Covert Computation in the Abstract Tile-Assembly Model

Robert M. Alaniz
Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

David Caballero
Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Timothy Gomez
Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA

Elise Grizzell
Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Andrew Rodriguez
Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Robert Schweller
Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Tim Wylie
Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Abstract

There have been many advances in molecular computation that offer benefits such as targeted drug delivery, nanoscale mapping, and improved classification of nanoscale organisms. This power led to recent work exploring privacy in the computation, specifically, covert computation in self-assembling circuits. Here, we prove several important results related to the concept of a hidden computation in the most well-known model of self-assembly, the Abstract Tile-Assembly Model (aTAM). We show that in 2D, surprisingly, the model is capable of covert computation, but only with an exponential-sized assembly. We also show that the model is capable of covert computation with polynomial-sized assemblies with only one step in the third dimension (just-barely 3D). Finally, we investigate types of functions that can be covertly computed as members of P/Poly.

2012 ACM Subject Classification Theory of computation → Computational complexity and cryptography

Keywords and phrases self-assembly, covert computation, atam

Digital Object Identifier 10.4230/LIPIcs.SAND.2023.12

Funding This research was supported in part by National Science Foundation Grant CCF-1817602.

1 Introduction

With the ability to manufacture nanoscale structures and to use DNA as building blocks for structures [28] or for data storage [10], there has been a great increase in the need to process and compute information at the same level. Thus, the study of self-assembling computation has been an important and active area of research over the last two decades.

Designing self-assembling systems that compute functions is an active and well-studied area of computational geometry and biology [4,19]. This ability to craft monomers capable of placing themselves – especially when doing precision construction and computation at scales where conventional tools are incapable of operating, e.g., the nanoscale – has tremendous power. One of the few downsides to self-assembly computation is that the entire history of the computation is visible. In certain cases, this may be undesirable for privacy or security reasons, which we motivate below. Thus, we build on recent work [6,7,9] to explore *covert computation*, where we build Tile Assembly Computers (TACs) designed with the goal of

obtaining the output of computation while obscuring the inputs and computational history. We do this by proving that covert computation is possible even in one of the simplest standard models of self-assembly: the Abstract Tile-Assembly Model (aTAM) [29].

Motivation. The development of covert computation as a model and method of designing self-assembling systems was driven by several areas of concern in cryptography, biomedical engineering, privacy, and might even help protect intellectual property in systems that use "products of nature," such as DNA, as they cannot be patented in the United States as of 2013 [14]. Covert computation has also emerged as a powerful complexity tool, being used to show the coNP-completeness of the *Unique Assembly Verification* problem in the negative glue aTAM [9], and the PSPACE-completeness of the Unique Assembly Verification problem in the Staged Assembly Model [6]. As this paper focuses on systems without detachment, there might also be important applications in implantable systems where even the possibility of displacement from free-floating DNA could cause unknown side effects or destabilization of the assembly [23].

1.1 Previous Work

The Abstract Tile-Assembly Model (aTAM) was first introduced in [29] and inherited the ability to perform Turing computation from Wang tiles. Since then, investigation into the model has led in many directions, such as Intrinsic Universality [18, 22], efficient assembly of shapes [25], and parallel computation [5, 24]. Many generalizations have also appeared, such as allowing for RNA tiles that can be deleted [1, 15], multiple stages of growth [6, 12, 16], and even negative glues [9, 17]. The aTAM is powerful because not only can the tile set store information, but work has also gone into using the seed [3], or even the temperature [11, 26], for making systems more complex.

Tile Assembly Computers were defined in [5, 24], and Covert Computation, as defined in the field of self-assembly, was first introduced in 2019 [9] for negative growth-only aTAM. In *negative* variations of tile self-assembly models, tiles are capable of not only attachment to but also detachment from an assembly if the remaining assembly is still stable. In negative *growth-only* aTAM, tiles are never allowed to detach even though there may be glues providing a repellent force, and the system must be designed so that detachment does not occur. This paper introduced the covert construction framework to answer an open complexity problem for Unique Assembly Verification (UAV) with negative growth-only glues in the aTAM model, showing it to be coNP-complete. Notably, without negative glues, the UAV problem is solvable in polynomial time [2].

Covert computation has been explored in two other models of self-assembly as well: Staged Self-Assembly [6] and Tile Automata [7]. The staged self-assembly model, one of the most powerful *passive* tile self-assembly models, abstracts the process of scientists mixing test tubes together by allowing multiple self-assembly processes occurring in separate "bins" that may be combined in subsequent "stages". The authors show that 3-stages suffice for covert computation and used the techniques to show that the UAV problem directly relates the number of stages to a specific level of the polynomial hierarchy. Thus, with no restrictions on the number of stages, UAV in the staged model is PSPACE-complete. Covert Computation in the *active* self-assembly model of Tile Automata was shown to be rather simple as tiles in the model are capable of changing states (instead of having static glues), easily erasing computational history.

Table 1 Known Covert Circuits for n-bit function $f(x)$. Let MCS be the minimum circuit size that computes $f(x)$. **Input Size** is the size of the input assembly. **Output Size** is the size of the output template, where we use k to describe the number of output bits. * currently only works for binary functions.

Class	Model	Size Of				Ref
		Input	Tile Set	Output	Assembly	
Bool. Circuits	Neg_{GO}	$\mathcal{O}(n)$	$\mathcal{O}(MCS)$	$\mathcal{O}(k)$	$\mathcal{O}(MCS)$	[9]
Bool. Circuits	3D	$\mathcal{O}(n)$	$\mathcal{O}(MCS)$	$\mathcal{O}(k)$	$\mathcal{O}(MCS)$	Thm. 1
Rev. Circuits*	2D	$\mathcal{O}(n+MSC)$	$\mathcal{O}(MCS)$	$\mathcal{O}(1)$	$\mathcal{O}(2^n)$	Thm. 2

1.2 Our Contributions

In this work, we further explore the problem of designing covert tile assembly computers (TACs) in the aTAM, focusing on TACs that have a polynomial size description. We provide two new covert computers in the aTAM with only positive glue strengths of $\{1, 2\}$ in Sections 3 and 4. The 3D construction uses a similar technique to the circuits in [9] by implementing a NAND gate using dual rail logic and backfilling. We refer to this covert TAC as having a strict polynomial size since the systems defined by the TAC all produce assemblies of polynomial size. This only uses a single-step into the third dimension, which is occasionally referred to as *just-barely* 3D [20, 21].

