Preserving Privacy in Distributed Computation via Self-Assembly

Yuriy Brun and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{ybrun,neno}@usc.edu

Abstract

We present the tile style, an architectural style that allows the creation of distributed software systems for solving NP-complete problems on large public networks. The tile style preserves the privacy of the algorithm and data, tolerates faulty and malicious nodes, and scales well to leverage the size of the public network to accelerate the computation. We exploit the known property of NP-complete problems to transform important real-world problems, such as protein folding, image recognition, and resource allocation, into canonical problems, such as 3-SAT, that the tile style solves. We provide a full formal analysis of the tile style that indicates the style preserves data privacy as long as no adversary controls more than half of the public network. We also present an empirical evaluation showing that problems requiring privacy-preservation can be solved on a very large network using the tile style orders of magnitude faster than using existing alternatives.

1 Introduction

Solving certain important computational problems currently requires time exponential in the size of that problem, and thus single computers can practically solve only small instances of such problems. Large networks, such as the Internet, have the potential to solve larger instances significantly faster. The ability to solve moderate-sized instances of such problems hundreds of times faster than a single machine has substantial academic, financial, and social implications and greatly impacts such fields as medicine, management, and systems engineering. For example, the ability to determine proteins' minimal-free-energy structures within days (as opposed to years) could lead to cures or treatments of cancers, HIV, and other life-threatening diseases. The ability to predict the optimal allocation of resources to a project could cut costs of public and private projects dramatically and assist in properly planning for such large endeavors as Boston's Big Dig.

Some have already used the Internet to solve complex computationally-intensive problems (e.g., SETI@home [16] and Folding@Home [18]); however, these techniques do not apply to problems with data that must remain private throughout the computation. We call techniques for distributing computation *privacy-preserving* if the involved data remain private during and after the computation. In this paper, we present the tile architectural style, a solution which allows distributing complex computationally intensive problems on large public networks while preserving the privacy of the algorithm and data, tolerating faulty and malicious nodes, and scaling well to leverage the size of the public network to accelerate the computation. The following are two example scenarios the tile style targets.

1. A pharmaceutical company has generated a series of candidate proteins for treating a particular cancer. The company needs to predict the 3-D structure of the proteins as they would fold within the human body but the proteins' amino acid sequences are valuable intellectual property and must remain private. The protein folding problem is NP-complete [2], and thus for reasonably-sized proteins, it could take years on a single computer, or even on small private networks, to compute the desired structures. The company is unwilling to use existing approaches [18] to distribute the computation on a public network because they distribute the amino acid sequences to all helping nodes.

2. Image recognition is at the heart of many advanced artificial intelligence and security tasks. Matching faces seen in a camera to a database of known criminals allows automated intruder detection and aids security at public locations such as airports and casinos. However, facial recognition and image matching problems are NP-complete [15] and many people may enter the location of interest at once. Further, any employed solution must execute quickly to deliver results in real-time. In order to protect the identity and privacy of the innocent individuals entering the location, the system must either guarantee that the entire computation

takes place on a completely secure large private network, or use a privacy-preserving technique, such as the tile style, to distribute the computation on a public network.

The tile style decomposes a computational-problem algorithm into some of its most basic operations and assigns individual nodes on a network to deploy objects representing each of the input and intermediary data bits. The objects then communicate over the network to compute the results, in some sense self-assembling the computation. While this process requires significant network communication, because the tile style targets NP-complete problems, which have a large (exponential in the size of the input) number of independent threads, many nonblocking computations can be executed in parallel, thus ensuring that no node ever waits for network communication. This property allows tile-style systems to leverage network size to enhance the speed of the computation. The high distribution of data yields preservation of privacy. And finally, redundancy techniques directly aid fault- and adversary-tolerance.

We demonstrate a formal mathematical analysis of the communication and computation our systems perform and prove bounds on their time requirements. We also formally prove that tile style systems preserve the privacy of the data used in the computation as long as no adversary controls more one than half of the public network. We have previously demonstrated that tile-style systems are fault- and adversary-tolerant [8], and do not repeat those arguments here. To further strengthen the feasibility and applicability of our technique and confirm our theoretical results, we present an implementation of a tile-style system and execute it on small- and moderate-sized networks.

This paper is a significant revision and expansion of our previously published workshop proposal [7]. We augment, correct, and replace many of the algorithms we had originally proposed. We also improve the tile style's privacy preservation by creating a tile solution for the well-known NP-complete 3-SAT problem. Finally, we present the formal mathematical and empirical analyses of the tile style, and demonstrate an implementation of tile-style systems.

The primary contribution of this paper is an architectural style that allows the creation of distributed software systems for solving NP-complete problems on large public networks while preserving the privacy of the algorithm and data, tolerating faulty and malicious nodes, and scaling well to leverage the size of the public network to accelerate the computation. While this style relies on a biologically-inspired tile model, in order to apply the style, an architect needs only familiarity with architectural concepts such as components and connectors, and not the underlying tile mechanisms. Those mechanisms are under-the-hood details, much like the machine code that executes when one writes a high-level program.

The rest of this paper is structured as follows: Section 2 positions our work in terms of related research. Section 3 explains how computation is possible in the tile assembly model. Section 4 describes the tile architectural style. Section 5 discusses tile style's key properties and analysis. Finally, Section 6 concludes the paper.

2 Related Work

In this section, we describe related work in the areas of software architecture, distributing computation onto untrusted hosts, and privacy-preserving computation.

2.1 Software Architecture

Software architecture has been identified as an important part of building almost all large systems [23]. A poor underlying software architecture can be disastrous, while a good one helps to ensure the system's key properties, such as performance, reliability, portability, scalability, and interoperability. Architectural styles can be used to "force" a software system to conform to certain rules, thus resulting in some desired properties. For example, mandating that two components communicate via implicit invocation can result in systems that are more easily evolvable. However, it is also possible to provide desired system properties as an emergent behavior of the architecture without forcing restrictions on the system designer. Several researchers have argued that for a system to be self-healing, the system must be self-observant and alter its behavior in hostile environments (e.g., [17, 20]). However, in our proposed tile architectural style, the system exhibits properties of self-healing naturally, without observing or altering its behavior. Similarly, Devanbu et al. have argued that security, a crucial property of most modern software systems, may be implemented in the connectors mediating the interactions among the system's components [12]. Accordingly, the tile style, in principle, allows for security in the connectors; however, privacy preservation, one aspect of security, is an *emergent* property of the style.

