Preserving Privacy in Distributed Systems

Yuriy Brun and Nenad Medvidovic, *Member, IEEE Computer Society*

Abstract—We present sTile, a technique for distributing trust-needing computation onto insecure networks, while providing probabilistic guarantees that malicious agents that compromise parts of the network cannot learn private data. With sTile, we explore the fundamental cost of achieving privacy through data distribution and bound how much less efficient a privacy-preserving system is than a non-private one. While that cost is significant, we find that sTile-based systems execute orders of magnitude faster than homomorphic encryption systems, the alternative promising approach to preserving privacy. This paper focuses specifically on NP-complete problems and demonstrates how sTile-based systems can solve important real-world problems, such as protein folding, image recognition, and resource allocation. We present the algorithms involved in sTile and formally prove that sTile-based systems preserve privacy. We develop a reference sTile-based implementation and empirically evaluate it on several physical networks of varying sizes, including the globally distributed PlanetLab testbed. Our analysis demonstrates sTile's scalability and ability to handle varying network delay, as well as verifies that problems requiring privacy-preservation can be solved using sTile orders of magnitude faster than using today's state-of-the-art alternatives.

1 Introduction

Recent advances in Internet technology, such as cloud computing, are forcing the nature of computation to evolve. For example, users no longer run computations on private machines or machines of which they have physical awareness. The same evolution has taken place for data storage: many users no longer keep data, such as email, on their machines, but rather allow "the cloud" to maintain and safeguard that data. This transformation has allowed ubiquitous access to computation and data with higher availability and reliability than possible with personal machines and local servers. Simultaneously, however, this transformation has created new challenges in computing. This paper addresses the challenge of executing computations on untrusted machines in a trustworthy manner.

The rapid evolution of how our systems execute and how our data is handled has affected the meanings of the terms *security* and *privacy*, when referring to software systems. Security of a computation used to mean "the computer running the computation should be protected from malicious compromise," and privacy of data used to imply "unauthorized entities could not gain access to the data." Today, however, with computations on private data running on remote, unknown, potentially untrusted machines, security should mean "no computer, including the one(s) running the computation, may undetectably compromise it" and privacy should imply "no entity, including the one(s) executing the computation, should gain access to the data." Cyber systems, such as clouds, have not yet embraced these new definitions, largely due to the intellectual hurdles and costs of developing systems that conform to these high standards.

- *Y. Brun is with Computer Science & Engineering, University of Washington, Seattle, WA, 98195. E-mail: brun@cs.washington.edu*
- *N. Medvidovic is with the Computer Science Department, University of Southern California, Los Angeles, California, 90089. E-mail: neno@usc.edu*

To provide a common example, many of us rely on Google to deliver, maintain, properly replicate, etc. our email, but the notion that it may be possible to do that without Google having access to our email seems unreasonable by today's practices. Instead, we rely on legal contracts and promises to guarantee that Google will not abuse its privileged access to our data. While this may be acceptable for companies with "a lot to lose," such as Google, the key question is whether we are willing to accept and trust smaller companies that may, for example, compute our taxes, calculate suggestions for future purchases based on our past behavior, and mine our social networks to introduce us to new friends, potential employees, and businesses that may benefit us? Some of these services may be provided by nothing more than a desktop in someone's garage. Even when we only allow well-respected and trusted companies to have access to our data and computation, over $50 billion are still lost each year through identify theft perpetrated by either pretending to be a reputable company or simply relying on a user's trust in an unknown service [34].

In this paper, we present sTile, a technique that is based on a theoretical model of self-assembly and consists of (1) a software architecture, (2) a set of algorithms, and (3) an implementation framework. sTile allows building software systems that distribute a particular large and important class of computations onto an untrusted network while providing guarantees that the computers that execute the computation cannot learn the computation's private data. The work described in this paper has a specific focus: it is aimed at large, distributed computational systems solving NP-complete problems, such as protein folding and image recognition. However, the ideas we present are a foundation for follow-on techniques to privately distribute arbitrary computations such as email and tax calculations.

sTile-based systems offer a trade-off between privacy and efficiency. In other words, if a domain does not require privacy, the same problems can be solved more efficiency using non-privacy-preserving methods. There exists a fundamental cost of achieving privacy through data distribution. In this paper, we explore this cost and bound it by producing a reference

implementation and executing it on several distributed testbed networks. In some sense, we push the idea of distribution for privacy to its limit, exposing no more than 2 bits of data to each distributed unit of computation. This implementation represents an upper bound on the cost and we envision that future work can improve this bound. In a number of ways, our technique is similar to homomorphic encryption [29]. Both provide probabilistic privacy guarantees with exponential drop-offs in the likelihood of a malicious party compromising private data. However, homomorphic encryption provides such guarantees even when the computation is performed by a single node, whereas sTile is explicitly geared toward the distributed setting and requires that fewer than half of the network nodes are compromised. The restriction that no adversary controls more than half of the network allows sTile to outperform homomorphic encryption, executing computations several orders of magnitude faster. In fact, while we have deployed sTile-based systems on real-world networks, to date, the efficiency issue has prevented deployment of homomorphic encryption systems.

Existing approaches to using the Internet's computational resources to perform NP-complete computations [10], [22], [25], [32] have resulted in commercial enterprises deployed on over a million machines [36], [39], [49], providing results to some of the largest computational problems solved to date in the areas of artificial intelligence [20], physics [38], and neuroscience [19]. However, these approaches have assumed reliable and trustworthy underlying networks, and employed only rudimentary fault-tolerance and privacy safeguards, rendering them useful only (1) for rich companies that own their own, large, trustworthy clusters, and (2) in research but not for wide-scale commercial applications. In contrast, our approach brings the privacy and security to the forefront and allows distribution of private data onto untrusted machines while preserving the data's privacy, with high probability. In Section 6, we will, in detail, compare sTile to existing, well-known approaches, such as BOINC [3], which includes SETI@home [36] and Folding@home [39], MapReduce [25], and homomorphic encryption [29]. While sTile can compete with existing techniques in a number of dimensions, in this paper, we focus on *scalability* and *efficiency*, while demonstrating sTile's unique ability to guarantee a quantified, high probability of *data privacy*. We have previously discussed sTile's ability to handle faults and malicious attacks [17], [18], and do not focus on that dimension here.

We formally analyze and prove that sTile-based systems preserve the privacy of the data used in the computation as long as no adversary controls more than one half of the public network — a highly unlikely occurrence. Furthermore, in order to demonstrate the computational feasibility of our solution, sTile includes Mahjong, an implementation framework realized on top of a middleware platform. We deploy Mahjong-based systems on three distinct networks, including the globally distributed PlanetLab testbed [43]. We also simulate a sTile-based system's execution on virtual networks of up to 1,000,000 nodes. We empirically verify that the speed of sTile computation is proportional to the number of nodes and that network delay has little-to-no effect on that speed. Finally, we formally analyze the communication and computation costs induced by sTile, provide bounds on their time requirements, and then use the Mahjong-based implementations to verify them empirically.

The rest of this paper is structured as follows: Section 2 elaborates on the paper's scope by presenting several problems sTile-based systems target. Section 3 describes sTile, its architectural underpinnings, and the associated algorithms. Section 4 discusses our Mahjong-based implementations and a set of experiments designed to demonstrate sTile's feasibility. Section 5 formally analyzes sTile's privacy-preservation. Section 6 positions our work in terms of related research. Finally, Section 7 summarizes the paper.

2 Target Problems and Research Scope

To highlight our research contributions, in this section we present two representative problems our technique helps to solve and then summarize the scope of our work.

2.1 Target Problems

Existing techniques that use the Internet to solve complex computationally-intensive problems [25], [36], [39] do not take into account that data involved in these computations may be sensitive and that network nodes may be malicious and may attempt to extract private data from the computation. In fact, these techniques make it trivial to learn the nature of the computation and the relevant data. By contrast, our primary objective behind sTile is *privacy-preservation*: the involved data must remain private during and after the computation. The following are two example scenarios sTile 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 [8], 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 [39] to distribute the computation on a public network because these approaches 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 captured by 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 [35] and many people may enter the location of interest at once. Further, any employed solution must execute quickly to deliver results in real-time, requiring far more resources during peak times than off-peak times, making this problem ideally suited for cloud computing. 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 private network, or use a privacy-preserving technique to distribute the computation on a public network.