The covert TAC in Section 4 is in the standard 2D aTAM. The TAC is of polynomial size, but produces an exponential-size terminal assembly. This works by computing the function non-covertly using Toffoli gates, getting the output, reversing the computation to recover the input, then building the next and previous circuit assemblies until all possible circuits are built. We utilize the Toffoli gates' reversibility property to have a symmetrical circuit assembly that displays its input on both sides that we can increment or decrement (the input used) to start the next computation.

In Section 5 we explore the classes of decision problems solvable by polynomial size covert TACs. Table 1 gives an overview of known covert circuits for functions based on the input size. Since covert has been defined as a non-uniform model, meaning different input sizes have different tile sets, we look at non-uniform complexity classes as well. Namely, the class P/poly, the class of problems solvable by polynomial size circuits. We prove that if a problem is solvable by a 3D covert TAC, then it is in P/poly. This, taken with the result in Section 3, shows an equivalence between these two models of computation.

2 Definitions

We begin with an overview of the Abstract Tile-Assembly Model, then follow with a definition of Tile Assembly Computers and covert computation.

2.1 Abstract Tile Assembly Model

At a high level, the Abstract Tile-Assembly Model (aTAM) uses a set of *tiles* capable of sticking together to construct shapes. These tiles are typically squares (2D) or cubes (3D) with *glues* on each side where they may attach to one another. A glue is labeled to indicate its type, governing what other tiles it may bond with and the *strength* of the bond. A tile with all of its labels is a *tile type*. A *tile set* contains all the tile types of the system. A single tile may attach at a location if the combined strength of the matching glues is greater than

or equal to the *temperature* τ. An *assembly* is a shape made up of one or more combined tiles. Construction is started around a designated *seed* assembly S. Any assembly capable of being made from the seed is called a *producible* assembly. An assembly is *terminal* if no more tiles can attach. A terminal assembly is said to be *uniquely produced* if it is the only terminal assembly that can be made by a tile system. A tile system is formally represented as an ordered triplet $\Gamma = (T, s, \tau)$ of the tile set, seed assembly, and temperature parameter, respectively.

2.1.1 aTAM Formal Definitions

Tiles. Let Π be an alphabet of symbols called the *glue types*. A tile is a finite edge polygon with some finite subset of border points, each assigned a glue type from Π. Each glue type $g \in \Pi$ also has some integer strength $str(g)$. Here, we consider unit square tiles of the same orientation with at most one glue type per face, and the *location* to be the center of the tile located at integer coordinates.

Assemblies. An assembly A is a finite set of tiles whose interiors do not overlap. If each tile in A is a translation of some tile in a set of tiles T, we say that A is an assembly over tile set T. For a given assembly A, define the *bond graph* G_A to be the weighted graph in which each element of A is a vertex, and the weight of an edge between two tiles is the strength of the overlapping matching glue points between the two tiles. Only overlapping glues of the same type contribute a non-zero weight, whereas overlapping, non-equal glues contribute zero weight to the bond graph. The property that only equal glue types interact with each other is referred to as the *diagonal glue function* property, and is perhaps more feasible than more general glue functions for experimental implementation (see [13] for the theoretical impact of relaxing this constraint). An assembly A is said to be τ*-stable* for an integer τ if the min-cut of G_A has weight at least τ.

Tile Attachment. Given a tile t, an integer τ, and an assembly A, we say that t may attach to A at temperature τ to form A' if there exists a translation t' of t such that $A' = A \cup \{t'\}$, and the sum of newly bonded glues between t' and A meets or exceeds τ. For a tile set T, we use notation $A \rightarrow_{T,\tau} A'$ to denote there exists some $t \in T$ that may attach to A to form A' at temperature τ. When T and τ are implied, we simply say $A \rightarrow A'$. Further, we say that $A \rightarrow^* A'$ if either $A = A'$, or there exists a finite sequence of assemblies $\langle A_1 \dots A_k \rangle$ such that $A \rightarrow A_1 \rightarrow \dots \rightarrow A_k \rightarrow A'$.

Tile Systems. A tile system $\Gamma = (T, S, \tau)$ is an ordered triplet consisting of a set of tiles T called the system's *tile set*, a τ-stable assembly S called the system's *seed* assembly, and a positive integer τ referred to as the system's *temperature*. A tile system $\Gamma = (T, S, \tau)$ has an associated set of *producible* assemblies, $\texttt{PROD}_\Gamma$, which define what assemblies can grow from the initial seed S by any sequence of temperature τ tile attachments from T. Formally, $S \in \texttt{PROD}_\Gamma$ is a base case producible assembly. Further, for every $A \in \texttt{PROD}_\Gamma$, if $A \rightarrow_{T,\tau} A'$, then $A' \in \texttt{PROD}_\Gamma$. That is, assembly S is producible, and for every producible assembly A, if A can grow into A', then A' is also producible.

We further denote a producible assembly A to be *terminal* if A has no attachable tile from T at temperature τ. We say a system $\Gamma = (T, S, \tau)$ *uniquely produces* an assembly A if all producible assemblies can grow into A through some sequence of tile attachments. More formally, Γ *uniquely produces* an assembly $A \in \texttt{PROD}_\Gamma$ if for every $A' \in \texttt{PROD}_\Gamma$ it is the case that $A' \rightarrow^* A$. Systems that uniquely produce one assembly are said to be *deterministic*.

2.2 Covert Computation

Here, we provide formal definitions for computing a function with a tile system and the further requirements for the covert computation of a function. Our formulation of computing functions is that used in [9], which is a modified version of the definition provided in [24] to allow for each bit to be represented by a subassembly potentially larger than a single tile.

Tile Assembly Computers (TAC). Informally, a Tile Assembly Computer (TAC) for a function f consists of a set of tiles, along with a format for both input and output. The input format is a specification for how to build an input seed to the system that encodes the desired input bit-string for function f. We require that each bit of the input be mapped to one of two assemblies for the respective bit position: a sub-assembly representing "0" or a sub-assembly representing "1". The input seed for the entire string is the union of all these sub-assemblies. This seed, along with the tile set of the TAC, forms a tile system. The output of the computation is the final terminal assembly this system builds. To interpret what bit-string is represented by the output, a second *output* format specifies a pair of sub-assemblies for each bit. The bit-string represented by the union of these subassemblies within the constructed assembly is the output of the system.