2.2 Getting Help with Computation

The growth of the Internet has made it possible to use public computers to distribute computation to willing hosts. Software designed to solve computationally intensive problems has emerged to take advantage of this phenomenon, enticing users to devote their computers' idle cycles to some academically or otherwise worthy cause. This notion focuses the underpinning of computational grids [13]. Among systems that concentrate on distributed computation are SETI@home [16] and Folding@Home [18]. These systems try to solve exactly the highly parallelizable problems toward which our work is geared, but unlike the tile style,

they do not preserve privacy. The majority of research on strategies for getting computational help without disclosing the input of the computation has focused on asking a single other computer for help. Yet, in classical computing, it is not possible to get help from a single entity in solving an NP-complete problem without disclosing most of the information about the input and the problem one is trying to solve [10]. Our approach consists of distributing such a request over many machines without disclosing the entire problem to any small-enough group of them.

2.3 Secure Computation

The field of secure multi-party computation explores whether multiple computers, each of whom knows part of an input, can compute a function of that entire input without sharing their parts with others. The tile style is a solution to a related, but fundamentally different problem: can computers be used to help compute a function if no sizable group of those computers knows the input to the computation? The seminal work in the area of secure multi-party computation introduced Yao's garbled circuit protocol that allows n nodes, each with access to a single input, to compute a function of the n inputs while disclosing only the value of the function to each node [26]. Zero-knowledge compilers bridge that work closer to our approach by making Yao's protocol secure even if the parties cannot be trusted [14]. Secure multi-party computation applies to functions on large distributed private data sets, while our work applies to functions on fairly small data sets, but ones that require exponential time or space to compute. Our work does, at times, leverage some of the work in secure multi-party computation, as we describe in Section 4.3.

3 Computing with Tiles

The tile assembly model is a formal model of molecular self-assembly that describes how simple molecules can form complex crystals. In this model, molecules are square tiles with special labels on their four sides. Tiles can stick together under certain conditions when their abutting sides' labels match. Winfree showed that this process allows molecules to compute functions proved that the model is Turing universal [25]. However, Winfree's acknowledges that using his approach to solve computational problems produces highly inefficient assemblies and resembles programming using Turing machines. We have continued Winfree's work by developing the notion of efficient computation within the tile assembly model and constructing efficient assemblies to add and multiply integers [3], factor integers [4], and solve NP-complete problems [5, 6]. Section 3.2 will describe the assembly that solves 3-SAT.

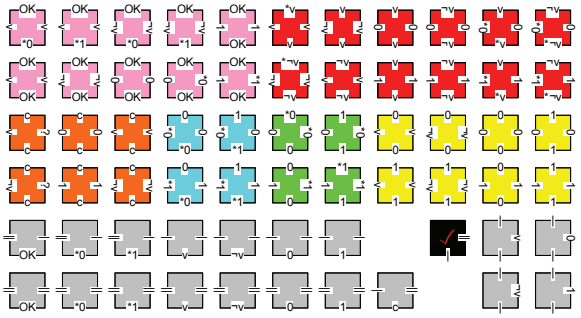

Figure 1. A tile assembly that solves 3-SAT consists of 64 tile types

3.1 Theoretical Underpinnings

The tile assembly model has *tiles*, or squares, that stick or do not stick together based on various *binding domains* on their four sides. Each tile has a binding domain on its north, east, south, and west side, and each distinct binding domain has an integer *strength* associated with it. The four binding domains, elements of a finite alphabet, define the type of the tile. The placement of a set of tiles on a 2-D grid is called a *crystal*; a tile may *attach* in empty positions on the crystal if the total strength of all the binding domains on that tile that match its neighbors exceeds the current *temperature*. Starting from a *seed crystal*, tiles may attach to form new crystals. Sometimes, several tiles may satisfy the conditions necessary to attach at a position, in which case the attachment is nondeterministic. A tile assembly $\mathbb{S}$ computes a function $f : \mathbb{N}^n \rightarrow \mathbb{N}^m$ if there exists a mapping i from $\mathbb{N}^n$ to crystals and a mapping o from crystals to $\mathbb{N}^m$ such that for all inputs $\vec{\alpha} \in \mathbb{N}^n$, $i(\vec{\alpha})$ is a seed crystal such that $\mathbb{S}$ attaches tiles to produce a terminal crystal F and $o(F) = f(\vec{\alpha})$. In other words, if there exists a way to encode inputs as crystals, the system must attach tiles to produce crystals that encode the output. For those systems that allow nondeterministic attachments, the terminal crystal F that encodes the output must contain a special *identifier* tile.

3.2 3-SAT Tile Assembly

3-SAT is a well-known NP-complete problem. The problem consists of determining whether a Boolean formula in conjunctive normal form (3-CNF) is satisfiable by a truth assignment. The input to the problem is the Boolean formula and the output is 1 if the formula is satisfiable and 0 otherwise.

