, based on whether the sum (max) of the distance losses across the two dimensions is among the smallest `n_sum` (`n_max`) different values from all strategies for the defect cluster. The other parameter, `n_sol_max_per_cluster`, directly caps the number of strategies that we keep for each

defect cluster. When set to a small value, this parameter helps us dramatically reduce the computational costs. To balance the optimality and diversity of the local solutions we keep, we group the strategies by distance loss and select the ones with the most active qubits from each group.

When `n_sum` and/or `n_max` are used, we also apply them inside the nested loops, to determine whether to keep a strategy till the end of the loops.

We define stopping conditions for the nested loops with the parameters `link_defect_to_ancilla`, `n_zombie`, and `n_skip`. The first one, `link_defect_to_ancilla`, is a flag that restricts the search for local strategies to those that strictly use ancilla repurposing for link defects (i.e., we do not use the data defect primitive for link defects). In this case we only iterate through the outer loop once, and do not consider mapping link defects to disabled data qubits. The second one, `n_zombie`, provides a mechanism for breaking from the inner loop. It is triggered when we fail to find a better strategy (in terms of the number of functional data qubits) after evaluating `n_zombie` strategies for $\vec{\ell}$. The third one, `n_skip`, defines another condition for breaking from the inner loop. It is triggered if the last `n_skip` items that we iterated through for the current $\vec{\ell}$ were all discarded due to not meeting the standard set up `n_sum` and/or `n_max`.

Finally, before applying boundary deformation to different combinations of local strategies, we can cap the number of these combinations by the parameter `n_sol_max`.

3. Time complexity

We can divide the algorithm in two parts. The first part ignores any boundary deformation and focuses on finding the optimal strategies in a virtually extended grid. The second part finds the optimal strategy including the boundary deformation.

Without any heuristics and clustering, the time complexity is exponential in the number of defective ancilla qubits and defective two-qubit gates: $\mathcal{O}(2^a 3^l)$ where a is the number of defective ancilla qubits and l is the number of defective two-qubit gates. This is because SnL use two possible primitive operations for defective ancilla qubits [see Fig. 2(c)] and three for defective two-qubit gates [see Fig. 2(d) and 2(e)].

The first improvement for the time complexity is clustering the defects so we can parallelize the local deformations of the surface code patch. The time complexity then reduces to $\mathcal{O}(\max_i 2^{a_i} 3^{l_i})$ where a_i is the number of defective ancilla qubits and l_i is the number of defective two-qubit gates in the ith defect cluster.

Secondly, if we set `link_defect_to_ancilla` to `True` then we force SnL to use ancilla repurposing for all defective two-qubit gates. The complexity then reduces to $\mathcal{O}(\max_i 2^{a_i} 2^{l_i})$. To further reduce the time complexity we apply a pre-processing step. We reorder all possible $2^{a_i} 2^{l_i}$ ways of repurposing by favoring the combinations with the highest number of repurposed checks that are not compromised by other defective components, followed by those that are oriented so that the code distance is preserved if those defective components were isolated. We then iterate one at a time through these combinations. If after evaluating `n_zombie` combinations we do not find a strategy with less disabled qubits then we stop the search. Assuming that the ordering of the combinations is such that the number of disabled qubits is indeed minimized for the first few combinations then we expect the time complexity to be $\mathcal{O}$(`n_zombie`). However, for large defect clusters and large `n_zombie`, the sorting might be suboptimal and thus lead to a subexponential time complexity.

If we instead set `link_defect_to_ancilla` to `False`, then we repeat the above for each combination of defective two-qubit gates that are replaced by defective data qubits as in DQD [see Fig. 2(d)]. There, `n_skip` is used to break the loop, by looking at whether the distance improved or not over the last `n_skip` iterations. The time complexity is subexponential and depends on the actual defect configuration.

During the second part of the algorithm, we apply boundary deformation to each global strategy we found in the first part: there are n^c global strategies where c is the number of defect clusters and n is the number of strategies we keep per cluster. Here n is controlled by `n_sol_max_per_cluster`, `n_max`, and `n_sum`. Moreover, we can decide to keep only `n_sol_max` global strategies with the largest code distances in the virtually extended grid. We then apply the boundary deformation routine (which has low time complexity) to each global strategy that we kept. With a finite `n_sol_max` the time complexity at this step can be constant, but otherwise is subexponential and depends on `n_sol_max_per_cluster`, `n_max`, and `n_sum`.

Overall the time complexity goes from exponential to subexponential with heuristics. If `link_defect_to_ancilla` = `True` and `n_zombie` = `n_max` = `n_sum` = `n_sol_max_per_cluster` = `n_sol_max` = 1 then the time complexity is the same as DQD. Changing any of these parameters can ultimately result in a subexponential complexity. Table II in Appendix H shows seven choices of heuristics ordered with increasing time complexity, with option No. 0 having the same complexity as DQD and No. 6 being exponential in the number of defective components.

4. Limitations

It is more costly to search for optimal strategies for bigger defect clusters, since the number of possible solutions is exponential in the number of ancilla and link defects. After we apply the heuristics introduced above, we observe that the runtime and memory overhead of the algorithm

TABLE II. Hierarchy of heuristics that was used for the sampling of $d = 13 - 17$. The "· · ·" marks indicate that no value was set for this parameter, i.e., this condition was not used.

Option/#	0	1	2	3	4	5	6
`link_defect_to_ancilla`	True	True	True	False	False	False	False
`n_skip`	· · ·	· · ·	· · ·	10	10	· · ·	· · ·
`n_zombie`	1	10	· · ·	· · ·	· · ·	· · ·	· · ·
`n_max`	1	2	· · ·	· · ·	· · ·	· · ·	· · ·
`n_sum`	1	2	· · ·	· · ·	· · ·	· · ·	· · ·
`n_sol_max_per_cluster`	1	2	2	3	3	10	· · ·
`n_sol_max`	1	2	10	10	· · ·	100	· · ·

still strongly depend on the size of the largest defect cluster in the patch.