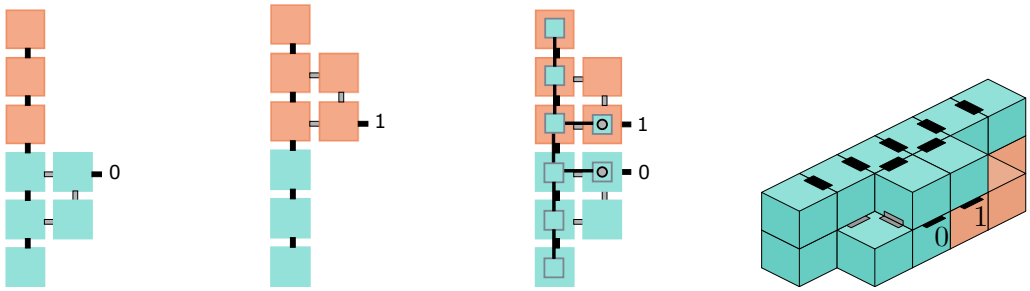
For a TAC to *covertly* compute f, the TAC must compute f and produce a unique assembly for each possible output of f. We note that our formulation for providing input and interpreting output is quite rigid and may prohibit more exotic forms of computation. Further, we caution that any formulation must take care to prevent "cheating" that could allow the output of a function to be partially or completely encoded within the input. To prevent this, a type of *uniformity* constraint, akin to what is considered in circuit complexity [27], should be enforced. We now provide the formal definitions of function computing and covert computation.

Input/Output Templates. An n-bit input/output template over tile set T is a sequence of ordered pairs of assemblies over T: $A = (A_{0,0}, A_{0,1}), \ldots, (A_{n-1,0}, A_{n-1,1})$. For a given n-bit string $b = b_0, \ldots, b_{n-1}$ and n-bit input/output template A, the *representation* of b with respect to A is the assembly $A(b) = \bigcup_i A_{i,b_i}$. A template is valid for a temperature τ if this union never contains overlaps for any choice of b and is always τ-stable. An assembly $B \supseteq A(b)$, which contains $A(b)$ as a subassembly, is said to represent b as long as $A(d) \nsubseteq B$ for any $d \neq b$. We refer to the size of a template as the size of the largest assembly defined by the template.

Function Computing Problem. A *tile assembly computer* (TAC) is an ordered quadruple $\Im = (T, I, O, \tau)$ where T is a tile set, I is an n-bit input template, and O is a k-bit output template. A TAC is said to compute function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2^k$ if for any $b \in \mathbb{Z}_2^n$ and $c \in \mathbb{Z}_2^k$ such that $f(b) = c$, then the tile system $\Gamma_{\Im,b} = (T, I(b), \tau)$ uniquely assembles a set of assemblies which all represent c with respect to template O.

Covert Computation. A TAC *covertly* computes a function $f(b) = c$ if 1) it computes f, and 2) for each c, there exists a unique assembly A_c such that for all b, where $f(b) = c$, the system $\Gamma_{\Im,b} = (T, I(b), \tau)$ uniquely produces A_c. In other words, A_c is determined by c, and every b where $f(b) = c$ has the exact same final assembly.

Polynomial-Sized Tile Assembly Computers. We say a TAC is polynomial size if the input template, tile set, and output template are all polynomial in n. However, this requirement still allows the producible assemblies to be exponentially larger. We say a TAC is *strictly* polynomial size if the produced assemblies are also polynomial in size.



(a) Bit Assembly Template 0. **(b)** Bit Assembly Template 1. **(c)** Bit Assembly Output Template. **(d)** Bit Assembly Output Template (isometric view).

Figure 1 Input assemblies and their respective input templates. The blue squares represent the bit set to zero, and the orange squares represent a bit set to one. Grey glues are strength-1, black glues are strength-2.

3 3-Dimensional Covert Circuits

In this section, we show how to perform covert computation in the aTAM using 3 dimensions. The computation behaves similarly to the covert circuit construction in [9] by building NAND gates and FANOUTs using dual rail logic. We start with showing a NOT that switches which wire is "on", then extending to a NAND by utilizing cooperative binding.

The main difference between the two constructions is when *backfilling* occurs, which is the process of filling in the unused dual rail line once that line is no longer needed. Here, we do not backfill as we go, rather, we fill in the assembly once the computation is complete.

3.1 Input Assemblies

Our input assembly consists of n 1×6 columns with two of four tiles attached on the right (Figures 1a and 1b). The top two tiles will be included when the input is 1, and the bottom two tiles if the input is 0. These tiles have enough attachment strength to be stable when both are present, however, since the tiles only have strength 1 bonds, they may not attach alone. This initially prevents the growth of the other bit, which is not placed until the computation is complete, further elaboration of this process is described in section 3.5.

3.2 Wires and NOT Gates

Bit information is represented and transferred using a wire. A wire is constructed using two rows of tiles (Figure 2a), each representing a binary value of 0 or 1. This dual rail system initially grows only one of the rows from the input assembly based off the input and then builds into the gates. Before the circuit finishes growing, only one row of each wire will be constructed, and at the end, the other wire row will be built.

Gates such as the NOT grow off the wires. An example of a NOT gate can be seen in Figure 2b, notice how we utilize the third dimension to cross the wires over each other. This gate swaps the position of the rows of tiles; a row that represents a 0 will now be in the upper row and represent a 1. At the end of each gate is a diode gadget that was used in previous work [9]. The gadget is a 2×2 subassembly that grows only in one direction. If the first tile is placed, the whole thing will be first. If the last tile is placed, nothing else grows since it connects using two strength 1 glues. This prevents errors caused by "backward" growth.

(a) Dual Rail Logic. **(b)** NOT Gate.

Figure 2 (a) We use dual rail gates. The input glue of 1 grows the orange tiles and 0 grows the blue. (b) A NOT gate is implemented by crossing the wires over each other.

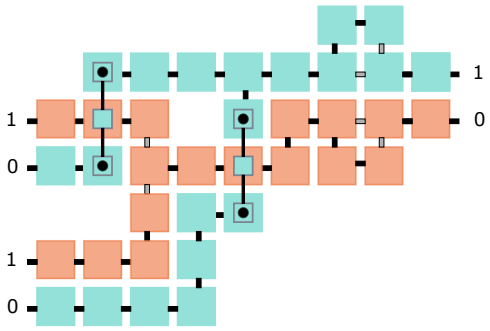


Figure 3 Full NAND Gate construction in the full circuit. The tiles in orange represent tiles that will be built from an input of 1 input, while the blue tiles come from an input of 0.

3.3 NAND Gates

We construct a NAND gate using the NOT gate and cooperative binding. The full NAND gate can be seen in Figure 3. If either input to a NAND gate is 0, the output is always 1. This can be seen in Figures 4a, 4b, and 4c. If any blue tile is placed, the 1 output of the gate will be built. If both inputs are 1, the 0 output can be constructed using cooperative binding.