The nature of NP-complete problems is that the ability to solve one such problem quickly implies the ability to solve all such problems quickly. For example, if one finds

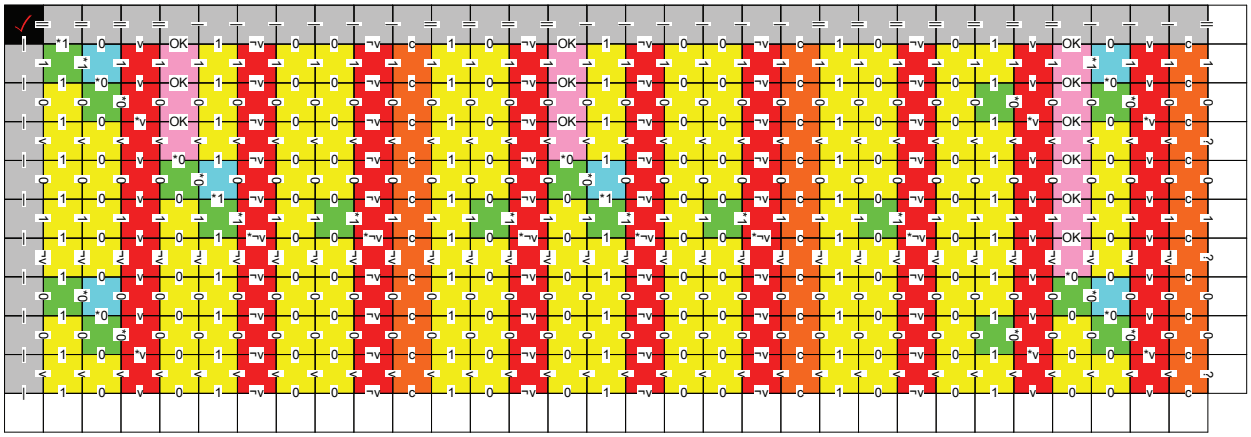

Figure 2. An example crystal of the 3-SAT-solving tile assembly. The clear tiles along the bottom row encode the input $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$. Because ϕ is satisfied when $x_0 = x_2 = TRUE$ and $x_1 = FALSE$, represented by the tiles in the second from the right column, the ✓ tile attaches in the northwest corner. If no truth assignment satisfied ϕ, no such tile could attach.

a polynomial-time algorithm to solve 3-SAT, one can then solve the traveling salesman, protein folding, and all other NP problems in polynomial time. Thus, it is sufficient to design a system that uses a large distributed network to solve one NP-complete problem, e.g., 3-SAT, while preserving privacy. We present a tile assembly that solves 3-SAT.

Figure 1 shows the tiles of the 3-SAT-solving assembly. The tiles "communicate" via their side interfaces. Some interfaces contain a 0 or a 1, communicating a single bit to their neighbors. Other interfaces include special symbols such as v and $\neg v$ indicating that a variable is being addressed, $\star$ meaning that a comparison should take place, ? meaning the given tile attaches nondeterministically, and | and || indicating the correctness of the computation up to this point. The assembly nondeterministically selects a variable truth assignment and checks if that assignment satisfies the formula. If and only if it does, a special ✓ tile attaches to the assembly as described below.

Figure 2 shows a sample crystal of the tile assembly that solves 3-SAT. The example asks the question whether or not $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ is satisfiable. This ϕ is encoded along the bottom row of the crystal. The crystal corresponds to the truth assignment $\langle x_0, x_1, x_2 \rangle = \langle TRUE, FALSE, TRUE \rangle$. The variables x_0, x_1, x_2 are encoded along the rightmost column of the crystal. Together, the bottom row and rightmost column form the seed of the computation, and the remaining tiles self-assemble to nondeterministically select a truth assignment (in the second from the right column) and check whether that assignment satisfies ϕ. Because this truth assignment satisfies ϕ, the ✓ tile attaches in the northwest corner. If no truth assignment satisfied ϕ, no such tile could attach. We are unable to provide the formal proof that this assembly solves 3-SAT due to space constraints and refer the reader to [6] for the complete proof. As part of this work, we have also provided a tile assembly solution for $SubsetSum$ another well-known NP-complete problem [5].

4 Tile Architectural Style

Section 3.2 described how tiles can solve a particular NP-complete problem. For each computable function, it is possible to create a tile assembly to compute that function, in an analogous fashion. A tile-style architecture is based on a tile assembly refined into software components and composed according to the style rules described below. While a user may wish to create a custom tile assembly to solve a particular NP problem, we advise taking an alternate approach. Because NP-complete problems are polynomially related, it is possible to translate all NP problems to 3-SAT [22]. Thus a user who wishes to solve a particular problem using the tile style needs to neither understand the tile assembly model nor program with tiles. The user should translate the problem to 3-SAT, or another problem with a known tile solution (e.g., $SubsetSum$ [5]), and use that solution to guide the tile-style architecture. In addition to ease of use, this procedure masks the problem the user is solving. Even if an adversary were to control an overwhelming portion of the public network and compromise the tile-style computation, that adversary could learn that the system is solving 3-SAT, but not the original problem. While this is a beneficial side effect of using the tile style, we discuss the much more important privacy-preservation property in

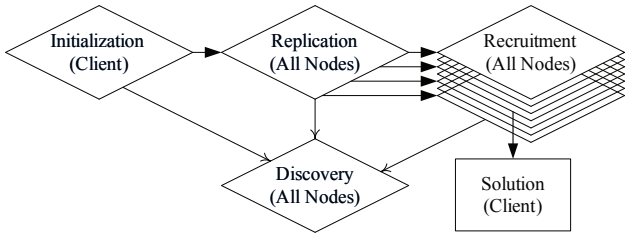

Figure 3. Overview of tile style node operations.

Section 5.1.

The components of the tile-style architecture are instantiations of the tile types of the underlying assembly. While a system based on such an architecture will have a large number of components, there is a comparatively smaller number of different *types* of components (e.g., 64 types for solving 3-SAT). Nodes on the network will contain these components, and components that are adjacent in a crystal can recruit other components to attach, by sampling nodes until they find one whose side labels, or interfaces, match. Note that many tile components can run on a single physical node, as we will further elaborate below.

In addition to defining the tile types, a tile assembly also directs the architecture how to encode the input to the computation. The input consists of a seed crystal, such as the clear tiles along the right and bottom edges in Figure 2. Figure 3 summarizes the steps a tile-style system takes to find a solution. During *initialization*, the system sets up a single seed crystal on the network that encodes the input. The seed then *replicates* to create many copies, and each of the copies *recruits* tiles to assemble larger crystals and eventually produce the solution. The solution tile components (e.g., the ✓ component for the 3-SAT assembly) then report their state to the user. We now describe in detail the four involved operations. Note that the nodes perform these operations autonomously, without central control, in essence self-assembling the computation.