APPENDIX H: HEURISTICS USED FOR SAMPLING

Here we discuss the performance of different heuristics we tested when sampling target distances 3–17 with a 1% defect rate. We refer the reader to Appendix G for the different parameters used in the heuristics. The hierarchy of heuristics is shown in Table II. We ran the patches with $d = 21 - 69$ using the settings labeled 0, but modified to use `n_zombie` $= 2$.

Heuristics $0 - 2$ ignore the data defect primitive for link defects, therefore reducing the number of strategies to consider per link defect from 3 to 2. Heuristic 0 prevents any exponential blow-up in memory and runtime by allowing a single optimal strategy per cluster. Runtime is also limited by keeping only the first valid strategy found after storing all the possible combinations of repurposings. Heuristic 1 allows for more exploration with `n_zombie` $= 10$ and keeps up to two strategies per defect cluster with a maximum of two unique max and sum distance loss values. We keep a maximum of two global strategies. This is leads to a smooth increase in memory and runtime. We now have a more global optimization of the code distance. Heuristic 2 lifts all constraints except for the number of strategies per cluster, kept at two, but allow for 10 global strategies. We allow for more optimization in terms of boundary deformation by having more global strategies to try. Heuristics $3 - 5$ allow for link defects to be mapped to data defects as well. Heuristics 3 and 4 but have a limited number of strategies per considered per link assignment ($n_{\texttt{skip}} = 10$) and allow for three strategies per cluster such as to still have global optimization. However, heuristic 3 allow for 10 global strategies whereas 4 is unconstrained. Finally, heuristic 6 is completely unconstrained, i.e., it looks for all possible global strategies.

When looking at the average output distance divided by the target distance (defined here as the distance ratio), and taking the best value from all prior heuristics and the current heuristic, we observe that the distance does increase smoothly with the heuristic number in Fig. 16 for target distances 7, 13, and 17 at 1% defect rate. In fact, more generally, heuristic 3 is generally sufficient for target code distances above 11. Indeed, a plateau can be observed for both 13 and 17. The main reason why this plateau is not present for distance 7 is because boundary deformation is more impactful at smaller code distances. For that reason, it is useful to consider a greater number of global strategies. Nonetheless, we can see that even for distance 7 the gain between heuristic 3 and 6 is about 0.1%. We also remark that the gain between heuristic 0 and 6 is on the order of only 0.3% for distance 7 (because isolated defects

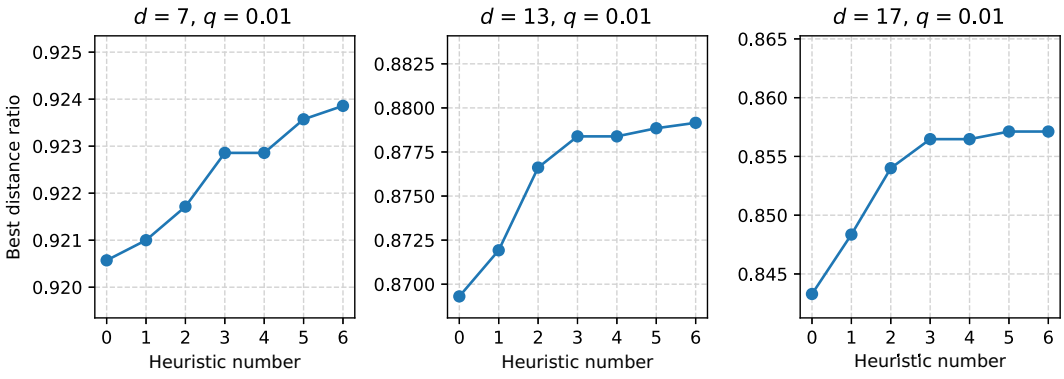


FIG. 16. Average output distance divided by the target distance (here defined as the distance ratio) as a function of the heuristic number (defined in Table II) for target distance 7, 13, and 17 at 1% defect rate.

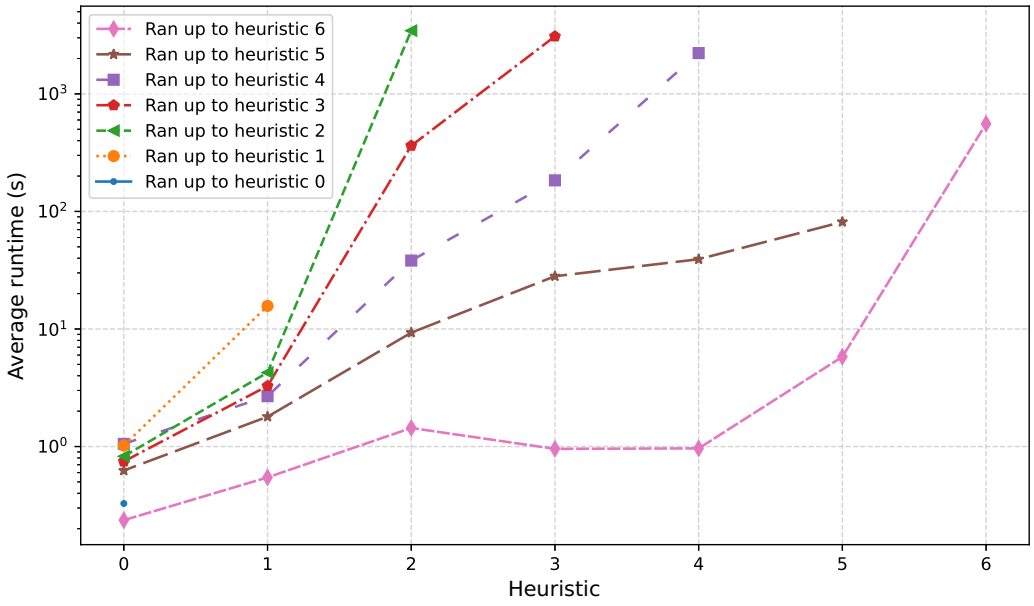


FIG. 17. Average runtime in seconds per heuristic defined in Table II. The sample defect configurations are divided such that each curve has all the samples that ran up to the last heuristic that is plotted within 8 hours. This is for target distances 3–17 and 1% defect rate.