2.2 Research Scope

The goal of this research is to provide an approach for leveraging distributed, inherently untrustworthy computational resources into a highly robust distributed software system. In particular, the resulting software system benefits from the Internet's large size while outperforming existing techniques by tolerating the insecurity and unreliability inherent to the Internet. The primary contribution of this paper is *sTile*: a software system-construction technique based on a theoretical model of self-assembly. sTile comprises

- an assembly of a set of tile types that solves a particular computational problem,
- a software architecture that reifies that assembly,
- a set of algorithms for instantiating and distributing that architecture on a set of networked nodes, and
- an implementation framework that natively realizes the architecture and algorithms.

sTile allows building software systems that distribute NP-complete problems onto large networks while (a) preserving the privacy of the data, (b) scaling well to leverage the resources, and (c) tolerating faulty and malicious nodes. In this paper, we provide a formal theoretical analysis of sTile's privacy preservation, efficiency, and scalability. Furthermore, to illustrate the feasibility of sTile, we also describe a reference implementation of the sTile framework. Finally, we deploy this implementation on a wide range of distributed testbeds, including the globally distributed PlanetLab, to empirically evaluate sTile.

It is important to note that we have designated several directions of research as outside of the scope of this paper. First, we do not consider issues that are common to all Internet-based systems. For example, the energy consumption of an Internet-sized computing network is important, but is not unique to our approach and has been argued non-prohibitive in volunteer computing literature [3]. Second, we do not focus on programming models and language-level solutions that can be envisioned as natural outgrowths of this work. In particular, it is relevant to discuss a tile-based language that would expand sTile's application outside of NP-complete problems, but we leave this direction as part of our future work. Third, while we focus on scalability, efficiency, and privacy, there are certainly other relevant dimensions of Internet-sized computing systems (e.g., energy efficiency, which has been mentioned above, as well as fault tolerance, which we have discussed in [17], [18]), but we consider them outside of this paper's scope. Where applicable, we will provide the intuition for sTile's ability to compete with existing techniques in these dimensions.

3 sTile

sTile is a technique for designing, implementing, and deploying software systems that distribute computation onto large, insecure, public networks. Figure 1 shows a high-level overview of sTile. sTile consists of four components: a tile assembly, the corresponding tile architecture, the associated algorithms, and the Mahjong implementation framework. For a given problem *P*, sTile uses a tile assembly (described in Section 3.1) to create a tile architecture (described in Section 3.2). sTile uses

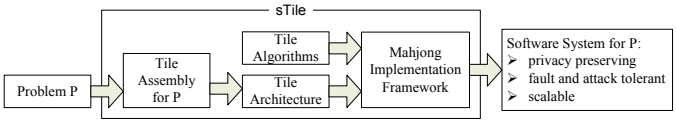

Fig. 1. A high-level overview of sTile.

that tile architecture and the tile algorithms (also described in Section 3.2) to compile a Mahjong-based implementation of the system (described in Section 3.3), which is the software system used to distribute the computation solving *P* onto a network, in a privacy-preserving manner.

It is important to point out that a software engineer who wishes to develop and deploy a sTile-based system does not need to understand the underlying computational model that we describe in Section 3.1 and use throughout this paper. In essence, sTile includes a compilation procedure that allows the engineer to automatically compile a computational problem to a sTile-based, but otherwise completely conventional-looking, software system. The underlying tile model is important for proving many of the properties of system correctness and privacy preservation, and we describe them all in this paper. However, from the point of view of the developer, these details are abstracted away, e.g., much like the assembly language that executes underneath a program written in C++.

3.1 Computing with Tiles

A key component of sTile is the tile architecture, which is based on a tile assembly. Tile assemblies are extensively studied mathematical objects [1], [9], [44], [48], [50]. Our own prior work has developed the notion of efficient computation with tile assemblies and constructed efficient assemblies to add and multiply integers [12], factor integers [13], and solve NP-complete problems [14], [15]. The tile assembly model is Turing-universal, so tile assemblies can implement all algorithms [9], [44].

Tile assemblies are not the focus of this paper: we concentrate here on building software systems that solve computational problems on large networks. To that end, we leverage existing tile assemblies, such as our previous work on NP-complete computation. Thus, we will only briefly describe tile assemblies here, before going into the details of the tile architecture and tile algorithms. For completeness, we formally describe the tile assembly model and the tile assembly we developed to solve 3-*SAT* [15] in the Appendix.

A tile assembly is a set of types of tiles, squares with static labels on their four sides. The tiles are not allowed to rotate but are allowed to "float" around and attach to one another under fairly simple rules. For the purposes of this discussion, tiles may attach when the labels on their respective sides match. The key to having a tile assembly do something useful is properly choosing which tile types to include in the assembly. The set of tile types encodes the "program" the assembly will "execute." Again, we do not focus on this process here (a more complete description of the process of developing tile assemblies to solve computational problems can be found in [15]), but one can imagine that design of tile assemblies

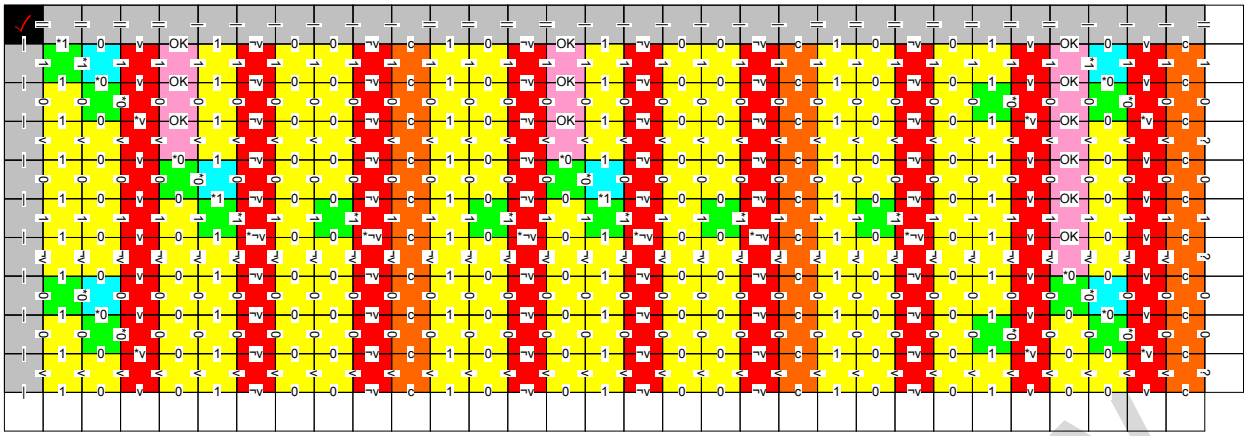

Fig. 2. An example crystal of a 3-*SAT*-solving tile assembly. Here, the clear tiles encode the input Boolean formula $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$. This crystal represents one possible nondeterministic execution that checks whether the assignment $x_0 = x_2 = TRUE$ and $x_1 = FALSE$ satisfies ϕ. Because it does, the ✓ tile attaches in the top left corner. The Appendix describes the details of the tile assembly model and this 3-*SAT*-solving tile assembly.

is not unlike the design of cellular automata systems or even programming in assembly language.

Figure 2 shows one possible example execution (crystal) of a tile assembly that solves 3-*SAT*. The Appendix describes the details of this assembly. The tiles of a tile assembly create many such crystals, each nondeterministically exploring whether an assignment satisfies a 3-*SAT* Boolean formula. If some crystal, such as the one in Figure 2, finds a satisfying assignment, the ✓ tile attaches in its top left corner, indicating that a solution has been found. In Section 3.2, we describe how sTile uses a network of computers to essentially simulate tile assemblies, resulting in a software system that distributes computation to solve computationally-intensive problems. In subsequent sections, we will argue that such systems are efficient and possess properties that are important to software systems deployed on large public networks such as the Internet.

3.2 Tile Architecture and Algorithms

A sTile-based system is a software system that uses a network of computers to solve a computational problem. Intuitively, the network will simulate a tile assembly: each computer on the network will pretend to be a tile (many tiles, actually), and communicate with other computers to self-assemble a solution to a computational problem. Each computer will deploy tile components, each representing a tile in a tile assembly, and facilitate the proper communication channels and algorithms to allow the tile components to self-assemble. Thus, a tile architecture is based on a tile assembly; the software system employing that architecture solves the particular computational problem the tile assembly solves.