4.1 Initializing Computation

The client computer initializes the computation by performing three actions: creating the tile type map, distributing the map and tile type descriptions, and setting up a seed crystal. As our analysis in this section will show, the entire initialization procedure will take on the order of $\log N$ time for a network of N nodes, and each node will send a small amount of data proportional to its local neighborhood size.

4.1.1 Creating the Tile Type Map

A tile type map is a mapping from a large set of numbers (e.g., all 128-bit IP addresses) to tile types. It determines the type of tile components a computer with a given IP address (or other unique identifier that is harder for a potential adversary to control in bulk) deploys. The tile type map breaks up the set of numbers into k roughly equal-sized regions, where k is the number of types of tiles in the tile assembly. For the 3-SAT example from Section 3.2, there are 64 different tile types, so the tile type map would divide the set of all 128-bit numbers into 64 regions of size 2^{122}. The size of the tile type map, which will later be sent to all the nodes on the network, is small. For an assembly with k tile types, the map is k 128-bit numbers.

For our analysis, we assume that every node on the network is connected to p other nodes, distributed roughly randomly. This is a first-order approximation of the Internet, but our analysis will extend to more accurate models. Every computer may contact its neighbors directly and may query its neighbors for their lists of neighbors. A number of our algorithms are designed specifically to work on such a distributed network, on which no single node knows a large portion of the network. On more highly connected networks, our algorithms can be simplified.

4.1.2 Distributing the Map and Tile Descriptions

The client node distributes the tile type map and a short description of one tile type to a node that deploy that tile type, as determined by the tile type map. A tile type's description consists of the four tile component interfaces, which can be described using just a few bits. The client node contacts at least one node that deploys each tile type by contacting its neighbors, then their neighbors, etc. until at least one node of each type knows the tile type map and its tile type description. For a system with k tile types, it will take, with high probability, less than $2k \log k$ time to "collect" a node of each type. This analysis comes from a well-known *coupon collector* problem [21].

The nodes that learn their types from the client computer propagate the information to their neighbors whose IPs map to the same tile types, and so on, until every computer on the network learns the type of tile component that computer will deploy. Thus every computer receives the tile type map and the description of its own tile type. Each computer might receive its tile type information and the tile type map several times, up to as many times as it has neighbors, which on our network is only p. Each node sends only $\Theta(p)$ data because roughly $\frac{1}{k}$ of a node's p neighbors will have to be sent the $128k$ bits, and $\left(\frac{128kp}{k} = \Theta(p)\right)$. Because the diameter of a network of N nodes with randomly distributed connections is $\Theta(\log N)$ [21], the tile type map and the tile types will propagate through the network in $\Theta(\log N)$ time.

Until now, we have ignored the case of a network with fewer nodes than the number of types of tiles. If the network is that small, it is possible to create multiple virtual nodes

on each machine and proceed as before, though a single physical node will have knowledge of more than one tile type, compromising privacy. In the limit, for a network with a single node, it has been analytically shown that privacy preservation is not possible [10].

4.1.3 Creating a Seed

The client is responsible for creating the first seed on the network through a fairly straightforward procedure. For each tile in the seed crystal described by the underlying tile assembly, the client selects a node that deploys that tile type (as we describe in Section 4.2), and asks that node to deploy a tile. The client then informs each deployed tile component who its neighbors on the network are. This procedure is significantly faster and requires significantly less network communication than the distribution of the tile type map.

4.2 Discovery

The node discovery operation is central to the tile style because initialization, replication, and recruitment all use this operation. The discovery operation, given a tile type, returns a *uniformly-random* IP of some computer deploying tile components of that type, meaning that if a node performs this operation repeatedly, the frequencies of the IP addresses it returns asymptotically approach the uniform distribution. Thus, every suitable computer has an equal chance of being returned, in the long run. Our algorithm for discovery will guarantee uniform-randomness, which in turn will guarantee that all nodes on the network perform a similar amount of computation. The algorithm will use a property of random walks to ensure uniform-randomness.

In order to quickly return the IP address of a computer that deploys tile components of a certain type, each node will keep a table, called the node table, of three IP addresses for each component type, as we explain below. For 3-SAT, the size of this table will be $64 \times 3 = 192$ IPs. The table contains only an identifier for each tile type, and not the details about the side labels. The preprocessing necessary to create the node table is simple: first a node fills in the table with all its neighbors and then gets help from neighbors (by requesting their neighbor lists). The analysis of this procedure is identical to the analysis of distributing the tile type map; this preprocessing procedure will take $\Theta(k \log k)$ time per node (happening in parallel for each node), for k different tile types. The amount of data sent by each node is limited to $\Theta(k \log k)$ packets. For 3-SAT's $k = 64$, that is fewer than 300 packets, which for typical UDP packets amounts to only 15 kilobytes.

After the preprocessing, when queried for the IP of a computer that deploys tile components of a given type, the node performs two steps: (1) it selects one of the three entries in the node table for that tile type, at random, and (2) it replaces its list of three entries in the table with the selected node's corresponding three entries. The reason for the replacement is that we want the selection of IPs to emulate a random walk on the node graph [21]. The request packet only needs to contain the tile type (e.g., a 32-bit number) and the answer packet must contain three IPs (three 128-bit numbers). This entire procedure takes $\Theta(1)$ time.

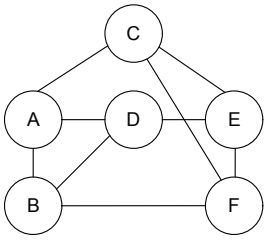

Figure 4. A network with six nodes. We assume that every node in our underlying network has p neighbors (here $p = 3$). It is straightforward to convert most networks into such networks: nodes with too few neighbors can discover more via their neighbors, and nodes that have too many can ignore those.