are more common) but otherwise 1.0–1.5% for distance 13 and 17. This implies that by doing only local optimization in the defect clusters we can generally recover most of the target distance. This is particularly true at small defect rates where defects are more likely to be isolated.

We now move to an analysis of the runtime. It is particularly challenging to have a good estimate for the runtime. This is because it strongly depends on (1) the number of defect clusters, (2) the average size of the defect clusters, (3) the number of ancilla and link defects, (3) the details of the sorting that we did in Appendix G 1, (4) the optimization of corner placement during the boundary deformation. A simple rule of thumb is that the larger the number of ancilla and link defects is per cluster, the more challenging the local optimization is and therefore, the longer the runtime. In Fig. 17 we plot the average runtime for distance 13 as a function of the heuristic number. Here we divided the data into seven lines: the nth line has all the random defect samples that we could run up to heuristic n under 8 hours. First we remark that for all random samples, heuristic 0 is fast (i.e., on the order of 1–10 s). Similarly, heuristic 1 is generally guaranteed to run under 11 min. Heuristic 2 then shows a steep increase in runtime for some of the configurations. This is most likely a direct results of a lack of any stopping condition during the local optimization in each cluster. Heuristic 3 now also considers mapping link defects to data defects, which yields more strategies, however, it includes a stopping condition $n_{\texttt{skip}}$ for the local optimization (which explains the small drop in runtime for the longest line). However, for most of the lines, having more strategies per link defect increases the runtime, especially in the presence of a significant number of link defects. Heuristic 4 now allows for more global strategies that each require to run the boundary deformation part of the algorithm, which can become costly with corner holes. Generally, if heuristic 4 can be run, heuristics 5 and 6 also can. We however decided to not run the next heuristic if the current heuristic took more than 20 min to run.

We remark that the standard variation is too high in Fig. 17 to be of any use or even shown. The main takeaway messages are that the larger the defect clusters, the more the ancilla and link defects, and the more corner holes the harder it is to run the algorithm without heuristics. It is most probable that the heuristics in Appendix G could be further refined to offer a better predictability for the runtime. It should however be noted that constraining the number of global strategies already strongly constrains memory usage.

Furthermore, from Figs. 17 and 7, it becomes clear that heuristic 0 (the strongest heuristic that only uses local optimization) is already sufficient and results in a high plateau value of 0.79 at large distances. Even at distances smaller than 11, using no heuristic would only improve by approximately 0.1% on average. For moderate distances we are talking about approximately 1–2% improvement. From the plateau in Fig. 7(a), we can still expect only a few percents gain at larger distances. Finally, we also find that adding `n_skip` to heuristic 1 and 2 can strongly help reduce the runtime with compromising the performance of the algorithm. We however leave a more detailed analysis of the heuristics proposed here for future work.

APPENDIX I: ADDITIONAL NUMERICAL RESULTS

In this appendix, we report additional numerical results from sampling done in Secs. IV A and IV B.

1. Code distance

Here we focus on the code distances that were obtained in Sec. IV A. In Fig. 18(a) we report the average output distance d_{out} as a function of the target distance d_{targ} for both SnL and DQD, at $q = 0.001$ and $q = 0.01$ defect rates.

In Fig. 18(b), we plot the correlation between the distances outputted by SnL and DQD. For defect configurations sampled at different values of q and d_{targ}, the SnL algorithm consistently outperforms DQD, and the ratio between the two outputted distances remain nearly constant when q is fixed and d_{targ} is varied.

We have thus far focused on the case where the probabilities of the three types of defects, data, ancilla, and link are all equal. Let q_d, q_a, and q_l be the data, ancilla, and link defect rates, respectively. We also studied the case where we have only one type of defect and the other two types of defects have zero probability, e.g., $d_q = 0.01, d_a = d_l = 0$. We sampled 1000 random defect configurations for each distance and applied both our SnL algorithm and DQD to each configuration. Figure 19 shows the average distance relative to the designed distance with different ratios of defect rates. The non-zero defect rates are set to 1%. With data, ancilla, and link defects at equal rate, we see that SnL outperforms DQD increasingly at larger distance. This reflects how DQD performs particularly badly for large defect clusters, which become more common as we increase the code patch size. With only data, link, or ancilla defects, we see greater relative distance for both strategies compared with $q_d = q_a = q_l = 0.01$. This is expected because we have few defects in total. Note that with only data defects, SnL and DQD are identical approaches. The distance loss in this case ($q_d = 0.01, q_a = q_l$) is close to that of SnL with $q_d = q_a = q_l = 0.01$. This demonstrates how the presence of ancilla and link defects has relatively little effect on the average distance with SnL. With only link or only ancilla defects, SnL recovers nearly the full distance, while DQD gives a significant drop in distance.