One thing to note in the case of one output being 0 and the other being 1 is that the blue tiles will be placed along the other wire. However, this will not cause any issues since it can only build back up to the output of the previous gate due to the diode gadget.

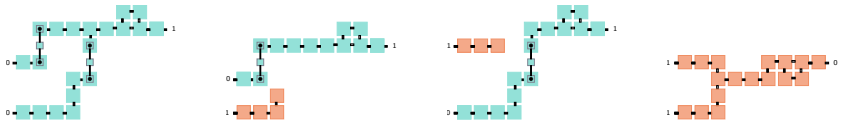
3.4 Fan Out and Crossover

Two other gadgets that assist in creating circuits are the fan out and crossover gadgets. The fan out (Figure 5a) splits a wire in order to copy the value to two gates. It does this by having each tile path split, and then use the third dimension to swap the positions.

The crossover gadget (Figure 5c) allows for the creation of non-planar circuits. Using the third dimension, a wire can go over another wire in order to reach its input. While such 3D crossovers simplify constructions greatly, we note that such crossovers are not necessarily needed, as planar circuits can simulate such crossovers using XOR gates [9].

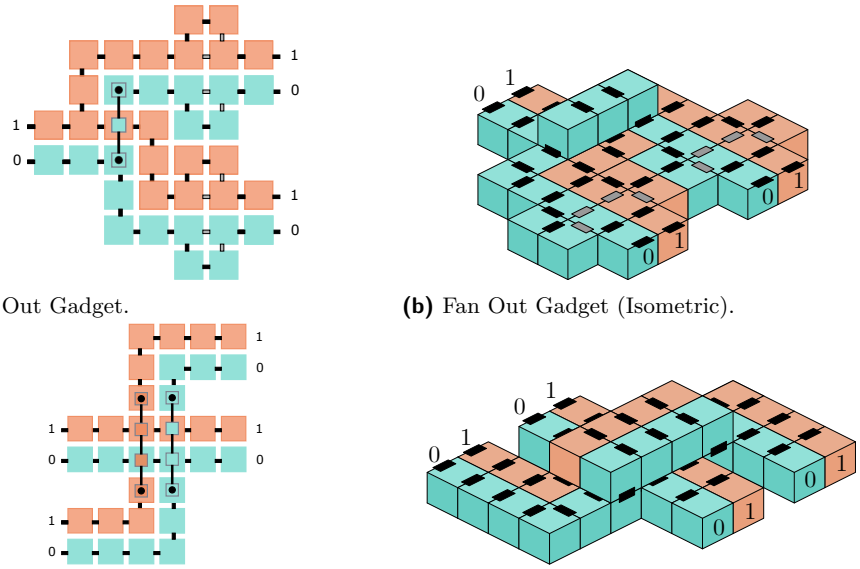
3.5 Backfilling and Target Assemblies

In order to perform covert computation, there must exist a unique assembly for each output. The gray tile at the end of the circuit in Figure 6a is one of two flag tiles that denotes the output of the circuit. Once this tile is placed, a row of tiles is built back towards the input



(a) Input 00. **(b)** Input 01. **(c)** Input 10. **(d)** Input 11.

Figure 4 Growth of possible inputs to a NAND gate. The gate will stay like this after computing, before the history is hidden.



(a) Fan Out Gadget. **(b)** Fan Out Gadget (Isometric).

(c) Crossover Gadget. **(d)** Crossover Gadget (Isometric).

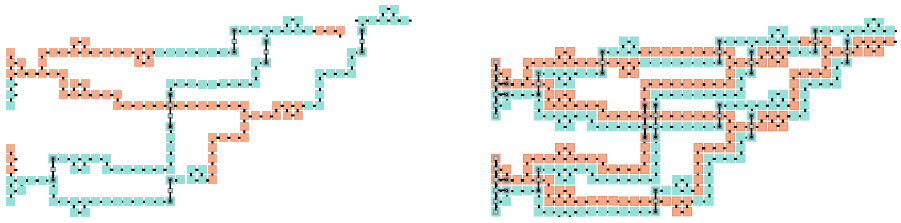
Figure 5 (a) A fan out gadget. (b) Isometric view of the fan out gadget. (c) While a crossover is not required for universal computation, we can easily implement one by using the 3rd dimension. (d) Isometric view of the crossover gadget.

(Figure 6b). Once the input assembly is reached, the tiles above the input are placed, thus allowing for the input assemblies to be filled in. This causes the entire circuit to be filled out, which hides the original input and computation history.

▶ **Theorem 1.** *For any n-bit function f that is computable by a Boolean circuit, there exists a Tile Assembly Computer $\Im$ which covertly computes f in the 3D aTAM with only positive glues. Further, $\Im$ is strictly polynomial in n.*

Proof. We can construct the tile set T_c from the circuit c that computes f. Arrange the gates and wires on the square grid using $\mathcal{O}(n^2)$ space, and scale up each gate and wire by a constant factor. Wires are scaled up by a factor of 2 to account for the dual rail logic wires. The gates are scaled up by a factor depending on which gate it is, however, all the gates we present are only a constant size. This creates assembly $A_{c,Full}$.

We now show that $\Im$ computes f. Consider an n-bit input x to f, using the input template create seed assembly A_x. Each gate will grow from A_x, computing the circuit on each input. Since backfilling does not occur until the circuit finishes computing, we guarantee only the correct outputs grow from the final gate. The circuit is computed covertly since the output then grows back to the start of the circuit and places the unused inputs. ◀



(a) Computed Circuit. **(b)** Output Assembly.

Figure 6 Example structures of the computation circuit of an XOR using NOTs and NANDs. The circuit before backfilling is on the left, and the final output is shown on the right side. (a) A circuit once the output is computed. (b) Once the output grows backward, the other input bits are placed.

The 3rd dimension is vital in this construction to allow signals to cross over for the NOT gate. Notice the part of the NAND gadget that is computing the AND gate and how the diode uses cooperative binding. Additionally, it would not be possible to build the full input gadget to allow the circuit to backfill. The positions that must be filled will be blocked on one side by the input assembly and on the other by wire. The backfilling here is used differently than in [8] since there each gate would backfill its input wires. There the negative glues were used to allow the tiles to cross over signals to build a NOT gate.

4 Exponential Assembly Covert Computer in 2D

In this section, we show that covert computation is possible in 2D in the standard aTAM, where the input can be described in polynomial size, yet the final terminal assembly is exponential in size. Thus, while we are able to achieve strictly polynomial-sized covert computation in 3D, we achieve (non-strict) polynomial-sized covert computation in 2D.