We now help clarify the preprocessing and discovery operations with the use of an example. Suppose the network in Figure 4 represents the connectivity of six nodes that all map to the same tile type. In creating its node table, A first checks its neighbors B, C, and D, and records them in the three slots for that tile type. A's node table (for that tile type) is now complete, but had A not found three valid nodes to fill its table, it would expand its neighbor list by querying one of its neighbors for its neighbors, until it discovered a sufficiently large portion of the network. B follows the same procedure as A and creates a node table and records its neighbors A, D, and F as the three nodes deploying the same tile type. When A needs a node of that type later (for reasons discussed below), it selects a random node from its three entries. Suppose it selects B. A then replaces its node table entries with B's entries (A, D, F). Note that it is possible for a node to store itself on its node table.

Lemma 1. *On a network on N nodes, after filling only $\Theta(\log N)$ requests for an IP of a computer that deploys a certain tile type using the above-outlined procedure, the probability of each valid IP being returned is uniformly distributed.*

Proof. Because the node table keeps independent lists of three nodes of each type, it is sufficient to prove the theorem for a single tile component type. Consider the directed graph G formed by representing every node as a vertex with three outgoing edges to the vertices representing the nodes on the node table. Now consider a sequence of nodes de-

rived by the above-outlined procedure of picking a random node from the three entries, and replacing those three entries with that node's entries. That sequence corresponds to a random walk on G. From [21], we know that a random walk on G mixes rapidly, which means that if selecting nodes via this random walk after $\Theta(\log N)$ steps, the probability of getting the IP of each node becomes proportional to that node's in degree. Thus on a uniform graph, every IP is equally likely to be returned. □

We have discussed how to convert a random network into one such that each node has exactly three neighbors. Again we emphasize that this simplification is made to aid our analysis. In fact, the random walk theorem from [21] holds for all graphs with nodes having three or more neighbors, so this result is directly applicable to all reasonable distributed networks. For small networks, discovering the entire network does not pose computational difficulty, and selecting nodes uniformly-randomly is trivial.

4.3 Recruitment

The seed crystal grows into a full assembly by recruiting tile attachments. In a computational tile assembly (such as the assembly described in Section 3.2 that solves 3-SAT), a tile that has both a north and a west neighbor recruits a new tile to attach to its northwest. Figure 5 indicates several places in a sample crystal where tile components are ready to recruit new tiles. A recruiting tile component X (highlighted in Figure 5), for each tile type, picks a potential attachment node Y of that type from its node table, as described in Section 4.2, and sends it an attachment request. An attachment request consists of X's north neighbor's west interface and X's west neighbor's north interface. If those interfaces match Y's east and south interfaces, respectively, then Y can attach. At that point, X informs Y of the IPs of its two new neighbors, and those neighbors of Y's IP. Note that X can perform this operation without ever learning its neighbors' interfaces by using Yao's garbled protocol [26], which is crucial for privacy preservation.

Each component's recruitment can be described as a five-step process: X asks N (its north neighbor) to encode its west domain, N asks W (X's west neighbor) to encode its north domain, W responds to X, X sends attachment requests to a set of potential attachments Y, and those Ys reply to X. We will analyze these five steps in Section 5.2 when we compute the speed of tile-style systems.

In the example from Section 3.2, the successful crystal recruits 310 tile components (non-clear tiles in Figure 2). An unsuccessful crystal, which we discuss further in Section 4.5 can recruit fewer, but no more than 310 tiles.

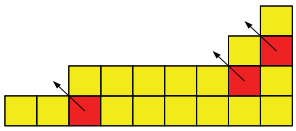

Figure 5. Tile components that have both a north and a west neighbor (highlighted in the diagram) can recruit new components to attach to their northwest.

4.4 Replication

Whenever network nodes have extra cycles they are not using for recruitment, they replicate the seed. Each node X uses its node table, as described in Section 4.2, to find another node Y on the network that deploys the same type components as itself, and sends it a replication request. A replication request consists of up to two IP addresses (four 128-bit numbers) of X's neighbors. X lets its neighbors know that Y is X's replica (by sending Y's IP to X's neighbors). Those neighbors, when they replicate using this exact mechanism, will send their replicas' IPs to Y. Thus, the entire seed replicates. Each component's replication can thus be described as a three-step process: X sends a replication request to Y, Y replies to X, and X tells its neighbors about Y. We will analyze these three steps in Section 5.2 when we compute the speed of tile-style systems.

At the start of the computation, while there are very few recruiting seeds, the replication will create an exponentially growing number of identical seeds (the first seed will replicate to create two, those two will create four, then eight, etc.). When there are sufficiently many seeds to keep the nodes occupied recruiting, replication naturally slows down because recruitment has a higher priority than replication. As some seeds complete recruitment and free up nodes' cycles, replication will once again create more seeds.

The seeds continue to replicate and self-assemble until one of the assemblies finds the solution, at which time the client broadcasts a signal to cease computation by sending a small "STOP" packet to all its neighbors, and they forward that packet to their neighbors, and so on. As discussed above, the diameter of a large connected network of N nodes with randomly distributed connections is $\Theta(\log N)$ [21], so the "STOP" message will propagate in $\Theta(\log N)$ time.

4.5 Answering 3-SAT in the Negative

A crystal that finds the truth assignment that satisfies the Boolean formula reports the success to the client computer. Since for NP-complete problems the answer is always "yes" or "no," the notification is only a few bits. Deciding that

there is no satisfying assignment is more difficult. No crystal can claim to have found the proof that no such assignment exists. Rather, the absence of crystals that have found such an assignment stands to provide some certainty that it does not exist. Because for an input on n variables there are 2^n possible assignments, if 2^n randomly-selected crystals find no suitable assignment, then the client knows there does not exist such an assignment with probability at least $\left(1 - e^{-1}\right)$. After exploring $m \times 2^n$ crystals, the probability grows to at least $(1 - e^{-m})$. Thus as time grows linearly, the probability of error diminishes exponentially. Given the network size and bandwidth, it is possible to determine how long one must wait to get the probability of an error arbitrarily low. In the example from Section 3.2 with 3 variables, the probability of exploring $2^3 = 8$ crystals and not finding the solution is no more than e^{-1}. After exploring 80 crystals, that probability drops to $e^{-10} < 10^{-4}$. Note that no crystal can be larger than 310 tiles, so 80 crystals would require fewer than 25,000 tile components. Because the tile components are lightweight (each one is far smaller than 1 KiB), there is little reason why even a single computer could not deploy that many components.