2. Logical performance

We now move on to the analysis of the logical performance (see Table I for the threshold values). Here we consider two noise models: a standard depolarizing noise model and SI1000 [22].

a. Standard depolarizing noise model

The threshold plots for SnL (our approach), DQD (baseline), and the defect-free case using the standard depolarizing noise model are shown in Fig. 20. The simulations are run for $2d_{\text{targ}}$ rounds. Each point is the average logical error rate (between X and Z) on a sample of 1000 patches with 1% defect rate.

We plot the distributions of logical error rate per round with a series of histograms in Fig. 22. We took 100 random defect configurations for the $d = 11$ surface code with 1% probability for data, ancilla, and link defects. We observe a lower mean, a lower minimum, and a narrower distribution of logical error rates for SnL, relative to DQD, as expected from the width of the histograms of the output

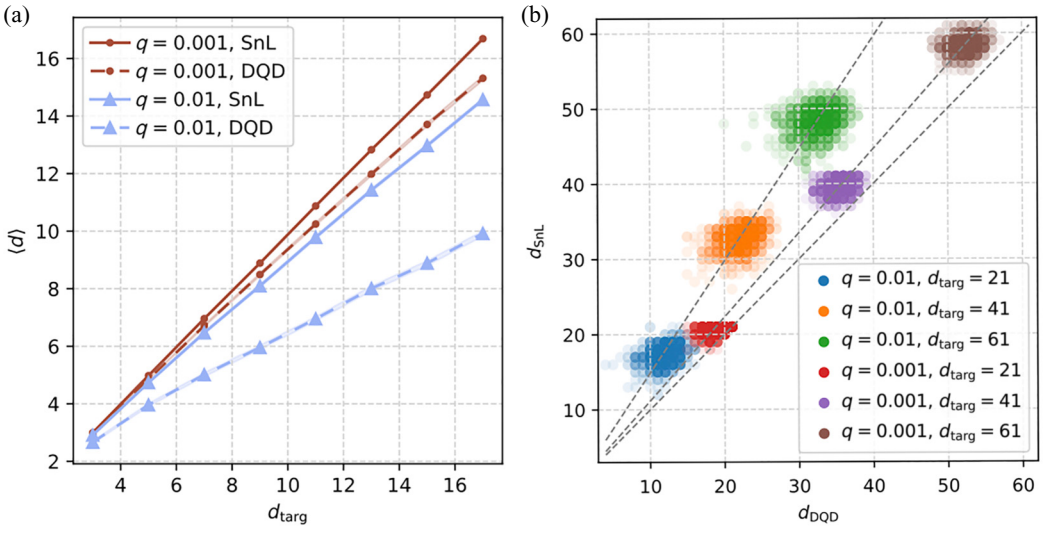


FIG. 18. (a) Average d_{out} for d_{targ} ranging from 3 to 17. (b) Correlations between the code distances outputted by SnL and DQD. We see that for all defect configurations SnL outperforms DQD. The dashed lines have slopes of 1, 1.12, and 1.49, where the latter two are based on the large d_{targ} plateau values from Fig. 7.

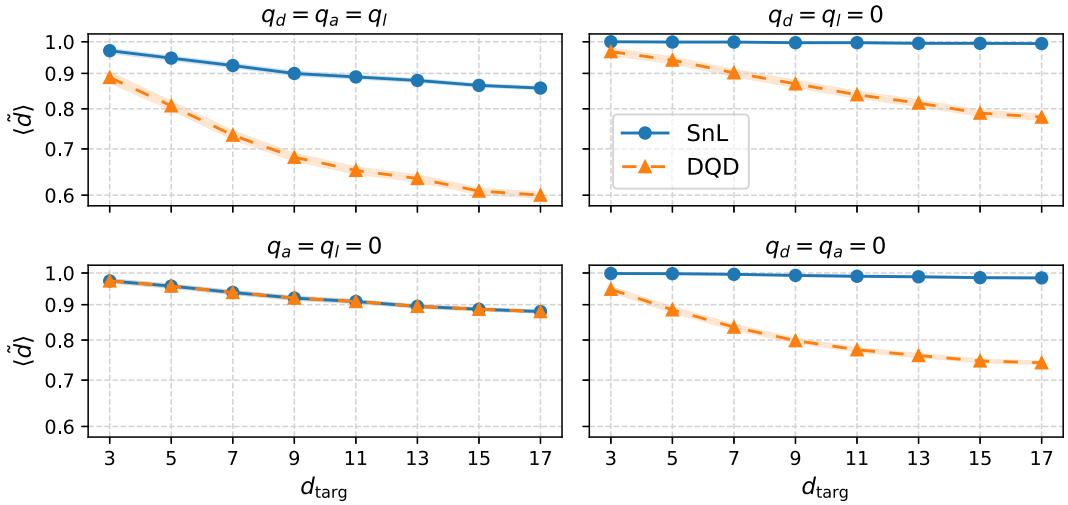


FIG. 19. The average code distance for SnL and DQD for code distances from 3 to 17. Each plot shows a different ratio of defect probabilities for the three types of defects, data, ancilla, and link. In each case, the non-zero probabilities are set to 1%. Notice that when we have only data defects, the two approaches are identical. SnL offers the greatest gains when we have link and ancilla defects in the code patch. Each average is composed of 1000 randomly sampled defect configurations.

code distances in Figs. 7 and 18. We also see that the distribution for SnL is skewed towards the lower error rates compared to the DQD distribution that is roughly symmetric. The shape of the distributions and the relationship between SnL and DQD is consistent across the different values of the physica error rate, p.