This construction is possible by first computing the function using reversible Toffoli gates, and then replicating and computing the circuit for all possible inputs. Once the output of the original input is placed, the Toffoli gate reverses its computation to build a mirror of the circuit with the input replicated on both the right and left. The output builds an assembly arm used to place tiles on either side of the assembly to increment and decrement the mirrored inputs based on the binary value of the original input, thus seeding a new input for exponential growth in each direction. Thus, for a 4-bit input, it builds the circuit for all 2^4 possible inputs after it builds the output template.

4.1 Toffoli Gate

The Toffoli gate is a 3-bit reversible universal logic gate (Figure 7a), we denote the inputs A, B, C, and the outputs A', B', C'. The first two input and output bits map to each other: $A = A'$ and $B = B'$. The third output flips the C bit if both A and B are 1. Logically expressed, this is $C \otimes (A \wedge B) = C'$.

We can express an n-bit d-depth reversible circuit as a $n \times d$ grid where each row represents a wire, and each column is a layer of gates and wires. Each gate can be represented by tiles computing the elementary 2-bit AND and XOR and implementing a fan out, as shown in Figure 7b.

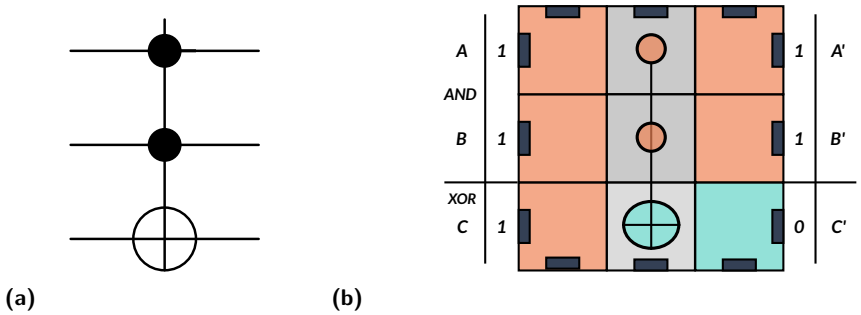


Figure 7 (a) Logical representation of Toffoli gate. (b) A Toffoli gate on a grid can be represented by the three vertical "cells" of elementary logic gates.

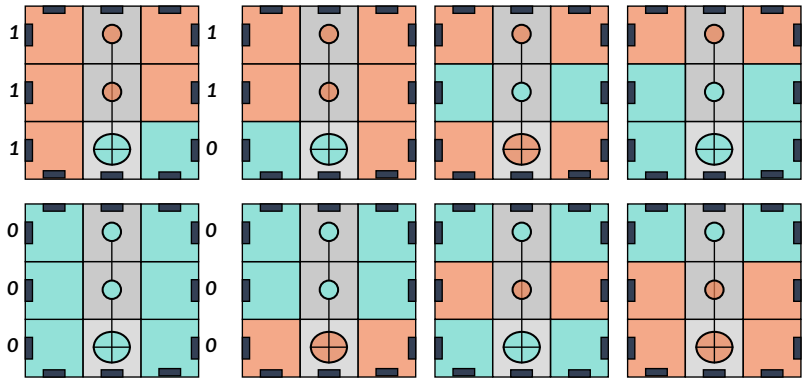


Figure 8 All possible computations of a single Toffoli gate. 1 (orange), 0 (blue). $111 \to 110, 110 \to 111, 101 \to 101, 100 \to 100$, Row 2: $000 \to 000, 001 \to 001, 010 \to 010$, and $011 \to 011$.

4.2 Covert Circuit

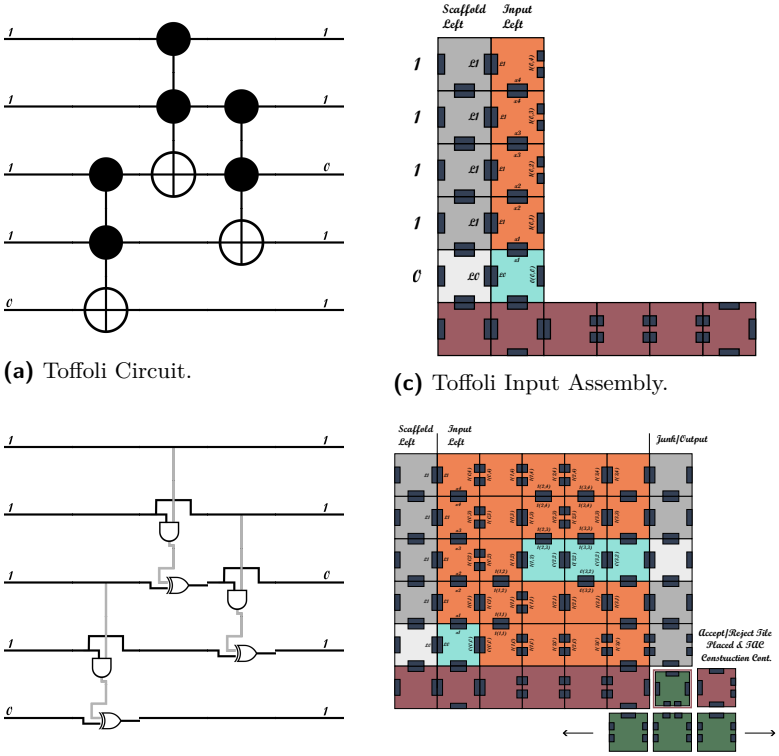
The input template is a specific tile for each bit. Given an n-bit string, we create a $n \times 1$ bit assembly with stability-granting left and bottom circuit construction scaffolds, as shown in Figure 9c.

The circuit assembly is a $n \times (d + 2)$ rectangle. Each Toffoli gate is a 3×1 subassembly. Three possible computations of a single Toffoli gate are shown in Figure 8. Typically, these gates must be reversible, meaning the circuit may grow from the east or west but produce the same assembly We note that the gate itself is not covert, and the "covertness" comes from the full construction.

An example Toffoli circuit is shown in Figure 9a along with the logical representation in Figure 9b. A constructed circuit assembly in one direction can be seen in Figure 9d.

4.3 Increment/Decrement Input to Next Circuit Logic

After completion of a circuit, three columns of tiles are built: mark for increment (left), copy or flip (center), and mark for decrement (right). The order of growth of these columns depends on the starting direction. Growing from the left to increment input to the next circuit or from the right to decrement it. Cooperatively with those columns, below the output arm begins its extension to transmit the outcome, accept or reject, of the original circuit. This arm extension continues to the center circuit output outcome tile location. From here, the circuit construction scaffold, previously provided in the input template, may loop back to



(a) Toffoli Circuit.