5 Analysis of the Tile Style

In this section, we first prove several privacy-preservation properties of tile-style systems. We then discuss the efficiency and scalability of the style. We do not discuss the details of fault and adversary tolerance of tile-style systems in this paper, but refer the reader to [8] for formal proofs. Briefly, the tile style allows the architect to specify an upper bound on the number of nodes that are faulty or malicious (e.g., 25%) and an acceptable error rate (e.g., 10^{-20}), and then, by applying redundancy techniques, automatically guarantees the system's probability of failure is below the acceptable error rate at a logarithmically small slowdown in execution speed.

5.1 Privacy Preservation

We call a distributed system *privacy preserving* if, with high probability, no node on the network can discover the entire input to the computational problem the system is solving. We make two arguments in showing that tile style-based systems preserve privacy: (1) given a single tile in a crystal, it is not possible to learn any information about the input and (2) controlling enough computers to learn the entire input is prohibitively hard on a large public network. Proofs and discussion of these statements illustrate that as long as an adversary controls less than half the network, there is an extremely high probability that the adversary cannot learn the input.

1. For a tile assembly, such as the one solving 3-SAT, each tile type encodes no more than one bit of the input. A special tile encodes the solution, but has no knowledge of the input. If every tile component in the crystal were deployed by a different node on the network, it would be trivial to argue that the computation preserved the privacy. However, since a single node on the network may deploy several tile components of the same type, the argument relies on the fact that each component is unaware of its location in the crystal, and thus does not know the location of the bits of the input. Thus, every node on the network may be aware of either some bits of the input or the solution, but not both, and a node cannot use the partial information it has about the bits of the input to recompose that input in its entirety. That is, the nodes can learn information such as "there is at least one 0 bit in the input," but no more.

2. It is clear that if an adversary controls or can see the internal data of the entire network, that adversary can learn the input to the problem. However, the likelihood of such a scenario on a very large public network is exceptionally low. An interesting question becomes how much of the network an adversary must control in order to learn the n-bit input.

Lemma 2. *Let c be the fraction of the network that an adversary has compromised, let s be the number of seeds deployed during a computation, and let n be the number of bits (tiles) in an input. Then the probability that the compromised computers contain an entire input seed to a tile-style system is* $1 - (1 - c^n)^s$.

Proof. If an adversary controls a c fraction of the network nodes, then for each tile in a seed, the adversary has a probability c of controlling it. Thus for a given n-bit seed, distributed independently on the nodes, the adversary has probability c^n of controlling all the nodes that deploy the tiles in the seed, and thus the probability that the seed is not entirely controlled is $1 - c^n$. Since there are s independent seeds deployed, the probability that none of them are entirely controlled is $(1 - c^n)^s$. Finally, the probability that the adversary controls at least one seed is $1-(1 - c^n)^s$. □

Let us examine a sample scenario. Suppose we deploy a tile-style system on a network of $2^{20} \approx$ 1,000,000 machines to solve a 38-variable 100-clause 3-SAT problem. Let us also suppose a powerful adversary has gained control of 12.5% of that network. In order to solve this problem, the system will need to deploy no more than 2^{38} seeds, thus the adversary will be able to reconstruct the seed with probability $1 - \left(1 - 2^{-114}\right)^{2^{38}} < 10^{-22}$. Note that as the input size increases, this probability further decreases. The probabilty decays exponentially for all $c < \frac{1}{2}$ (that is, as long as the adversary controls less than one half of the network). In the above example, control of 25% of the network gives the adversary a probability of reconstructing the input

below 10^{-11}, and control of 33% of the network yields a probability no greater than 10^{-6}. An adversary who controls exactly half the network has a $\frac{1}{e} \approx 37\%$ chance of learning the input, and one who controls more than half the network is very likely to be able to learn the input, which is why our technique is geared towards large public networks.

One possible challenge to privacy preservation on large public networks is botnets. However, no single botnet comes even close to controlling a significant fraction, $\left(\text{say, more than } \frac{1}{1000}\right)$, of the Internet [11]. As the Internet grows, for any fixed-size botnet, the probability that botnet can affect a tile style system drops exponentially.

We have shown the analysis of the number on nodes necessary to compromise the entire input. The same analysis and exponential probability drop off applies to reconstructing fractional parts (e.g., one half or one third) of the input. It is somewhat simpler to reconstruct small fragments of the input (e.g., two- or three-bit pieces), but the information contained in those fragments is greatly limited, can be minimized by using efficient encodings of the data, and for such small fragments, cannot be used to reconstruct larger fragments [9].

Each tile component in the 3-SAT system handles at most a single bit of the input. Theoretically, this is sufficient for solving NP-complete problems; however, practically, handling more than a single bit of data at a time would amortize some of the overhead. Thus each tile component can be made to represent several bits. This transformation would result in a trade-off between privacy preservation and efficiency, as faster computation would reveal larger segments of the input to each node.

5.2 Efficiency and Scalability

One of the crucial questions about the tile style is whether it can be implemented efficiently enough to solve computationally intensive problems in a practical manner. At first glance, tile style's heavy use of the network may appear to make it less efficient than even a single computer. This intuition, however, is misleading. Examine a single node in an executing tile-style system and observe that it constantly performs computation: as soon as a node finishes replicating or recruiting with regard to a single tile, it moves on to the next tile. Thus, while there is some overhead to sending network messages, no node ever idly waits for the messages to arrive, but rather makes itself busy with other relevant computations. Because the problems the tile style targets are precisely the computationally intensive problems whose algorithms have a large number (exponential in the size of the input) of independent parallel threads, the nodes do not run out of subcomputations waiting to be executed. Since every node is constantly performing computation and never waiting for the communication, a network will perform faster than a single computer by a factor that is proportional to the size of the network and inversely proportional to the overhead of the tile style.