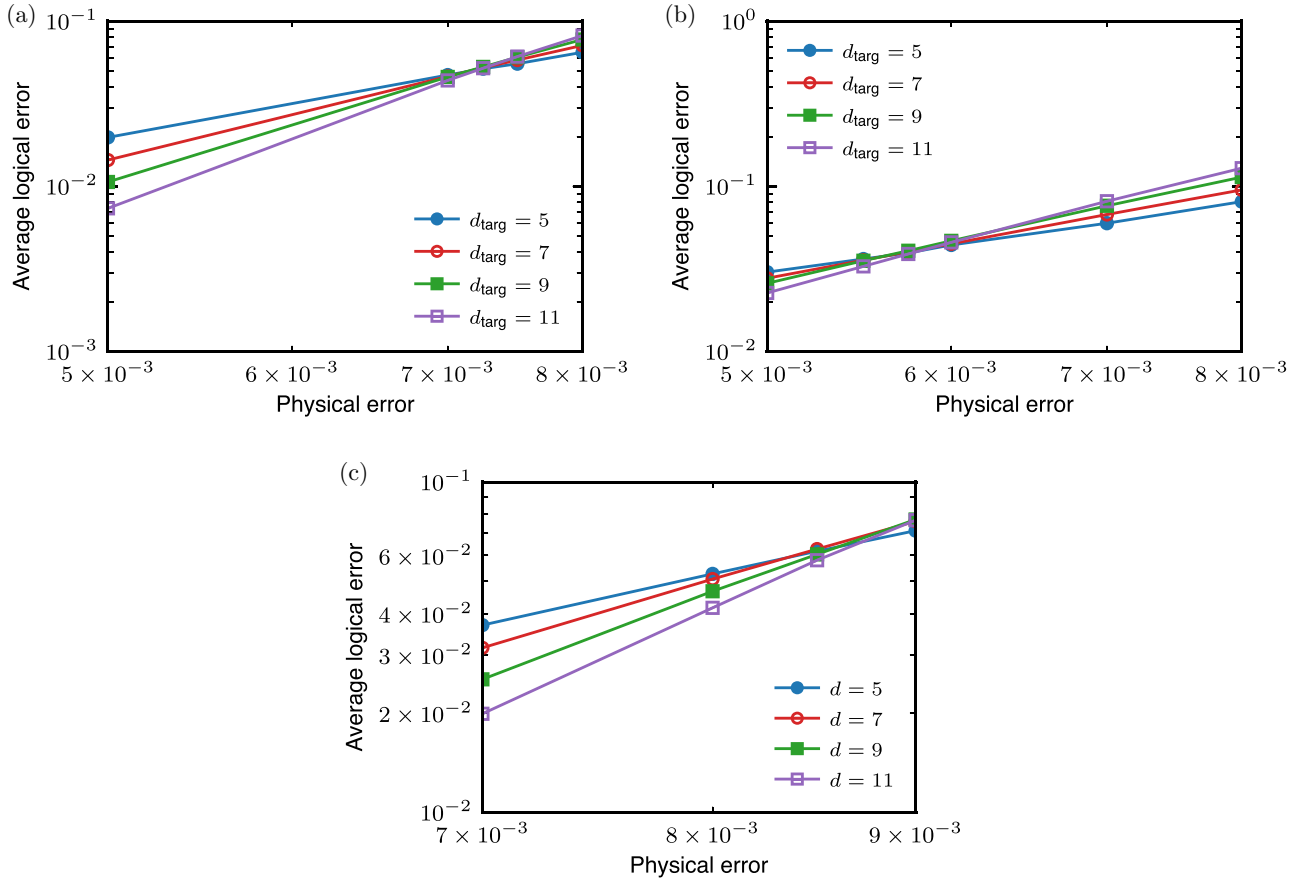


FIG. 20. Threshold plots with the standard noise model.