(c) Toffoli Input Assembly.

(b) AND/XOR Diagram.

(d) Toffoli Circuit Assembly.

Figure 9 (a) Example 5-bit Toffoli Circuit. (b) The Toffoli circuit represented with AND and XOR gates. (c) Example Input Assembly. For each bit (1 or 0), we place the scaffold (grey or white) and input bit tile (orange or blue). The bottom is a row of circuit construction scaffold tiles (maroon). (d) The Toffoli Circuit Assembly built in one direction. The (green) tile below the output/junk column represents the (positive) output and will allow the output control row to place.

the edge of the circuit so the new input scaffold and bits may place as illustrated in Figure 11. The circuit growth continues normally from that point forward, with the exception of the output tile placement.

4.4 Output Assembly

Once the output is built, the rows below have d tiles attached in the east and west directions that encode the output. Through cooperative attachment, tiles are placed to allow the strings to increment/decrement, as described above. The final terminal assembly contains every possible computation.

▶ **Theorem 2.** *For all functions $f(x)$ that are computable by a n-bit reversible circuit R, there exists a polynomial tile assembly computer $\Im = (T, I, O, 2)$ that covertly computes $f(x)$ and has an output assembly of size $\mathcal{O}(2^n)$.*

Proof. If there exists a n-bit reversible circuit R that computes $f(x)$, we construct tile assembly computer $\Im = (T, I, O, 2)$ as follows. From the circuit R that computes f, we design a circuit R' to compute f with Toffoli gates as described in section 4.2. Using R' and the developed input increment/decrement logic for circuit replication, we construct a tile set T_c.

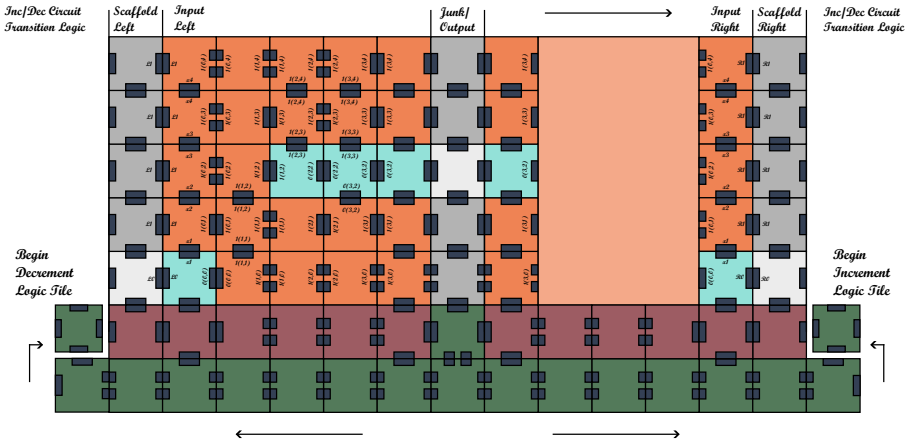


Figure 10 An example of a symmetrical circuit that has built both sides and is placing begin decrement and increment logic tiles.

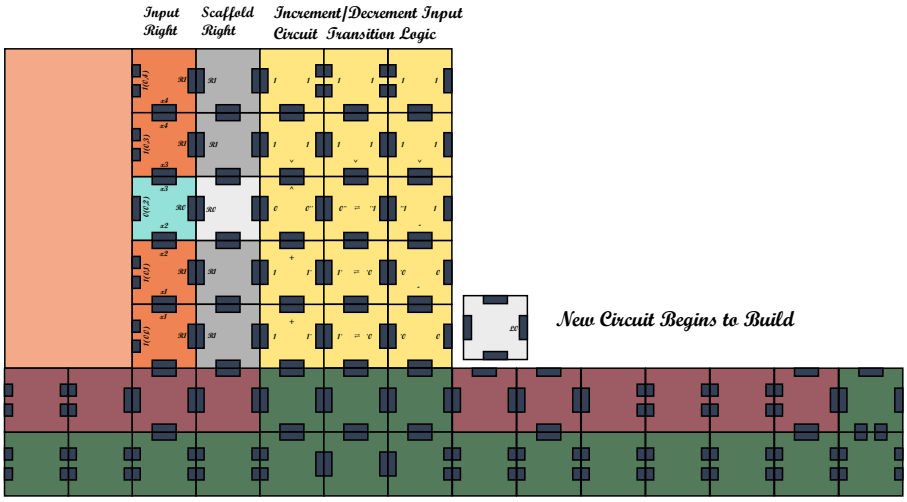


Figure 11 An example of a new circuit created by incrementing the output from a previously built assembly.

We create the input assembly I by converting the n-bit input string x to tiles L_i in scaffold left (figure 9c) and associated input, and a bottom row of tiles called the left circuit construction scaffold.

From here, the left assembly will grow into figure 9d, once the output is determined to be "accept" or "reject", the output indicator tile is placed, and the original output indicator arms grow to allow the Right Assembly the ability to grow as well as place begin decrement and increment logic tiles on the bottom left and right sides of the completed assembly respectively, as seen in figure 10.

All n-bit computations of $f(y)$ for y less than original input x will be computed to the left of the original assembly, and all $x_n > x$ after being decremented and incremented using the reversible and symmetric logic in yellow from figure 11. Growth is halted by the INC/DEC logic at overflow in either direction.

The ability to grow further left/right circuit construction scaffolds is dependent on the output arms from the original output indicator arms growing to the center of the circuit about to begin construction where the output accept/reject indicator tile would place, preserving the output status for every circuit built in the TAC.

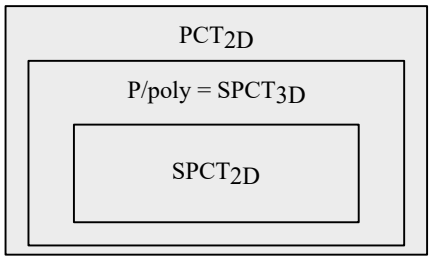


Figure 12 Diagram showing important classes defined in this section and their relation to P/poly. Note that none of these containments are known to be proper.

As there are only two possible assemblies that can be built, accept all or reject all, the Tile Assembly Computer is polynomial size in description and exponential in output size. ◀

We have shown that if the output assembly is allowed to be exponential in size, that covert computation is possible in the aTAM, even in two dimensions. However, in practice, this is not usually a plausible solution. Given that Unique Assembly Verification is in P [2], it is unlikely that covert computation is possible with a strictly polynomial-size TAC.