From the software system developer's point of view, building a sTile-based system may seem imposing. However, it is actually simple. In essence, sTile contains the entire processes and all the necessary tools to abstract the internal tile representation away from the developer, so that the developer never needs to worry about tile assemblies or to understand how tiles work. The key to this abstraction is compilation. We now briefly describe the two possible processes a developer may follow to create a sTile-based system: the first requires a deep understanding of tile assemblies and the second (the one sTile employs) requires none.

Since a tile architecture is based on a tile assembly, and a sTile-based system solves the same computational problem the underlying tile assembly solves, one way to build a sTile-based system to solve a particular computational problem P is to develop a tile assembly $\mathbb{P}$ that solves P, then follow the procedures we describe below to translate $\mathbb{P}$ into an architecture, and finally implement a software system by employing that architecture and the appropriate algorithms. This process can be quite painstaking and requires the design of a tile assembly, such as the one we describe in the Appendix.

Instead, we recommend an automated compilation procedure that allows the developer to create a sTile-based system without ever understanding the details of tile assemblies. Since we have developed several tile assemblies that solve NP-complete problems [14], [15], and since NP-complete problems are polynomially related, it is possible to translate all NP problems to the problems for which we already created tile assemblies using one of the well-established reductions [46]. This translation becomes a compilation of the problem for which the developer wishes to build a sTile-based system to a known tile assembly. This compilation, as virtually all compilations, may result in less efficient systems than the direct approach of developing a tile assembly for each particular problem. However, the benefits of automated compilation are numerous, and include a significant time and cost savings and a lower likelihood of bugs.

We will not focus on the well-studied compilation process [46] in this paper. We do, however, note three facts: (1) The time the compilation takes is insignificant, as compared to actually solving the NP-complete problems, such as 3-*SAT*. (2) Even without compilation, our sTile-based systems present a considerable contribution because solving 3-*SAT* itself has important implications for real-world systems [45]. And, (3) in addition to ease of use, compilation masks the problem the user is solving; while this is a beneficial side effect of

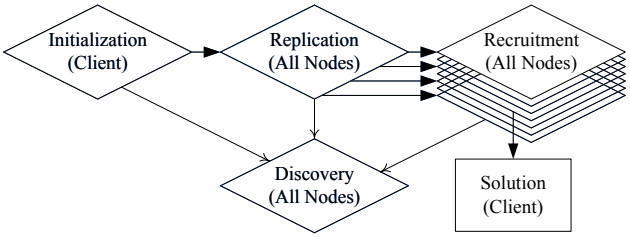

Fig. 3. Overview of tile architecture node operations.

using sTile, we discuss the much more important privacy-preservation property in Section 5.

We now describe what a tile architecture looks like and how it is based on a tile assembly. The components of the tile architecture are instantiations of the tile types of the underlying assembly. A sTile-based system employing such an architecture will have a large number of components; on the other hand, 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, thus dynamically completing the architectural configuration [47] corresponding to a tile crystal. The components recruit other components, by sampling nodes until they find one whose interfaces match. Note that many components in the sTile architecture can run on a single physical node (i.e., machine on a network), as we will further elaborate below.

In addition to defining the tile types, a tile assembly also directs sTile how to encode the input to the computation into the set of components comprising the initial, partial architectural configuration. The input consists of a seed crystal, such as the clear tiles along the right and bottom edges in Figure 2 (also shown separately in Figure 11a). Figure 3 summarizes the algorithms a sTile-based system follows to find a solution. During *initialization*, the system sets up a single seed crystal (i.e., partial sTyle architectural configuration) on the network to encode the input. The seed then *replicates* to create many copies, and each of the copies *recruits* tiles to assemble larger crystals (i.e., to complete the architectural configuration corresponding to each crystal) 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. Note that the nodes perform these operations autonomously, without central control, in essence self-assembling the sTile architecture and, by extension, the computation the architecture is intended to solve.

We elaborate on these operations next. We also discuss what happens when sTile is unable to find a solution to a computational problem (e.g., when no assignment of variables can satisfy 3-*SAT*).

3.2.1 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.

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 unique identifier (e.g., IP or MAC address) 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 the Appendix, 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.

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 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 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. The well-known *coupon collector* problem [41] indicates that 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.

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 $\Theta(\log N)$ [41], 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 [23]. It is important to note that while the tile architecture targets large networks, it can be made to work on small networks, although with weaker privacy guarantees. However, these small networks are not the central concern of this work.

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 3.2.2), 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.

3.2.2 Discovery

The node discovery algorithm is central to sTile because initialization, replication, and recruitment all use it. 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 interfaces. 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 [41]. 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.

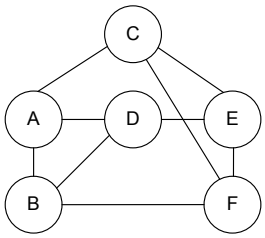

Fig. 4. A network with six nodes. We assume that every node in our underlying network has p neighbors (here $p = 3$).

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 an N-node network, 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 lemma 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 derived 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 [41], 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 [41] 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.

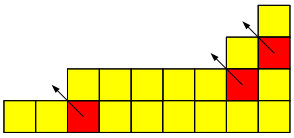

Fig. 5. Tile components that have both an upper and a left neighbor (highlighted in the diagram) can recruit new components to attach to their upper left.

3.2.3 Recruitment

The seed crystal grows into a full assembly by recruiting tile attachments. In a computational tile assembly (such as the assembly described in the Appendix that solves 3-*SAT*), a tile component that has both an upper and a left neighbor recruits a new tile to attach to its upper left. 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 3.2.2, and sends it an attachment request. An attachment request consists of X's upper neighbor's left interface and X's left neighbor's top interface. If those interfaces match Y's right and bottom 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 [52], which is crucial for privacy preservation.

Each component's recruitment is a five-step process: X asks N (its upper neighbor) to encode its left interface, N asks W (X's left neighbor) to encode its top interface, 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 4.5 when we compute the speed of sTile-based systems.

In the 3-*SAT* example from Section 3.1, the successful crystal recruits 310 tile components (non-clear tiles in Figure 2). An unsuccessful crystal, which we discuss further in Section 3.2.5 can recruit fewer, but no more than 310 tiles.

3.2.4 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 3.2.2, 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 (two 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 is thus 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 4.5 when we compute the speed of sTile tile architecture.

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. Since the diameter of a large connected network of N nodes with randomly distributed connections is $\Theta(\log N)$ [41], the "STOP" message will propagate in $\Theta(\log N)$ time.

3.2.5 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. For the assembly execution that solves a 3-variable 3-*SAT* problem from Figure 2, 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 KB), there is little reason why even a single computer could not deploy that many components.

3.3 Mahjong Implementation Framework

The final element of sTile is the Mahjong implementation framework which uses the tile architecture and algorithms to automatically compose a sTile-based software system.

The Mahjong framework [11] is realized as a Java-based middleware platform that faithfully implements the tile architecture and its algorithms. It takes as input a description

of a tile assembly, implements a software system using the tile architecture based on that assembly and employing the algorithms described in Section 3.2, and outputs (i.e., deploys) a complete sTile-based software system. The Mahjong implementation framework is available for download [11].

In Mahjong's implementation we leveraged Prism-MW [40], a middleware platform intended specifically for architecture-aware implementations in highly-distributed and resource-constrained environments. Prism-MW provides explicit implementation-level constructs for declaring components, interfaces, interactions, network communication, etc., as well as the ability to encode architectural constraints as first-class middleware-level constructs. Each node on a network runs a Prism-MW Architecture, which forms a "sandbox" within which all of the Prism-MW (and, in our case, Mahjong) code deployed on a given node executes. The use of system resources on each participating hardware host is hence restricted and can be released at any time. The tile components deploy inside the Architecture objects and perform their functionality as part of the tile algorithms via their interfaces, which are implemented as Prism-MW Ports. Mahjong 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 sTile-based system. Mahjong consists of 29 objects and 2,900 lines of Java code. It has proven to be flexible and robust to variations in the various networks on which we have deployed it to date: we have not had to make changes to the source code to adapt Mahjong to distinct networks and a developer unfamiliar with the project was able to deploy it without consultation with us.