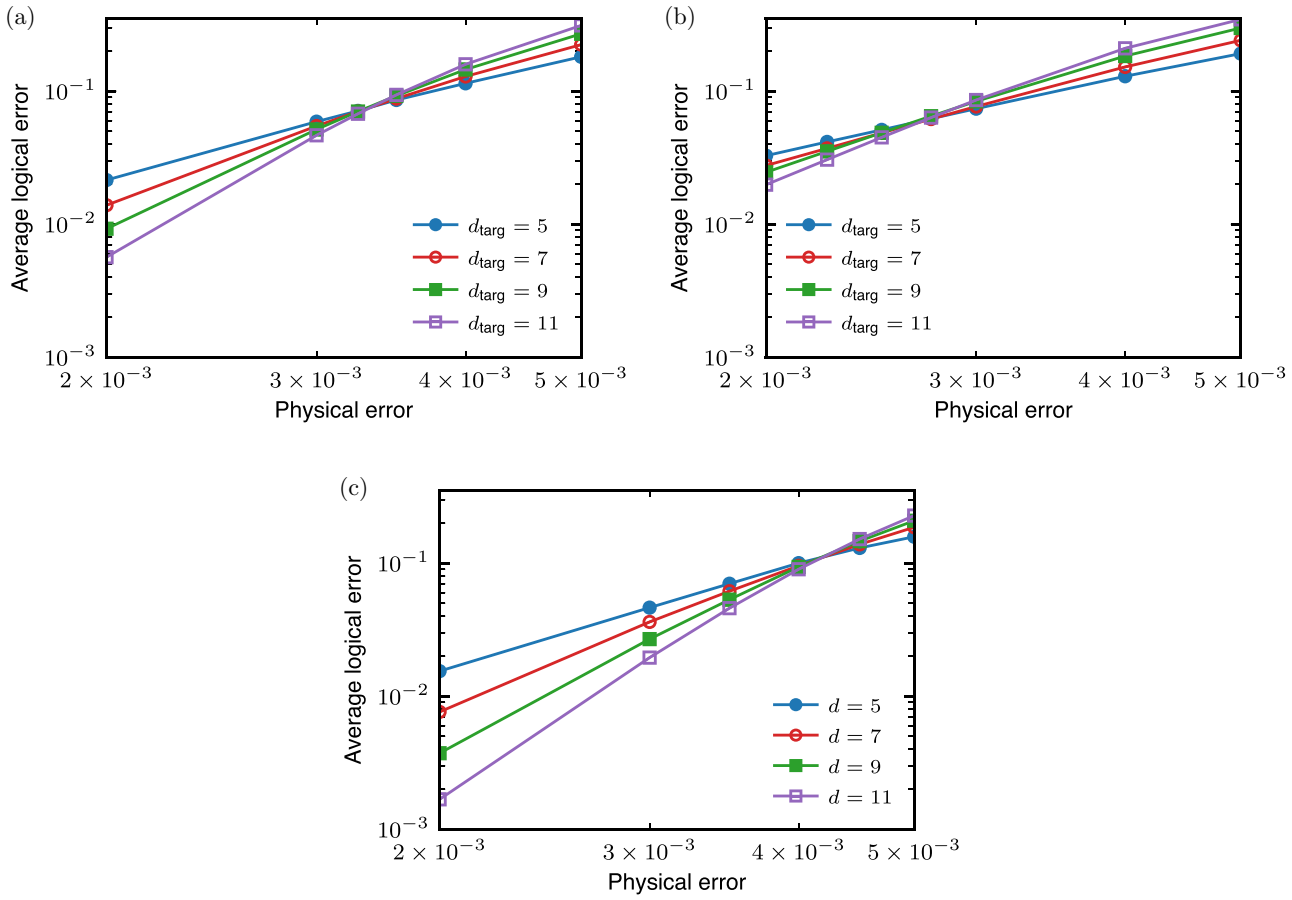


FIG. 21. Threshold plots with the SI1000 noise model.

b. SI1000 noise model

We use the same methodology as for the standard noise model but now consider the SI1000 noise model, a more realistic noise model for superconducting circuits. The resulting threshold plots are shown in Fig. 21.

An important difference between the SI1000 noise model and the standard noise model, beyond measurement and reset errors, is the presence of noise during idling. Requiring more measurement rounds to measure all super-stabilizers also leads to more idling errors.

APPENDIX J: CNOT GATE

In this appendix, we explain the methodology behind Fig. 11.

1. Estimate of the infidelity

We now discuss how we estimate the CNOT gate infidelity shown in Fig. 11(d). To avoid non-deterministic observables in `stim`, we divide the lattice surgery CNOT into four flows in a manner similar to the approach in Ref. [27]. Put simply, we consider the two merge-split measurements (control-ancilla and ancilla-target) separately, and for each of these we initialize and measure the patches both in the X or Z basis. To determine the errors in each flow, we measure the deterministic observables summarized in Table III. Taking the sum of the errors in all the deterministic observables in Table III, divided by two for the X and Z bases, approximately gives the infidelity of the CNOT gate.

We also expect this quantity to be close to the logical error rate for a memory experiment with a distance-$(3, 7)$ surface code over the same number of measurement rounds. This is because the performance of the merged patches mostly determines the infidelity of the CNOT gate in this case. This is confirmed in Fig. 23(d): the worst logical error rate for $10d$ and $5d$ measurement rounds (corresponding to the number of rounds for the defective and non-defective cases, respectively), with $d = 3$, match with the estimate for the CNOT infidelity for both the defective and non-defective cases in Fig. 11.

2. Yield experiment

We now explain how we set up the yield experiment in Fig. 11(e). We consider a $3d_{\text{targ}} \times 3d_{\text{targ}}$ processor such that three patches with code distance d_{targ} can fit side by side in

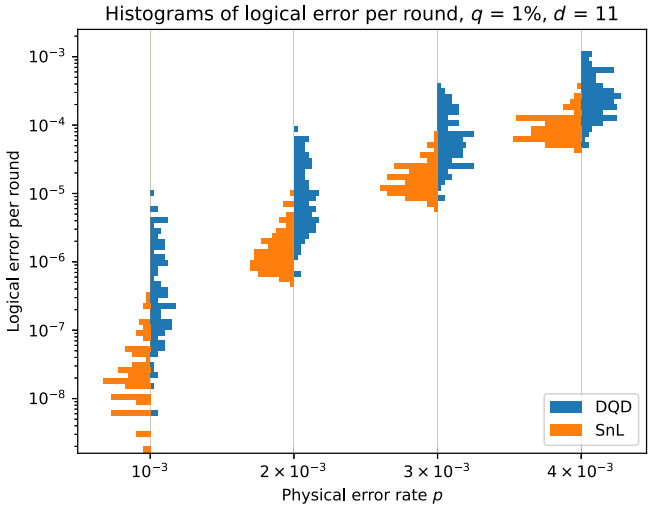


FIG. 22. Distributions of logical error rates per round for 100 random defect configurations sampled with a 1% probability for each of data, ancilla, and link defects using a standard depolarizing noise model with noise strength p. The initial surface code patches were $d = 11$, and we simulated memory in the X and Z basis for 44 rounds before extracting a per-round error rate for the average of X and Z. We gathered 2 million shots for each basis for each point, except for the points with physical error rate $p = 10^{-3}$, for which we gathered 4 million shots for each basis. For the SnL histogram with $p = 10^{-3}$ there were two samples that gave no logical errors after 8 million total shots. These are plotted in the lowest bin.