▶ **Conjecture 3.** *There does not exist a strictly polynomial-size Tile Assembly Computer in the 2D Abstract Tile-Assembly Model.*

5 Polynomial-Sized Covert Circuits

In this section, we define and investigate complexity classes based on decision problems computable by polynomial-sized covert computers. We start by introducing the class P/poly and defining three classes of covertly computable problems: the class of problems covertly computable by a strictly polynomial 3D system (SPCT_{3D}), the class of problems computable by a strictly polynomial 2D system (SPCT_{2D}), and the class of problems computable by a (non-strict) polynomial 2D system (PCT_{2D}). We show how these classes relate to each other, including the result that P/poly is equal to SPCT_{3D}. Our results in this section are summarized in Figure 12.

5.1 Complexity Classes

The class P/poly is a well-studied complexity class defined as the class of problems solvable by a polynomial-sized circuit. One note about this class is it puts no requirement on the circuit other than that it exists. This has an equivalent definition as the problems solvable by a polynomial-time Turing machine with a polynomial advice string. We can think of this as the Turing machine being given a description of the circuit and evaluating it. Here, the advice string or circuit must be identical for all inputs of length n. [1]

▶ **Definition 4** (P/poly). *The class of problems solvable by a polynomial-sized Boolean circuit. Alternatively, defined as the problems solvable by a polynomial-time Turing machine $M < x, a_{|x|} >$, where x is the input and $a_{|x|}$ is an advice string that is based only on the length of x. That is, if two inputs x, y have the same size $|x| = |y|$, then they must use the same advice string.*

[1] Under this definition, every unary language is in this class, including UHALT.

We define the following three complexity classes to categorize the functions that are computable by polynomial-size covert TACs.

▶ **Definition 5** ($\mathsf{SPCT_{3D}}$). *The class of problems solvable by a strict polynomial sized covert tile assembly computer in the 3D Abstract Tile-Assembly Model.*

Formally, a language L is in $SPCT_{3D}$ if there exists a sequence of covert TACs $C = \{C_1, C_2, ...\}$ such that the i^{th} TAC, C_i, is strictly polynomial in i and if it correctly computes all $x \in L$ where $|x| = i$.

▶ **Definition 6** ($\mathsf{SPCT_{2D}}$). *The class of problems solvable by a strict polynomial sized covert tile assembly computer in the 2D Abstract Tile-Assembly Model.*

▶ **Definition 7** ($\mathsf{PCT_{2D}}$). *The class of problems solvable by a polynomial sized covert tile assembly computer in the 2D Abstract Tile-Assembly Model.*

5.2 Strict Polynomial Size Equivalence

To show equivalence between P/poly and $\mathrm{SPCT_{3D}}$, we first define the 2-Promise Unique Assembly Verification problem, a modified version of Unique Assembly Verification where we are given two assemblies, a and b, rather than a single target. The problem asks to separate two cases: accept if an assembly containing a as a subassembly is produced, and reject if an assembly containing b is produced. We assume it is promised that one of these cases is true. This problem is solvable in polynomial time since you only need to attach tiles until one of the two assemblies is produced (Lemma 9).

▶ **Definition 8** (2-Promise Unique Assembly Verification problem). ***Input:*** *Assemblies a, b and an aTAM system (T, s, τ) which is promised to uniquely produce one of two assemblies, A or B, such that $a \subseteq A$ and $b \subset B$.* ***Output:*** *"Yes", if Γ uniquely assembles A, and "No", if Γ uniquely assembles B.*

▶ **Lemma 9.** *The 2-Promise Unique Assembly Verification problem is solvable in polynomial time in the 3D aTAM.*

Proof. Call greedy grow (from [2]) to get maximal producible assembly C. If Γ uniquely assembles C and $a \subseteq C$, return "yes". Otherwise, return "no". ◀

Equipped with the algorithm for the 2-promise problem, and taking the description of a covert computer as an advice string, it follows that we can compute the seed assembly from the input template, and the two possible output assemblies from the output template, and then run the algorithm for the 2-Promise UAV problem (Lemma 10). This puts any problem solvable by a polynomial-sized covert circuit in the class P/poly. The other direction of equivalence is given by the 3D covert computer constructions.

▶ **Lemma 10.** *If a language L is computable by a strict polynomial-sized covert tile assembly computer in the 3D aTAM, then L is in P/poly.*

Proof. Let $\Im_n(T, I, O, \tau)$ be the covert computer for the strings in language L of size n. Since $\Im_n$ is of strict polynomial size, we can encode the tile set, input/output templates, and temperature in $poly(n)$ bits. Thus, $\Im_n$ will be our advice string for membership in P/poly. Further, we are only considering decision problems. Thus, there are only two output templates which we denote as a_a and b_r for accept and reject, respectively.

Consider a Turing machine given the string x and covert circuit $\Im_{|x|} = (T, I, (a_a, b_r), \tau)$ that does the following:

- Convert x to an assembly $I(x)$ using the input template.
- Call the algorithm for 2-Promise UAV on input $((T, I(x), \tau), a_a, b_r)$.
- If the algorithm accepts then $x \in L$, else $x \notin L$

This Turing machine essentially runs the covert computer on x and then checks the output by seeing which template is included in the final assembly. ◀

▶ **Theorem 11.** *The classes $SPCT_{3D}$ and P/poly are equivalent.*

Proof. By Lemma 10, if a language is in P/poly there is a Boolean circuit of polynomial size which computes it, giving us P/poly $\subseteq$ PCT$_{3D}$. In Theorem 1 we show that if there exists a Boolean circuit, there exists a strictly polynomial sized covert computer that computes the circuit. ◀

5.3 Polynomial Sized 2D Covert Circuits

Here, we use previous constructions to show that the class of polynomial sized 2D covert circuits is at least as strong as strict polynomial covert circuits. That is every language in $SPCT_{3D}$ is in PCT_{2D}.

▶ **Theorem 12.** *If a language L is in P/poly then L is in PCT_{2D}*

Proof. In Lemma 10 we show that if a language is in P/poly there is a Boolean circuit of polynomial size which computes it. Any Boolean circuit can be turned into a reversible circuit, thus by Theorem 2, if there exists a reversible circuit, there exists a polynomial tile assembly computer that computes it in 2D. ◀

6 Conclusion and Future Work

Previous work in the aTAM required negative glues in order to build covert Tile Assembly Computers. We have provided two new covert computers in the aTAM with only positive glue strengths, one in (just-barely) 3D and one in 2D with an exponential-sized output assembly. These covert TACs add new tools to the field that may find use in future complexity results, or in future applications related to privacy, cryptography, or biological computation. We have further initiated the study of covert computers in the context of known complexity classes, showing connections to the well-studied class P/poly. These results motivate future work to find functions that can be covertly computed in the 2D aTAM with strict polynomial size, such as (perhaps) Branching Programs.