4 Computational Feasibility

In order to demonstrate that sTile is a feasible solution for building software systems that distribute computationally intensive problems on very large networks, we must show that (1) such systems' computational speed is proportional to the size of the underlying network, (2) such systems are robust to network delay, and (3) real-world-sized problems can be solved on real-world-sized networks in reasonable time.

To that end, we have built two Mahjong-based implementations and a simulator-based Simjong, three software systems that employ sTile to solve NP-complete problems. The Mahjong-based implementation distribute computation onto computers on a physical network. Simjong is a discrete-event simulator that creates a simulated network of virtual nodes and distributes computation onto that simulated network while employing accurate models of network message delays. In addition to empirically illustrating the above three properties, the Mahjong-based implementations establish the correctness of our algorithms and demonstrate the distribution of a sTile-based system onto a physical network. Because access to Internet-sized networks presents numerous logistical challenges, (the largest physical distributions we could secure are 186 collocated nodes and 100 globally distributed nodes, as part of PlanetLab [43]), Simjong's goal is to accurately simulate sTile-based system distributions on very large networks (e.g., 1,000,000 nodes).

We present the details of the Mahjong-based implementations and Simjong in Section 4.1 and our experimental setup in Section 4.2. We then discuss our experiments testing the scalability, robustness to network delay, and efficiency in Sections 4.3, 4.4, and 4.5, respectively. Finally, we will summarize some of the potential threats to the validity of our evaluation in Section 4.6.

4.1 sTile-based Implementations

The Mahjong-based implementations are instances of the Mahjong implementation framework. We have built two Mahjong-based implementations, one for each of the 3-*SAT* and *SubsetSum* problems. These implementations are available for download as part of the framework [11].

Simjong [11] is a Java-based discrete-event simulator with network-delay simulation capabilities. Simjong executes on a single machine and creates a user-specified number of virtual hardware Node components, each capable of deploying tiles. A central Clock component keeps track of virtual time and allows each Node to execute one instruction per clock cycle. Whenever a Node's tile needs to communicate to another Node's tile, it sends a message via the Network component that determines the delay for that message's delivery. Simjong's network model allows for message delivery time to be constant (e.g., 100ms), chosen at random from some distribution (e.g., Gaussian around 100ms with a 20ms standard deviation), or proportional to the geographic distance between locations assigned to each virtual node (e.g., 50ms between Amsterdam and London, and 500ms between New York and Hong Kong). Simjong's network model is a simplification of the network simulator standard ns-2 [27] because it abstracts away the exact topology of the network, which is not important for our needs.

Simjong's virtual nodes follow the tile algorithms, simulating a deployment of a sTile-based system on a real distributed network. While executing, Simjong keeps track of the number of completed seeds and reports its progress. Thus, it is possible to use Simjong to estimate the time required for a computation to complete after executing only a fraction of that computation. This is an important characteristic that allows us to simulate very large problems on very large networks in comparatively short time.

4.2 Experimental Setup

We leveraged three distributed networks for our experimental evaluation: (1) A private heterogeneous cluster of 11 Pentium 4 1.5GHz nodes with 512MB of RAM, running Windows XP or 2000. (2) A 186-node subset of USC's Pentium 4 Xeon 3GHz High Performance Computing and Communications (HPCC) cluster [33], whose nodes were distributed in several locations, but all within one city. (3) A 100-node subset of PlanetLab [43], a globally distributed network of machines of varying speeds and resources that were often heavily loaded by several experiments at a time.

The cross-section of data we present in this paper used four representative instances of NP-complete problems, to which we will refer by their labels:

$\mathfrak{A}$: 5-number 21-bit *SubsetSum* problem,
$\mathfrak{B}$: 11-number 28-bit *SubsetSum* problem,
$\mathfrak{C}$: 20-variable 20-clause 3-*SAT* problem, and
$\mathfrak{D}$: 33-variable 100-clause 3-*SAT* problem.

Our experimental goals were to verify sTile's scalability with respect to network size and robustness to network delay. Our experiments had three independent variables — the number of nodes, the network communication speed between nodes, and the size of the NP-complete problem — and one dependent variable — the time the computation took to complete.

First, to verify the correctness of sTile-based systems, we used Mahjong-based implementations to solve over one hundred of *SubsetSum* and 3-*SAT* problems, including $\mathfrak{A}$, $\mathfrak{B}$, and $\mathfrak{C}$. As a rule of thumb, we chose the sizes of problem instances to each execute in under 4 hours on our 186-node cluster. We verified that, on each of the above three networks, the implementations found the correct solution to each instance, it sent no unexpected communication between network nodes, and no node produced undesired connections between tile components. Further, we verified that, when provided inputs with a negative answer, the implementations appeared to execute indefinitely, as expected.

We were able to perform experiments with Mahjong-based implementations on networks of up to 186 nodes and with Simjong on virtual networks of up to 1,000,000 nodes. The need for simulation arose from the difficulty of obtaining time on dedicated machines. Today's distributed-system testbeds are (1) much smaller than the networks sTile targets, (2) not aimed at computationally intensive (as opposed to data-sharing or threat-prevention) techniques, and (3) under heavy load. For example, PlanetLab, distributed at 485 locations around the world, consists of only 1090 nodes[1] of which almost half are typically unresponsive; of the responsive nodes, some are heavily overloaded or exceedingly slow. Because of these well known issues with PlanetLab [26], we were unable to repeat our experiments as many times as on the other networks. In the end, PlanetLab allowed us to demonstrate that sTile-based systems can be successfully deployed on globally distributed networks and provided us with useful numerical trends.

4.3 Scalability

To verify that the speed of the computation is proportional to the number of nodes on the underlying network, for each of the three networks described above, we deployed Mahjong-based implementations on the entire network and on randomly selected halves of the network. We varied the size of the problem and measured the average time in which the implementations found the solution over 20 executions (except on PlanetLab, as we explained above). We then also deployed Simjong on virtual networks of increasing size from 125,000 to 1,000,000 nodes (with a constant network delay of 100ms for all packets) and simulated the first 10^{-4}% of the seeds to estimate the time required to complete the entire computation. This allowed each Simjong execution to complete in about an hour of actual time, while executing a sufficiently large number of seeds. Our measurements have shown that, after the first few thousand seeds, our implementations make fairly constant progress through the seeds and that extrapolating from the 10^{-4}% fraction is accurate.

1. By comparison, DETER [7] users can gain access to only a subset of 300 closely located physical nodes that rely on a network delay simulator similar to the one Simjong uses.

Network & Problem	Number of Nodes	Execution Time	Speed-up Ratio
Private Cluster	5	43.2 sec.	
$\mathfrak{A}$	10	22.9 sec.	1.89
HPCC	93	220 min.	
$\mathfrak{C}$	186	116 min.	1.90
PlanetLab	50	9.2 min.	
$\mathfrak{B}$	100	4.8 min.	1.92
Simjong	125,000	8.7 hours	
	250,000	4.5 hours	1.93
	500,000	2.1 hours	2.14
$\mathfrak{D}$	1,000,000	64 min.	1.97

Fig. 6. The effect of doubling the network size on the system's execution time. The speed-up ratio is the factor speed improvement over the network of half the size.

We hypothesized that as we double the size of the underlying network, the Mahjong-based implementations and Simjong would take approximately half as much time to complete. Figure 6 shows a cross-section of the results of our scalability experiments, which confirm this hypothesis. For example, executing $\mathfrak{D}$ on a 1,000,000-node virtual network took a factor of 1.97 less time than on a 500,000-node virtual network. We speculate that the slight inefficiency on the physical networks (1.9 instead of 2) comes from the constant underlying-network bandwidth; by contrast, increasing the size of a global network is likely to add communication pathways and increase overall bandwidth. The experimental results provide confirmation that the speed of a sTile-based system is proportional to the size of the network, resulting in a desirable scaling trend for large networks.

4.4 Robustness to Network Delay

To measure the effect of network delay on the speed of sTile-based systems, we compared the times Mahjong-based implementations took to solve the same problems on equal-sized subsets (up to the maximum 11 nodes) of the private and HPCC clusters and PlanetLab. We then compared the times Simjong took on five virtual networks of 1,000,000 nodes each, with respective network delays of 0ms, 10ms, 100ms, 500ms, drawn from a Gaussian distribution around 100ms with a 20ms standard deviation, and ones proportional to the geographic distance between randomly assigned world-wide locations (varying from 20ms for collocated nodes to 500ms for most distant ones). We again simulated the first 10^{-4}% of the seeds for the same computation on each of those networks to estimate the time required to complete the entire computation.