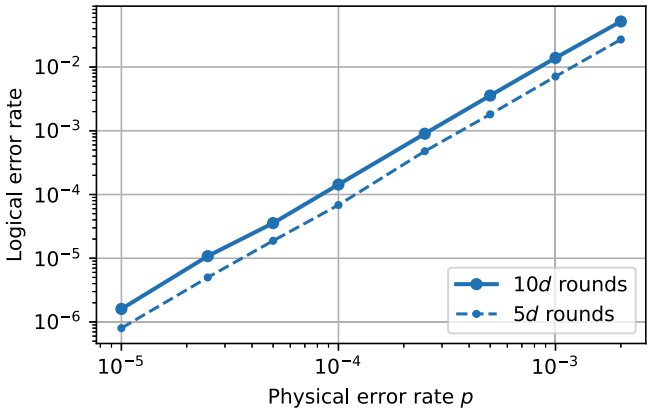


FIG. 23. Logical error rate (in the shortest and horizontal direction) for the distance-$(3, 7)$ defect-free surface code over $10d$ and $5d$ measurement rounds, for $d = 3$, corresponding to the number of measurement rounds for the CNOT gate in Figs. 11(a)–11(d).

both directions. We position the control, ancilla and target patches as in Fig. 11(a): the control patch is at the top left corner, the ancilla patch is at the bottom left corner, and the target patch is at the bottom right corner. The merged patches are defined on the left and bottom halves of the processor as shown in panels (b) and (c).

In this experiment, we ultimately want three distance d_{targ} patches to operate so that they realize the CNOT gate. However, a random defect configuration generally results in distance loss. To recover the logical performance of the individual patches, we might want to extend them into the routing space such that their distance is d_{targ}. There is however a limit to how much we can grow the three patches into the available routing space. The goal of this experiment is to determine how often can we recover three distance d_{targ} patches in a $3d_{\text{targ}} \times 3d_{\text{targ}}$ processor to realize an operation such as the CNOT gate. This experiment fails when the three resulting patches overlap (i.e., if there is not at least one row and column of data qubits separating the patches). The failure rate is shown in Fig. 11(d) as a function of the defect rate, for different target distances.

We remark that we always find the optimal strategy for the entire processor first, and then define subpatches with optimal boundary deformation, as it was done for Figs. 11(a)–11(c). First we define a target window for the entire processor and find the strategy yielding the best code distance overall. We then define three rectangular target windows for the subpatches: w_C, w_A, and w_T for the control, ancilla, and target qubits, respectively. w_C, w_A, w_T has its top left, bottom left, bottom right corner fixed at the top left, bottom left, bottom right corner of the processor's window. They each start with the minimum area such that they and contain ideal d_{targ}-surface code patch. We iteratively grow them in both directions (adding one row and column of data and ancilla qubits to the subpatch's target window at a time) until the surface code patch contained in the windows have minimum distance d_{targ}. We stop and declare a failure as soon as there is no longer a row and column of data qubits left between the patches since routing would then be impossible.

Using DQD always leads to bigger distance loss, and consequently, requires digging more into the routing space. It is not surprising that the failure rate is significantly higher for DQD than for SnL. What is however interesting is that the failure rate grows quickly with the target distance for DQD. Intuitively, this is because more defective components are found in bigger codes and the holes end up being generally bigger. This implies that having a fixed

TABLE III. Deterministic observables for the four flows. Here X_i means the X operator on the qubit i, where $i = C$ (control), A (ancilla), or T (target). Moreover, $X_{i\oplus j}$ refers to the X operator of the patch resulting from merging patches i and j. Alternatively, $X_{i\oplus j}$ corresponds to X_iX_j after applying a Z correction on X_j depending on the measurement outcome of the X string in the routing space during the split [16].

Merge-split experiment	Z init/meas basis	X init/meas basis
Control-ancilla (ZZ)	Z_CZ_A, Z_C, Z_A	$X_{C\oplus A}$
Ancilla-target (XX)	$Z_{A\oplus T}$	X_AX_T, X_A, X_T

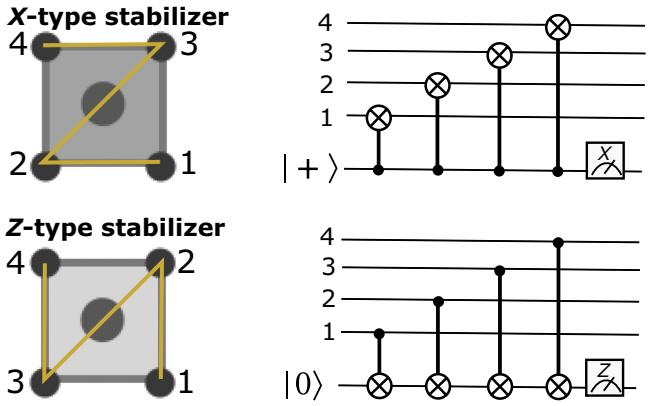


FIG. 24. Syndrome extraction schedule for X- and Z-type stabilizers.

$3d_{\text{targ}} \times 3d_{\text{targ}}$ processor is generally not sufficient to have a successful experiment. SnL, on the other hand, shows a near constant failure rate with increasing distance. This results from having smaller holes and distance-preserving strategies. This analysis could be extended to larger code distances with SnL and to different processor sizes. The bigger the processor size for fixed d_{targ} is, the smaller the failure rate.

APPENDIX K: SYNDROME EXTRACTION SCHEDULE

The syndrome extraction extraction schedule we use for the numerical simulations is shown in Fig. 24. We remark that flipping the X and Z boundaries of the rotated surface code in Fig. 1(b) also swaps the schedules of X- and Z-type stabilizers.

[1] A. Kitaev, Fault-tolerant quantum computation by anyons, Ann. Phys. (N. Y.) **303**, 2 (2003).

[2] S. B. Bravyi and A. Y. Kitaev, Quantum codes on a lattice with boundary, ArXiv:quant-ph/9811052.