Some additional specific directions for future work are as follows. We show the containment of the class of strict polynomial computers to be in P/poly. Can this be improved? Could we possibly use the P/poly log space analogue L/poly? What about in smaller classes, such as covert computers with non-cooperative binding or at temperature-1?

References

1 Zachary Abel, Nadia Benbernou, Mirela Damian, Erik D. Demaine, Martin L. Demaine, Robin Flatland, Scott D. Kominers, and Robert Schweller. Shape replication through self-assembly and rnase enzymes. In *Proceedings of the 2010 Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'10, pages 1045–1064, 2010. doi:10.1137/1.9781611973075.85.

2 Leonard M. Adleman, Qi Cheng, Ashish Goel, Ming-Deh A. Huang, David Kempe, Pablo Moisset de Espanés, and Paul W. K. Rothemund. Combinatorial optimization problems in self-assembly. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 23–32, 2002.

3 Andrew Alseth and Matthew J. Patitz. The need for seed (in the abstract tile assembly model). In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA'23, pages 4540–4589, 2023.

4 Spring Berman, Sándor P Fekete, Matthew J Patitz, and Christian Scheideler. Algorithmic foundations of programmable matter (dagstuhl seminar 18331). In *Dagstuhl Reports*, volume 8. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

5 Yuriy Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Comp. Sci.*, 378:17–31, 2007.

6 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Covert computation in staged self-assembly: Verification is pspace-complete. In *Proceedings of the 29th European Symposium on Algorithms*, ESA'21, 2021.

7 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Verification and computation in restricted tile automata. *Natural Computing*, 2021. doi:10.1007/s11047-021-09875-x.

8 Angel A. Cantu, Austin Luchsinger, Robert Schweller, and Tim Wylie. Covert Computation in Self-Assembled Circuits. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:14, 2019.

9 Angel A. Cantu, Austin Luchsinger, Robert T. Schweller, and Tim Wylie. Covert computation in self-assembled circuits. *Algorithmica*, 83:531–552, 2019.

10 Luis Ceze, Jeff Nivala, and Karin Strauss. Molecular digital data storage using dna. *Nature Reviews Genetics*, 20(8):456–466, 2019.

11 Cameron Chalk, Austin Luchsinger, Robert Schweller, and Tim Wylie. Self-assembly of any shape with constant tile types using high temperature. In *Proc. of the 26th Annual European Symposium on Algorithms*, ESA'18, 2018.

12 Cameron T. Chalk, Eric Martinez, Robert T. Schweller, Luis Vega, Andrew Winslow, and Tim Wylie. Optimal staged self-assembly of general shapes. *Algorithmica*, 80(4):1383–1409, 2018. doi:10.1007/s00453-017-0318-0.

13 Qi Cheng, Gagan Aggarwal, Michael H. Goldwasser, Ming-Yang Kao, Robert T. Schweller, and Pablo Moisset de Espanés. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34:1493–1515, 2005.

14 Supreme Court. Ass'n for molecular pathology v. myriad, 2013.

15 Erik Demaine, Matthew Patitz, Robert Schweller, and Scott Summers. Self-assembly of arbitrary shapes using rnase enzymes: Meeting the kolmogorov bound with small scale factor. *Symposium on Theoretical Aspects of Computer Science (STACS2011)*, 9, January 2010. doi:10.4230/LIPIcs.STACS.2011.201.

16 Erik D. Demaine, Sándor P. Fekete, Christian Scheffer, and Arne Schmidt. New geometric algorithms for fully connected staged self-assembly. *Theoretical Computer Science*, 671:4–18, 2017. Computational Self-Assembly. doi:10.1016/j.tcs.2016.11.020.

17 David Doty, Lila Kari, and Benoît Masson. Negative interactions in irreversible self-assembly. *Algorithmica*, 66(1):153–172, 2013.

18 David Doty, Jack H Lutz, Matthew J Patitz, Robert T Schweller, Scott M Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 302–310. IEEE, 2012.

19 Pim WJM Frederix, Ilias Patmanidis, and Siewert J Marrink. Molecular simulations of self-assembling bio-inspired supramolecular systems and their connection to experiments. *Chemical Society Reviews*, 47(10):3470–3489, 2018.

20 David Furcy, Samuel Micka, and Scott M. Summers. Optimal program-size complexity for self-assembled squares at temperature 1 in 3d. *Algorithmica*, 77(4):1240–1282, March 2016. doi:10.1007/s00453-016-0147-6.

21 David Furcy, Scott M. Summers, and Logan Withers. Improved Lower and Upper Bounds on the Tile Complexity of Uniquely Self-Assembling a Thin Rectangle Non-Cooperatively in 3D. In Matthew R. Lakin and Petr Šulc, editors, *27th International Conference on DNA Computing and Molecular Programming (DNA 27)*, volume 205 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DNA.27.4.

22 Daniel Hader, Aaron Koch, Matthew J Patitz, and Michael Sharp. The impacts of dimensionality, diffusion, and directedness on intrinsic universality in the abstract tile assembly model. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2607–2624. SIAM, 2020.

23 Adam M Kabza, Nandini Kundu, Wenrui Zhong, and Jonathan T Sczepanski. Integration of chemically modified nucleotides with dna strand displacement reactions for applications in living systems. *Wiley Interdisciplinary Reviews: Nanomedicine and Nanobiotechnology*, 14(2):e1743, 2022.

24 Alexandra Keenan, Robert Schweller, Michael Sherman, and Xingsi Zhong. Fast arithmetic in algorithmic self-assembly. *Natural Computing*, 15(1):115–128, March 2016.

25 Paul WK Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468, 2000.

26 Robert Schweller, Andrew Winslow, and Tim Wylie. Complexities for high-temperature two-handed tile self-assembly. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, pages 98–109, Cham, 2017. Springer International Publishing.

27 Heribert Vollmer. *Introduction to Circuit Complexity*. Springer Berlin Heidelberg, 1999. doi:10.1007/978-3-662-03927-4.

28 Klaus F Wagenbauer, Christian Sigl, and Hendrik Dietz. Gigadalton-scale shape-programmable dna assemblies. *Nature*, 552(7683):78–83, 2017.

29 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.