We hypothesized that the network delay will have virtually no effect on the time sTile-based systems take to execute. The intuition behind this hypothesis is that each node in a sTile-based system handles the deployment of thousands

Problem	Number of Nodes	Network Delay	Execution Time
Mahjong-Based Implementations			
$\mathfrak{A}$	11	Private Cluster	20.1 sec.
		HPCC	19.3 sec.
		PlanetLab	18.5 sec.
$\mathfrak{B}$	11	Private Cluster	41.6 min.
		HPCC	41.2 min.
		PlanetLab	43.9 min.
Simjong			
$\mathfrak{D}$	1,000,000	0ms	65 min.
		10ms	57 min.
		100ms	64 min.
		500ms	60 min.
		Gaussian	68 min.
		Distance-based	59 min.

Fig. 7. The effect of network delay on system execution time.

of lightweight tiles and whenever a packet travels between nodes, nodes handle other tiles rather than waiting idly for the network communication to arrive.

Figure 7 shows a cross-section of our empirical data on the physical networks, as well as on virtual networks using Simjong. We found that the execution times were closely clustered and saw no pattern to the small variances (e.g., while the 100ms-network run took more time than the 10ms-network run, the 500ms-network run took less time than the 100ms-network run).

The experimental results are consistent with our hypothesis and provide confirmation that network delay has little to no effect on the speed of a sTile-based system. This suggests that sTile-based systems can be successful on vastly different networks, from local, highly connected ones to globally distributed, sparse ones.

4.5 Efficiency

The final claim we address in demonstrating sTile's feasibility for industrial systems is that real-world-sized problems can be solved on real-world-sized networks in reasonable time. In particular, we posit that sTile-based systems can outperform existing privacy-preserving methods for solving NP-complete problems. There are three ways to solve a highly parallelizable problem while preserving the data privacy: (1) on a large insecure network by using sTile, (2) on a single private computer, or (3) on a 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 and then discuss the actual time necessary to solve problems.

Suppose a network with N nodes uses a sTile-based system 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 3.2.4, 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 3.2.3, 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). 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. (Note that our analysis here assumes the naïve algorithm that runs in $O(2^n)$ time, but can be extended to more efficient algorithms, such as those used in today's *SAT* solvers. We use the simple algorithms to allow us to fairly compare sTile-based and traditional systems. However, both types of systems can employ the more efficient algorithms. We have already begun work on building tile assemblies, and thus sTile-based systems, that leverage the complex, faster algorithms for NP-complete problems [16]. We discuss the reasoning and implications for our assumption further in Section 4.6.)

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

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

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 the hash set contains that literal. The overhead of using sTile 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 $8n\lg^2 n$, a sTile-based system will execute faster than a single machine. For the sizes of problems we discuss next, that network size is several thousand nodes.

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 sTile over a private network is $\frac{8n\lg^2 nM}{N}$. In other words, if the size of the public network exceeds $8n\lg^2 nM$, a sTile-based system will execute faster than the private network.

We estimated the time a sTile-based system will take to solve a given problem by two methods: (1) empirically determining the values of the constants r, i, and u, and (2) running Simjong. We measured the constants on a 2.4GHz machine running Windows XP and Sun JDK 6.0 by executing several million benchmark tests and averaging their running times. We

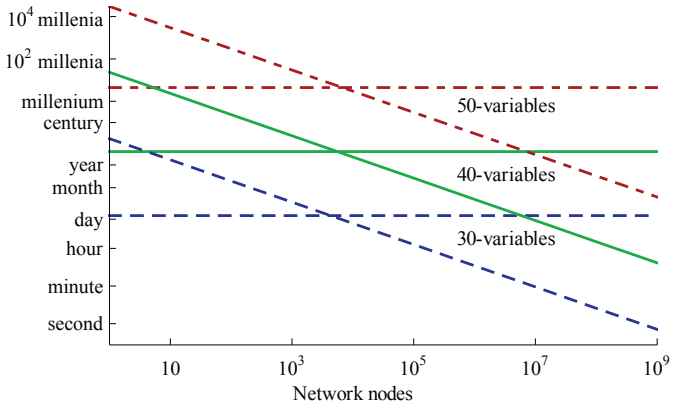

Fig. 8. Expected execution times for single-computer (horizontal lines) and sTile-based (diagonal lines) solutions for 30-, 40-, and 50-variable, 100-clause 3-*SAT* problems.

Number of Nodes	Execution Time	
	Simjong	Estimate
125,000	8.7 hours	9.1 hours
250,000	4.5 hours	4.5 hours
500,000	2.1 hours	2.3 hours
1,000,000	64 min	68 min.

Fig. 9. Comparison of execution time for solving $\mathfrak{D}$ as measured by Simjong and estimated by Equation (1).

found that $r \approx 3.6 \times 10^{-7}$ seconds (≈ 2.8 MHz), $i \approx 2.8 \times 10^{-7}$ seconds (≈ 3.8 MHz), and $u \approx 4.1 \times 10^{-7}$ seconds (≈ 2.4 MHz). With these measurements and Equations (1) and (2), we can estimate the speeds of a sTile-based 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 ≈ 1 year. However, the same problem could be solved using sTile on a million-node network in 1.8×10^5 seconds ≈ 2.1 days.

Figure 8 compares the execution times of sTile 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 sTile-based system implemented in the Mahjong implementation framework and deployed on networks of varying sizes. For networks larger than about 4000 nodes, sTile-based solutions outperform their competitors; for extremely large networks, sTile-based 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 a sTile-based solution implemented in Mahjong and deployed on the network the size of SETI@home (1.8 million nodes [49]) would take 7 days.

To confirm these results, we compared the execution times measured by Simjong with the estimates from Equation (1) for $\mathfrak{D}$. Figure 9 shows the comparison. We have consistently found that Equation (1) was within 8% of the Simjong-measured execution times.

4.6 Threats to Validity

In our evaluation, as well as in targeting our technique, we make several assumptions that may threaten the validity of our results.

In our comparisons between sTile-based and other approaches to solving NP-complete problems, we have used simple underlying algorithms for those problems (e.g., ones that take $\Theta(2^n)$ time to solve n-bit-sized problems.) Some alternative systems that exist today employ much faster algorithms; however, since the tile assembly model is Turing-universal, there exists tile assemblies that implement these efficient algorithms and sTile can leverage those assemblies to create efficient sTile-based systems. In fact, we have already implemented some such efficient tile assemblies (e.g., one that solves 3-*SAT* in $O(1.8394^n)$ time [16]). In our analysis, we have made the assumption that our comparisons would be similar to comparisons between these efficient systems. In part, this assumption is justified because using the same efficient algorithm for sTile-based and conventional systems simply reduces the amount of required computation by the same factor. There is even some reason to believe that the analysis of efficient systems would favor sTile-based systems because in our analysis of the simple algorithm, we accounted for a number of shortcuts and efficiency "hacks" that apply to conventional systems, but that are unlikely to apply to the efficient algorithms. Nonetheless, this assumption poses a potential threat to the validity of our analysis.

One of the uses of sTile we have suggested involves distributing the software system onto multiple clouds, ensuring that no entity controls too large a fraction of the underlying network. While certainly feasible, such a distribution presents several challenges we have not described here. Notably, today's clouds tend to lack interoperability, making it difficult to deploy a single system on multiple clouds. While we have addressed part of this issue since Mahjong-based systems only require the underlying nodes to be able to execute Java virtual machines, some engineering challenges may remain in deploying such systems on multiple clouds and allowing for proper communication between the nodes.

We have taken into account accurate models of how the underlying network handles message delivery and the involved delays. However, we have assumed that the volume of network traffic created by sTile-based systems will not affect message delivery, in particular, that the volume will not be significantly larger than typical volumes. Our deployments on physical networks suggest that this assumption holds for the networks we have explored; however, it is conceivable that for some networks, the traffic volume will significantly increase when executing sTile-based systems and message delivery may suffer.