[3] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, Topological quantum memory, J. Math. Phys. **43**, 4452 (2002).

[4] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, Surface codes: Towards practical large-scale quantum computation, Phys. Rev. A **86**, 032324 (2012).

[5] M. Kjaergaard, M. E. Schwartz, J. Braumüller, P. Krantz, J. I.-J. Wang, S. Gustavsson, and W. D. Oliver, Superconducting qubits: Current state of play, Annu. Rev. Condens. Matter Phys. **11**, 369 (2020).

[6] A. Bilmes, A. Megrant, P. Klimov, G. Weiss, J. M. Martinis, A. V. Ustinov, and J. Lisenfeld, Resolving the positions of defects in superconducting quantum bits, Sci. Rep. **10**, 3090 (2020).

[7] P. V. Klimov, *et al.*, Fluctuations of energy-relaxation times in superconducting qubits, Phys. Rev. Lett. **121**, 090502 (2018).

[8] J. Burnett, L. Faoro, I. Wisby, V. L. Gurtovoi, A. V. Chernykh, G. M. Mikhailov, V. A. Tulin, R. Shaikhaidarov, V. Antonov, P. J. Meeson, A. Y. Tzalenchuk, and T. Lindström, Evidence for interacting two-level systems from the 1/f noise of a superconducting resonator, Nat. Commun. **5**, 4119 (2014).

[9] C. Müller, J. H. Cole, and J. Lisenfeld, Towards understanding two-level-systems in amorphous solids: Insights from quantum circuits, Rep. Progr. Phys. **82**, 124501 (2019).

[10] J. M. Auger, H. Anwar, M. Gimeno-Segovia, T. M. Stace, and D. E. Browne, Fault-tolerance thresholds for the surface code with fabrication errors, Phys. Rev. A **96**, 042316 (2017).

[11] A. Strikis, S. C. Benjamin, and B. J. Brown, Quantum computing is scalable on a planar array of qubits with fabrication defects, Phys. Rev. Appl. **19**, 064081 (2023).

[12] A. Siegel, A. Strikis, T. Flatters, and S. Benjamin, Adaptive surface code for quantum error correction in the presence of temporary or permanent defects, Quantum **7**, 1065 (2023).

[13] S. F. Lin, J. Viszlai, K. N. Smith, G. S. Ravi, C. Yuan, F. T. Chong, and B. J. Brown, in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24* (Association for Computing Machinery, New York, NY, USA, 2024), p. 216.

[14] K. Yin, X. Fang, Y. Shi, T. Humble, A. Li, and Y. Ding, Surf-deformer: Mitigating dynamic defects on surface code via adaptive deformation, ArXiv:2405.06941.

[15] Z. Wei, T. He, Y. Ye, D. Wu, Y. Zhang, Y. Zhao, W. Lin, H.-L. Huang, X. Zhu, and J.-W. Pan, Low-overhead defect-adaptive surface code with bandage-like super-stabilizers, ArXiv:2404.18644.

[16] D. Horsman, A. G. Fowler, S. Devitt, and R. V. Meter, Surface code quantum computing by lattice surgery, New J. Phys. **14**, 123011 (2012).

[17] D. Litinski and F. V. Oppen, Lattice surgery with a twist: Simplifying Clifford gates of surface codes, Quantum **2**, 62 (2018).

[18] https://en.wikipedia.org/wiki/Snakes_and_ladders.

[19] C. Leroux, S. F. Lin, P. Bienias, K. R. Sankar, A. Benhemou, A. Kubica, and J. K. Iverson, Snl: Adapting the surface code to defects, 2024, https://github.com/amazon-science/snakes_and_ladders_adapting_the_surface_code_to_defects.

[20] C. Gidney, Stim: A fast stabilizer circuit simulator, Quantum **5**, 497 (2021).

[21] O. Higgott and C. Gidney, Sparse blossom: Correcting a million errors per core second with minimum-weight matching, Quantum **9**, 1600 (2025).

[22] C. Gidney, M. Newman, A. Fowler, and M. Broughton, A fault-tolerant honeycomb memory, Quantum **5**, 605 (2021).

[23] Our numerical simulations for small codes do not establish a threshold theorem. We may require "shells" [11] for a guaranteed threshold.

[24] With two-qubit depolarizing noise on two-qubit gates, one-qubit depolarizing noise on one-qubit gates, and X/Z noise on reset and measurement, all with the same error rate p.

[25] We exclude defect configurations where our software fails to produce adapted patches for DQD. These excluded cases constitute $<1\%$ of the sampled cases for $d_{\text{targ}} = 5$ and are very rare for higher d_{targ}; they also arise in prior works.

[26] O. Higgott and N. P. Breuckmann, Subsystem codes with high thresholds by gauge fixing and reduced qubit overhead, Phys. Rev. X **11**, 031039 (2021).
[27] C. Gidney and D. Bacon, Less bacon more threshold, ArXiv:2305.12046.
[28] J. P. Bonilla Ataides, D. K. Tuckett, S. D. Bartlett, S. T. Flammia, and B. J. Brown, The XZZX surface code, Nat. Commun. **12**, 2172 (2021).
[29] Q. Xu, N. Mannucci, A. Seif, A. Kubica, S. T. Flammia, and L. Jiang, Tailored XZZX codes for biased noise, Phys. Rev. Res. **5**, 013035 (2023).
[30] A. Dua, A. Kubica, L. Jiang, S. T. Flammia, and M. J. Gullans, Clifford-deformed surface codes, PRX Quantum **5**, 010347 (2024).
[31] D. M. Debroy, M. McEwen, C. Gidney, N. Shutty, and A. Zalcman, Luci in the surface code with dropouts, ArXiv:2410.14891.