There are three ways to solve a highly parallelizable problem while preserving the data privacy: (1) on a large insecure network by using the tile style, (2) on a single private computer, or (3) on a small private network of trustworthy computers. We will first discuss the time needed to solve such a problem using the three methods in terms of the number of operations in Section 5.2.1, and then discuss the actual time necessary to solve problems in Section 5.2.2.

5.2.1 Analytical Evaluation

Suppose a network with N nodes uses the tile style to solve an n-variable m-clause 3-SAT problem. In expectation, the system has to explore 2^n crystals to reach a solution, and each crystal contains $(3m+n)\lg n$ replicated tiles (clear tiles in Figure 2) and no more than $3nm\lg^2 n$ recruited tiles (non-clear tiles in Figure 2). On average, each node will need to replicate $\frac{(3m+n)\lg n}{N}2^n$ tiles and recruit $\frac{3nm\lg^2 n}{N}2^n$ tiles. The replication procedure requires three distinct operations, as described in Section 4.4, each concluded by sending a single network packet; let the time for these operations be denoted as $3i$. Similarly, the recruitment procedure requires five operations, as described in Section 4.3, each also concluded by sending a single network packet; let the time for these operations be denoted as $5u$. Thus, the time required by each node is summarized by Equation (1). Note that this analysis is specific to 3-SAT, but the running times for other NP-complete problems will be very similar, since the fastest growing factor of 2^n will be the same.

$$\left(3i\frac{(3m+n)\lg n}{N} + 5u\frac{3nm\lg^2 n}{N}\right)2^n \tag{1}$$

$$2^n(n+3m)r \tag{2}$$

Now suppose a user wishes to solve the 3-SAT instance on a single computer. That computer would need to examine 2^n possible assignments, and check each n-variable assignment against the m clauses. Equation (2) describes the time this procedure would take using the most efficient available technique, assuming r is the amount of time each operation takes to execute: for each assignment, create a hash set containing the n literal-selection elements and check for each of the $3m$ literals whether or not the hash set contains that literal. The overhead of using the tile style over a single computer is the ratio of (1) and (2). Assuming $m > n$ and $i = u = r$, meaning that it takes roughly the same amount of time to perform each operation (e.g., looking up a value in a hash set and releasing a message on the network), the ratio is no greater than $\frac{8n\lg^2 n}{N}$. In other words, if the size of the public network exceeds $8n\lg^2 n$,

the tile style will execute faster than a single machine. For the sizes of problems we discuss in Section 5.2.2, that network size is several thousand nodes. Since the speed up on a tile-style system is linear in the size of the network, solving such a problem on a several-million-node network would execute 1000 times faster than a single-computer solution.

Finally, suppose a user wishes to solve the 3-SAT instance on a private network of M computers. Assuming the best possible distribution of computation and that the network communication is nonblocking, the time this system would require to solve the problem is no less than $\frac{2^n(n+3m)r}{M}$. In this case, the overhead of using the tile style over a private network is $\frac{8n \lg^2 nM}{N}$. In other words, if the size of the public network exceeds $8n \lg^2 nM$, the tile style will execute faster than the private network.

5.2.2 Numerical Evaluation

We have implemented Mahjong [1], a Java-based distributed software system whose architectural style is faithful to the descriptions of the algorithms of the tile style (recall Section 4) to (1) verify the correctness of the tile style, (2) verify that the speed of a tile-style system scales linearly with the size of the network, and (3) measure the above-defined execution-time constants i, u, and r. To build Mahjong, we leveraged Prism-MW [19], a middleware platform intended specifically for style-driven implementation of software architectures in highly-distributed and resource-constrained environments. The tool takes a user-provided description of the set of tiles for an NP-complete problem and the input to the computation (the tiles for solving two NP-complete problems, $SubsetSum$ and 3-SAT, are included) and automates the remaining steps of building a distributed tile-style system.

We have performed a number of empirical measurements of Mahjong in solving $SubsetSum$ and 3-SAT problems on dedicated networks as large as 186 nodes. These evaluations are preliminary and are only intended to demonstrate that tile-style systems do in fact solve NP-complete problems, show the speed up trends on growing networks, and measure execution-time constants. While we are in the process of a large-scale empirical investigation that involves deploying tile-style systems on much larger public networks, that ongoing work is a substantial effort of its own and is not the focus of this paper. Here, we only wish to summarize our current findings to support our analytical results:

1. To verify the correctness of the tile-style algorithms, we solved several $SubsetSum$ and 3-SAT problems (up to 32 bits in size). We chose the sizes of instances to each execute in under 4 hours on our relatively small 186-node network. We verified that Mahjong found the correct solution to each instance, that it sent no unexpected communication between nodes, and that no node produced undesired connections between tiles. Further, we tested Mahjong on inputs that returned a negative answer. As expected, it executed indefinitely.

2. In a series of experiments varying the size of the underlying network upto 186 nodes, we found that doubling the size of the network consistently decreased the computation time by a factor of 1.9. While our theoretical analysis predicts that factor to be 2, we hypothesize that the slight inefficiency comes from our constant underlying-network bandwidth and the increased load; by contrast, increasing the size of a global network is likely to add communication pathways and increase overall bandwidth. The results provide confirmation that the speed of a tile-style system is proportional to the size of the network. This provides a desirable scaling trend for large networks.