5 Privacy Preservation

We have demonstrated our evidence that sTile-based systems are scalable and efficient enough to be able to solve real-world problems. We now argue that sTile-based systems exhibit the very property that motives them: privacy of the data. It is difficult to prove privacy preservation empirically because any

such analysis can only be as strong as the threat model and its implementation. Instead, in this section, we will analytically argue that, as long as no adversary controls more than half the network, the probability of that adversary learning the input can be made arbitrarily low. This argument provides a strong guarantee of privacy preservation.

sTile's privacy preservation comes from each tile being exposed only to a few intermediate bits of the computation (see Figure 2) and the tiles' lack of awareness of their global position. In order to learn meaningful portions of the data, an adversary needs to control multiple, adjacent tiles. We call a distributed software system *privacy preserving* if, with high probability, a randomly chosen group of nodes smaller than half of the network cannot discover the entire input to the computational problem the system is solving. (We will also discuss, at the end of this section, the probability of discovering parts of the input.) We argue that (1) a node deploying a single tile knows virtually no information about the input, (2) a node deploying multiple tiles knows virtually no information about the input, and (3) controlling enough computers to learn the entire input is prohibitively hard on large public networks.

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 a node were to deploy only a single tile, it would only be able to learn information such as "there is at least one 0 bit in the input," which is less than one bit of information.

2. Each node on the network may deploy several tiles (all of the same type). However, each tile is only aware of neighboring tiles and not of it's global position. Thus, if a node deploys several non-neighboring tiles, that node cannot reconstruct any more information than if it only deployed a single tile. The only way the node may gain more information is if it deploys neighboring tiles. (We handle this case next.)

3. Suppose an adversary controls a subset of the network nodes and can see all the information available to each of the tiles deployed on those nodes. Then the adversary can attempt to reconstruct the computation input from parts of the crystal that consist of tiles deployed on compromised nodes. Lemma 2 bounds the probability that an adversary can use this scheme to learn the 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 sTile-based 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 sTile-based 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 probability 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 close to controlling a significant fraction, $\left(\text{say, more than } \frac{1}{1000}\right)$, of the Internet [24]. As the Internet grows, for any fixed-size botnet, the probability that botnet can affect a sTile-based 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 (e.g., two- or three-bit) fragments of the input, 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 [21].

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.

6 Related Work

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

6.1 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 [28]. Among systems that concentrate

on distributed computation are BOINC systems [3] (such as SETI@home [36] and Folding@home [39]), MapReduce [25], and the organic grid [22]. A unique approach — FoldIt — uses the competitive human nature to solve the protein-folding problem [4]. FoldIt asks humans, as part of a game, to try to arrange a protein's amino acids to minimize the free energy. The hope is that humans will be more efficient than the brute force approaches and that a large number of users will help find the optimal solution. These systems try to solve exactly the highly parallelizable problems toward which our work is geared, but unlike sTile, they do not preserve privacy.

Cloud computing is a relatively new phenomenon that allows outsourcing computation. Corporations such as Google, Yahoo!, and Amazon have the computational resources to distribute these computations onto thousands of privately-owned, networked, fairly reliable machines. Clouds leverage technologies such as MapReduce [25] to handle the data and computation distribution. Today, a MapReduce system running on a 10,000-core cluster produces data used in every Yahoo! web search query [5]; as many as a thousand MapReduce jobs are executed on Google's and Amazon's clusters daily [2], [25]; and Facebook uses a MapReduce system to process more than 15 terabytes of new data every day [53]. While commercially viable, clouds rely on legal contracts to ensure privacy. For example, if a pharmaceutical company outsources a protein-folding problem to Google, that company must share the valuable amino acid sequence with Google and is protected from Google misusing the sequence or making that data public by a contract. Our approach allows the pharmaceutical company to distribute its problem, in principle, on several clouds without having to disclose the private data, while providing guarantees that the operators of these clouds, as well as potential attackers, cannot compromise that data.

Some research, rather than leveraging large networks, has attempted to accelerate NP-complete computation by developing faster algorithms for single machines and small clusters. This work ranges from developing efficient exponential-time algorithms [37], [51], to using runtime information to dynamically improve the speed of *SAT* solvers [6], to leveraging local message-passing protocols such as MPI and OpenMP to use small clusters to linearly accelerate the computation of specialized problems [42]. This work is not in competition with our technique, but is rather complementary. The tile architecture is based on a Turing-universal computational model [9], [44] and can implement each of these advanced algorithms on large distributed networks, leveraging both their efficiency and the tile architecture's privacy preservation, scalability, and fault tolerance. In fact, we have already built tile assemblies that implement fast 3-*SAT* algorithms that can be leveraged directly by sTile [16]. At times, in this paper, we compared simple algorithms that solve NP-complete problems implemented using the tile architecture versus using conventional methods. The same comparisons can be made for complex, efficient, state-of-the-art algorithms.

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 (as opposed to quantum) 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 [23]. Our approach avoids this shortcoming by distributing such a request over many machines without disclosing the entire problem to any small-enough subset of them.

6.2 Secure Computation

Gentry has theorized about using a fully homomorphic encryption scheme to encrypt a circuit describing a problem and then executing the encrypted circuit on a separate agent without disclosing the private data [29]. While theoretically exciting, practically, this approach cannot be used today because of the exponential amount of computation required to encrypt and decrypt. In a popular article, Gentry himself estimates that using his technique to perform a Google search, while keeping the query private, would require one trillion times as much computation as is needed today [31]. Gentry's technique is theoretically more powerful than ours because it keeps the data private from the entire network (as opposed to subsets of the network). However, as we demonstrated in Section 4, unlike homomorphic encryption, sTile is efficient enough to be used today.

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. sTile 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 [52]. Zero-knowledge compilers bridge that work closer to our approach by making Yao's protocol secure even if the parties cannot be trusted [30]. 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 described in Section 3.2.3.

7 Contributions

We have developed a new technique, called sTile, for designing, developing, and implementing software systems that distribute computation onto large, insecure, public networks. sTile provides the opportunity to design software systems aimed at large distributed networks without having to worry about distribution, privacy preservation, fault and adversary tolerance, and scalability, as those properties are inherent to sTile. We presented a rigorous theoretical analysis of sTile 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 create Mahjong, an implementation framework, and two reference

Mahjong-based implementations. We deployed these implementations on several networks, including the globally distributed PlanetLab [43] and simulated Mahjong-based executions on networks of up to 1,000,000 nodes to empirically verify (1) the correctness of sTile algorithms, (2) that the speed of sTile computation is proportional to the number of nodes, (3) that network delay has little to no effect on the speed of the computation, and (4) that our mathematical analysis of the time needed to solve large problems on large networks is accurate. For networks larger than about 4000 nodes, sTile outperforms optimized solutions that assume privately-owned, secure hardware?

With sTile, we have explored the fundamental cost of achieving privacy through data distribution and bound how much less efficient a privacy-preserving system is than a non-private one. While that cost is significant, we found that sTile-based systems execute orders of magnitude faster than homomorphic encryption systems, the alternative promising approach to preserving privacy. Further, we believe that our prototype system can be made more efficient and the bound can be tightened.

REFERENCES

[1] L. Adleman *et al.*, "On the decidability of self-assembly of infinite ribbons," in *FOCS*, 2002, pp. 530–537.

[2] "Amazon elastic MapReduce," http://aws.amazon.com/elasticmapreduce, 2009.

[3] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *the 5th IEEE/ACM Intl. Workshop on Grid Computing*, 2004, pp. 4–10.

[4] D. Baker, "Foldit," http://fold.it, 2009.

[5] E. Baldeschwieler, "Yahoo! launches world's largest hadoop production application," http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html, 2008.

[6] A. Balint *et al.*, "A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem," in *SAT*, 2009, pp. 284–297.

[7] T. Benzel *et al.*, "Design, deployment, and use of the DETER testbed," in *the DETER Community Workshop on Cyber Security Experimentation and Test*, 2007, pp. 1–8.

[8] B. Berger and T. Leighton, "Protein folding in the hydrophobic-hydrophilic (HP) is NP-complete," in *RECOMB*, 1998, pp. 30–39.

[9] R. Berger, *The undecidability of the domino problem*, ser. Memoirs Series. American Mathematical Society, 1966, no. 66.

[10] F. Berman *et al.*, "Adaptive computing on the grid using AppLeS," *TPDS*, vol. 14, no. 4, pp. 369–382, 2003.

[11] Y. Brun, "Mahjong tile style implementation," http://www.cs.washington.edu/homes/brun/Mahjong.

[12] ——, "Arithmetic computation in the tile assembly model: Addition and multiplication," *Theoretical Computer Science*, vol. 378, no. 1, pp. 17–31, 2007.

[13] ——, "Nondeterministic polynomial time factoring in the tile assembly model," *Theoretical Computer Science*, vol. 395, no. 1, pp. 3–23, 2008.

[14] ——, "Solving NP-complete problems in the tile assembly model," *Theoretical Computer Science*, vol. 395, no. 1, pp. 31–46, 2008.

[15] ——, "Solving satisfiability in the tile assembly model with a constant-size tileset," *Journal of Algorithms*, vol. 63, no. 4, pp. 151–166, 2008.

[16] ——, "Improving efficiency of 3-SAT-solving tile systems," in *DNA*, 2010, pp. 70–81.

[17] Y. Brun, G. Edwards, J. young Bang, and N. Medvidovic, "Smart redundancy for distributed computation," in *ICDCS*, 2011.

[18] Y. Brun and N. Medvidovic, "Fault and adversary tolerance as an emergent property of distributed systems' software architectures," in *the 2nd Intl. Workshop on Engineering Fault Tolerant Systems*, 2007, pp. 38–43.

[19] R. Buyya *et al.*, "Neuroscience instrumentation and distributed analysis of brain activity data: a case for eScience on global grids," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 15, pp. 1783–1798, 2005.

[20] M. Campbell *et al.*, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 57–83, 2002.

[21] M. Chaisson *et al.*, "Fragment assembly with short reads," *Bioinformatics*, vol. 20, no. 13, pp. 2067–2074, 2004.

[22] A. J. Chakravarti and G. Baumgartner, "The organic grid: Self-organizing computation on a peer-to-peer network," in *ICAC*, 2004, pp. 96–103.

[23] A. M. Childs, "Secure assisted quantum computation," *Quantum Information and Computation*, vol. 5, no. 456, 2005.

[24] D. Dagon *et al.*, "A taxonomy of botnet structures," in *the 23rd Computer Security Applications Conference*, 2007, pp. 325–339.

[25] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[26] J. Duerig *et al.*, "Flexlab: A realistic, controlled, and friendly environment for evaluating networked systems," in *the 5th Workshop on Hot Topics in Networks*, 2006, pp. 103–108.

[27] S. Floyd and V. Paxson, "Difficulties in simulating the Internet," *Transactions on Networking*, vol. 9, no. 4, pp. 392–403, 2001.

[28] I. Foster *et al.*, "The anatomy of the grid: Enabling scalable virtual organizations," *Intl. Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[29] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009, pp. 169–178.

[30] O. Goldreich *et al.*, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems," *Journal of the ACM*, vol. 38, no. 3, pp. 690–728, 1991.

[31] A. Greenberg, "IBM's blindfolded calculator," *Forbes Magazine*, 2009.

[32] A. S. Grimshaw *et al.*, "The Legion vision of a worldwide virtual computer," *Comm. of the ACM*, vol. 40, no. 1, pp. 39–45, 1997.

[33] "High performance computing and communications," http://www.usc.edu/hpcc.

[34] Javelin Strategy & Research, "2010 identity fraud survey report," http://www.marketresearch.com/product/display.asp?productid=2592343, 2010.

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

[36] E. Korpela *et al.*, "SETI@home — massively distributed computing for SETI," *IEEE MultiMedia*, vol. 3, no. 1, pp. 78–83, 1996.

[37] O. Kullmann, "New methods for 3-SAT decisions and worst-case analysis," *Theoretical Computer Science*, vol. 223, pp. 1–72, 1999.

[38] M. Lamanna, "The LHC computing grid project at CERN," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 534, no. 1–2, pp. 1–6, 2004.

[39] S. M. Larson *et al.*, *Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology*, 2002.

[40] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A style-aware architectural middleware for resource-constrained, distributed systems," *TSE*, vol. 31, no. 3, pp. 256–272, 2005.

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

[42] A. Nakano *et al.*, "Scalable atomistic simulation algorithms for materials research," *Scientific Programming*, vol. 10, no. 4, pp. 263–270, 2002.

[43] L. Peterson *et al.*, "A blueprint for introducing disruptive technology into the Internet," *Computer Communication Review*, vol. 33, no. 1, pp. 59–64, 2003.

[44] R. M. Robinson, "Undecidability and nonperiodicity for tilings of the plane," *Inventiones Mathematicae*, vol. 12, no. 3, pp. 177–209, 1971.

[45] S. M. Rubin, *Computer Aids for VLSI Design*. Addison-Wesley, 1994.

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

[47] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.

[48] H. Wang, "Proving theorems by pattern recognition," *II. Bell System Technical Journal*, vol. 40, pp. 1–42, 1961.

[49] Wikipedia, "SETI@home," http://en.wikipedia.org/wiki/SETI@home, 2008.

[50] E. Winfree, "Simulations of computing by self-assembly of DNA," California Institute of Technology, Tech. Rep. CS-TR:1998:22, 1998.

[51] G. J. Woeginger, "Exact algorithms for NP-hard problems: a survey," *Combinatorial Optimization - Eureka, You Shrink!*, vol. 2570/2003, pp. 185–207, 2003.

[52] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS*, 1986, pp. 162–167.

[53] M. Zaharia *et al.*, "Job scheduling for multi-user MapReduce clusters," UC Berkeley EECS Department, Tech. Rep. UCB/EECS-2009-55, 2009.

APPENDIX
TILE ASSEMBLY MODEL

In this Appendix, we formally describe the tile assembly model and the tile assembly we developed to solve 3-*SAT* [15].

THEORETICAL UNDERPINNINGS

The tile assembly model has *tiles*, or squares, that stick or do not stick together based on various *interfaces* on their four sides. Each tile has an interface on its top, right, bottom, and left side, and each distinct interface has an integer *strength* associated with it. The four interfaces, 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 interfaces 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 $f : \mathbb{N}^n \to \mathbb{N}^m$ 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 and the system attaches 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 that we will denote as the ✓ tile.

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.

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

Developing a tile assembly is a process similar to programming or specifying an algorithm. On the surface, tile assemblies are low-level programs, such as instances of Turing machines or cellular automata. However, it is possible to use high-level paradigms, such as encapsulation, abstraction, and recursion to engineer tile assemblies. For example, we have previously designed a multiplication tile assembly [12] that we later use as a subroutine in other assemblies [13].

Figure 10 shows the 64 possible types of tiles of the 3-*SAT*-solving assembly. The tiles "communicate" via their side interfaces. Some interfaces contain a 0 or a 1, communicating

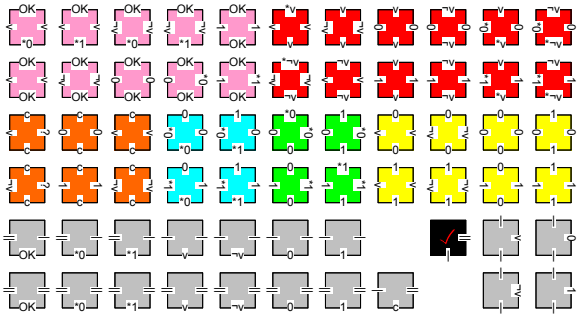

Fig. 10. A tile assembly that solves 3-*SAT* consists of 64 tile types.

a single bit to their neighbors. Other interfaces include special symbols such as v and ¬v indicating that a variable is being addressed, * 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.

Every 3-*SAT* input Boolean formula can be encoded as a sequence of tiles. Such a formula consists of a conjunction of clauses, each of which, in turn, consists of a disjunction of literals. Each literal, either a Boolean variable or its negation, can be encoded with a binary representation of the variable's index and a single bit indicating negation. For example, the literal x_2 can be encoded as three tiles, with labels 1, 0, and v, and the literal $\neg x_3$ can be encoded as three tiles with labels 1, 1, and ¬v. We insert a special c tile between the clauses.

Figure 11 shows the progress of the growth of a sample crystal of the tile assembly that solves 3-*SAT*. The example asks the question whether $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ is satisfiable.

Figure 11a shows the starting seed crystal of the computation. Note that this seed encodes ϕ. For example, the three leftmost tiles of the bottom row encode the literals $10\text{v} = x_2$, $01\neg\text{v} = \neg x_1$, and $00\neg\text{v} = \neg x_0$, which represent the first clause of ϕ. The rightmost column encodes the fact that ϕ contains three variables $10? = x_2$, $01? = x_1$, and $00? = x_0$.