3. We measured the constants r, i, and u on a 2.4 GHz machine running Windows XP and Sun JDK 6.0 by executing several million benchmark tests and averaging their running times. We found that $r \approx 3.6 \times 10^{-7}$ seconds ($\approx$ 2.8 MHz), $i \approx 2.8 \times 10^{-7}$ seconds ($\approx$ 3.8 MHz), and $u \approx 4.1 \times 10^{-7}$ seconds ($\approx$ 2.4 MHz). With these measurements and Equations (1) and (2), we can estimate the speeds of a tile-style system and a single computer solving a given NP-complete problem. For example, solving a 38-variable, 100-clause instance on a single computer would take 3.3×10^7 seconds $\approx$ 1 year. However, the same problem could be solved using the tile style on a million-node network in 1.8×10^5 seconds $\approx$ 2.1 days.

Figure 6 compares the execution times of tile-style and single-computer solutions. For each of the three depicted 100-clause 3-SAT instances (with 30, 40, and 50 variables), the graph shows the horizontal line indicating the running time of a single-computer solution, and the diagonal line indicating the running time of a tile-style system implemented in Mahjong and deployed on networks of varying sizes. For networks larger than about 4000 nodes, the tile-style solutions outperform their competitors; for extremely large networks the tile systems are much faster. For example, solving the 40-variable, 100-clause 3-SAT problem on a single computer would take 4 years, while doing so using the tile-style solution implemented in Mahjong and deployed on the network the size of SETI@home (1.8 million nodes [24]) would take 7 days.

6 Contributions

We developed a new architectural style, called the tile style, as a technique for distributing computation over a network. The tile style provides the opportunity to design large distributed computing systems without having to worry about distribution, privacy preservation, fault and adversary tolerance, and scalability, as those properties are

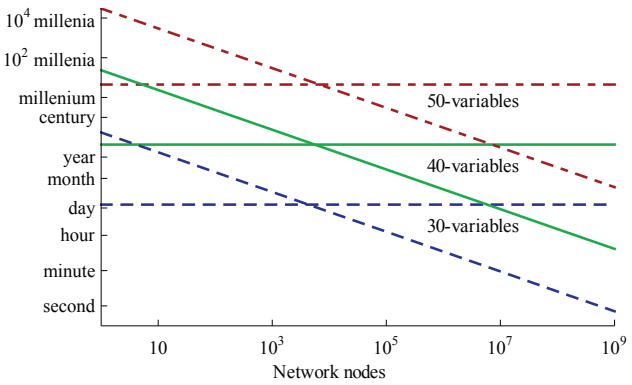

Figure 6. Expected running times for single-computer (horizontal lines) and tile-style (diagonal lines) solutions for 30-, 40-, and 50-variable, 100-clause 3-SAT problems on varying-size networks. For networks larger than about 4000 nodes, the tile-style solutions outperform their competitors.

inherent to the style. We presented a rigorous theoretical analysis of the style and formally proved that the resulting systems are efficient and scalable and preserve privacy as long as no adversary controls half of the public network.

We adapted an off-the-shelf middleware platform to produce Mahjong, a reusable implementation of the tile style, and built distributed systems to solve two NP-complete problems, $SubsetSum$ and 3-SAT. An initial empirical evaluation of the tile style verified that the style can be used to solve NP-complete problems, that our algorithms result in correct computations, and that as the underlying network grows, the computation time decreases inversely proportionally to the size of the network. For networks larger than about 4000 nodes, the tile style outperforms existing alternatives.

References

[1] Mahjong tile style implementation. `http://csse.usc.edu/~ybrun/Mahjong`.

[2] B. Berger and T. Leighton. Protein folding in the hydrophobic-hydrophilic (HP) is NP-complete. In *Proc. of the 2nd Annual International Conference on Computational Molecular Biology*, pages 30–39, 1998.

[3] Y. Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Computer Science*, 378(1):17–31, 2007.

[4] Y. Brun. Nondeterministic polynomial time factoring in the tile assembly model. *Theoretical Computer Science*, 395(1):3–23, 2008.

[5] Y. Brun. Solving NP-complete problems in the tile assembly model. *Theoretical Computer Science*, 395(1):31–46, 2008.

[6] Y. Brun. Solving satisfiability in the tile assembly model with a constant-size tileset. *J. of Algorithms*, In Press, 2008.

[7] Y. Brun and N. Medvidovic. An architectural style for solving computationally intensive problems on large networks. In *Proc. of Software Engineering for Adaptive and Self-Managing Systems*, 2007.

[8] Y. Brun and N. Medvidovic. Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In *Proc. of the 2nd International Workshop on Engineering Fault Tolerant Systems*, pages 38–43, 2007.

[9] M. Chaisson, P. Pevzner, and H. Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[10] A. M. Childs. Secure assisted quantum computation. *Quantum Information and Computation*, 5(456), 2005.

[11] D. Dagon, G. Gu, C. Lee, and W. Lee. A taxonomy of botnet structures. In *Proc. of the 23rd Annual Computer Security Applications Conference*, pages 325–339, 2007.

[12] P. T. Devanbu and S. Stubblebine. *Software Engineering for Security: A Roadmap*, pages 225–239. ACM Press, 2000.

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. of High Performance Computing Appl.*, 15(3):200–222, 2001.

[14] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. of the ACM*, 38(3):690–728, 1991.

[15] D. Keysers and W. Unge. Elastic image matching is NP-complete. *Pattern Recognition Letters*, 24:445–453, 2003.

[16] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home — massively distributed computing for SETI. *IEEE MultiMedia*, 3(1):78–83, 1996.

[17] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Proc. of ICSE*, pages 259–268, 2007.

[18] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande. *Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology.* Horizon Press, 2002.

[19] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE TSE*, 31(3):256–272, 2005.

[20] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *Proc. of the 1st Workshop on Self-Healing Systems*, 2002.

[21] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[22] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[23] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, 2008.

[24] Wikipedia. Seti@home. `http://en.wikipedia.org/wiki/SETI@home`, 2008.

[25] E. Winfree. Simulations of computing by self-assembly of DNA. Technical Report CS-TR:1998:22, Caltech, 1998.

[26] A. C.-C. Yao. How to generate and exchange secrets. In *Proc. of FOCS*, pages 162–167, 1986.