Figure 11b shows the first three tiles (instances of tile types from Figure 10) that attach to the seed. These tiles will make the nondeterministic decision on whether to try $x_0 = \mathit{TRUE}$ or $x_0 = \mathit{FALSE}$. Note that these tiles' left interfaces encode 00v, indicating that the assembly has nondeterministically chosen $x_0 = \mathit{TRUE}$ (00¬v would have indicated $x_0 = \mathit{FALSE}$).

Having selected the assignment for x_0, the assembly compares the rightmost literal in ϕ to that assignment. Figure 11c shows that comparison. The top left corner tile with a top interface containing a * indicates that the literal and the assignment match (they are both $00\text{v} = x_0$). If the assignment and literal did not match, the top left corner's top interface would contain no *. Figure 11d shows the comparison of the x_0 assignment to the rest of ϕ. Since the unnegated literal x_0 does not appear anywhere else in ϕ, the rest of the top-row tiles do not contain a * in their top interfaces.

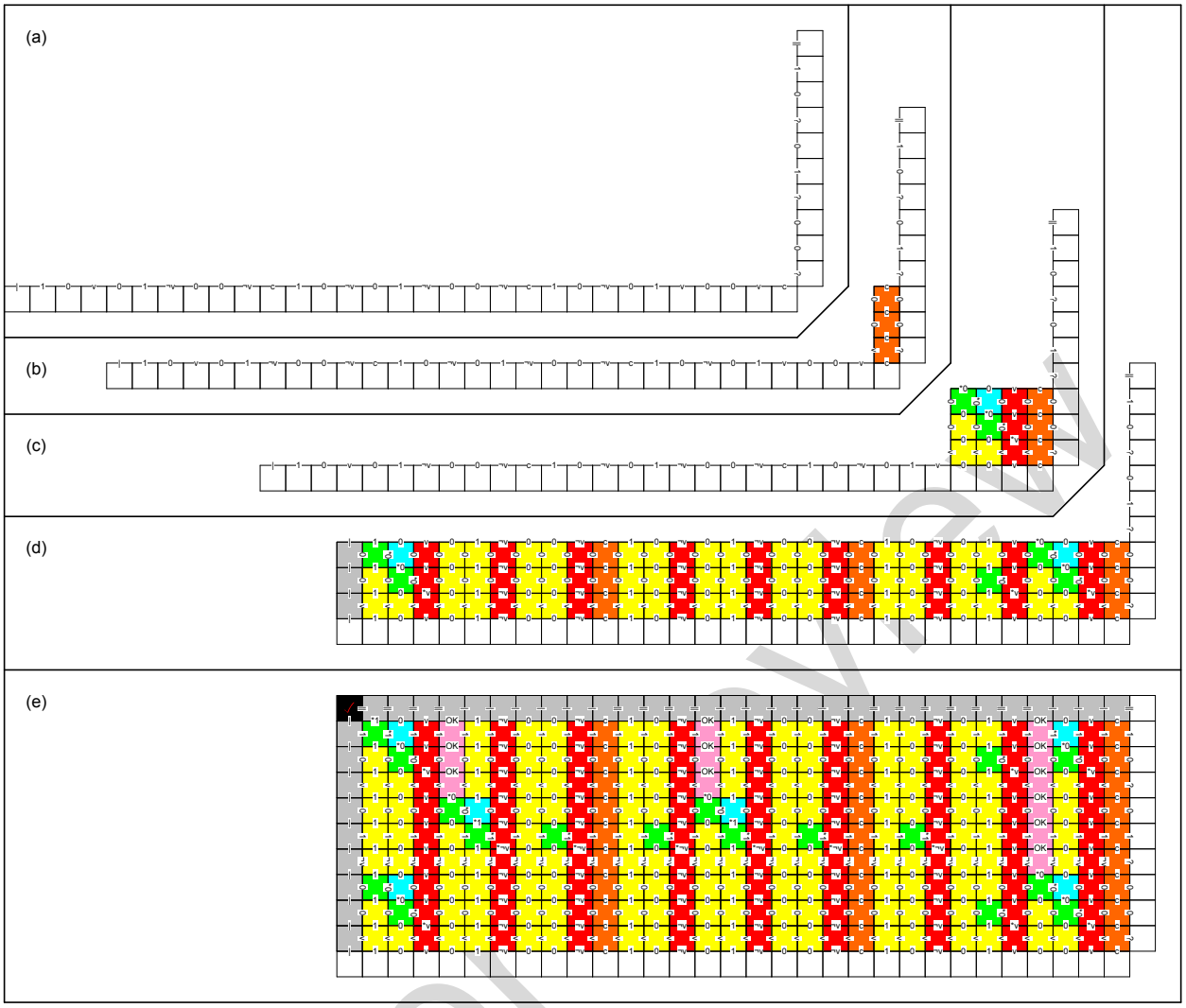

Fig. 11. An example progression of the growth of a crystal of the 3-*SAT*-solving tile assembly. The crystal seed (a) consists of the clear seed tiles, encoding the input $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$. Specially designed tiles attach to nondeterministically select a variable assignment for x_0 (b); compare that assignment to, first, a single literal in ϕ (c); and then, the rest of the literals in ϕ (d); and, finally, repeat those steps for variables x_1 and x_2 and ensure that each clause is satisfied at least once by the particular selected variable assignment (e). 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 in (e), the ✓ tile attaches in the top left corner.

In Figure 11e, the assembly repeats the above steps to nondeterministically select assignments for x_1 and x_2 and compares each of those to the literals in ϕ. Whenever a match occurs, tiles with *OK* top interfaces propagate that information up, to the top row. Finally, a series of gray tiles attach in the top row to check whether each clause has at least one literal match the assignment. If it does, the special ✓ tile can attach in the top left corner of the crystal. If some clause were not satisfied, no such tile could attach. The fact that Figure 11e contains the ✓ tile indicates that the nondeterministically chosen truth assignment $\langle x_0, x_1, x_2 \rangle = \langle TRUE, FALSE, TRUE \rangle$ satisfies ϕ.

We refer the reader to [15] for the formal proof that this assembly solves 3-*SAT*. As we explain in Section 3, a single tile assembly, such as the 3-*SAT*-solving one we have described here, is sufficient to develop sTile-based systems. However, as part of our work on sTile, we have also provided a tile assembly solution for *SubsetSum*, another well-known NP-complete problem [14]. This second assembly illustrates the flexibility of our work and provides some insight into possible sTile efficiency improvements.

The tile assembly we have described here follows the algorithm that runs in $O(2^n)$ time. It is possible to leverage more efficient algorithms that solve NP-complete problems to develop efficient tile assemblies. We have already designed one such assembly that solves 3-*SAT* in $O(1.8394^n)$ time, but do not describe it here because of its complexity [16]. It is important, however, to make clear that tile assemblies can implement the same algorithms used on today's fastest systems that solve NP-complete problems, such as *SAT* solvers.

Yuriy Brun is an NSF CRA postdoctoral Computing Innovation Fellow at the University of Washington. He received his Ph.D. degree in 2008 from the University of Southern California, as a USC Viterbi School of Engineering Fellow, and his M.Eng. degree in 2003 from the Massachusetts Institute of Technology. His doctoral research was a finalist in the ACM Doctoral Dissertation Competition in 2008. Brun's research interests are in the area of engineering self-adaptive systems, and, in particular, using mechanisms from nature to create engineering paradigms for robustness, fault and malice tolerance, scalability, and security. He does (1) theoretical work on design and complexity analysis of biologically inspired algorithms and (2) software engineering work on implementing these algorithms for Internet-sized distributed systems, grids, and clouds. He is a member of the ACM and the ACM SIGSOFT.

Nenad Medvidovic received the Ph.D. degree in 1999 from the University of California, Irvine. Currently, he is a professor in the Computer Science Department at the University of Southern California. He is a recipient of the US National Science Foundation CAREER award. His research interests are in the area of architecture-based software development. His work focuses on software architecture modeling and analysis; middleware facilities for architectural implementation; product-line architectures; architectural styles; and architecture-level support for software development in distributed, mobile, resource constrained, and embedded computing environments. He is a member of the ACM, ACM SIGSOFT, and IEEE.