Entrusting Private Computation and Data to Untrusted Networks

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

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 nonprivate one. 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.

Index Terms—Privacy, trusted computing, untrusted networks, distributed computation, PlanetLab, sTile, tile assembly model

1 Introduction

The emergence of cloud computing is evolving the nature of computation. Instead of using private machines, users allow the cloud to maintain, manipulate, and safeguard their data. This evolution has allowed ubiquitous access to computation and data with higher availability and reliability than possible with personal machines and local servers. Simultaneously, this evolution has affected the meaning of the term *privacy* when referring to software systems. To ensure data remain private, not only must they be kept confidential from potential intruders, but also from the machines that execute computation on the data. In other words, when computing on potentially untrusted machines, such as cloud nodes, no entity, including those executing the computation, should gain access to the data. Cloud providers have not yet embraced these new definitions, largely due to the intellectual hurdles and costs of developing systems that conform to these high standards. Instead, they rely on legal contracts and promises. For example, many of us rely on Google to deliver, maintain, and properly replicate, our e-mail, and while Google promises not to misuse our data, the notion that it may be possible to use Google's services without Google having access to our e-mail seems unreasonable by today's practices. Meanwhile, 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 identity theft perpetrated by either pretending to be a reputable company or simply relying on a user's trust in an unknown service [29].

This paper addresses the challenge of *executing computations on untrusted machines in a trustworthy manner*. Its focus is on preserving data privacy while solving computationally intensive problems on untrusted machines.

We present sTile, a technique for building software systems that distribute large computations onto the cloud while providing guarantees that the cloud nodes cannot learn the computation's private data. sTile is based on a nature-inspired, theoretical model of self-assembly. While sTile's computational model is Turing universal [37], in this paper, we present a prototype implementation that solves NP-complete problems.

sTile explores the fundamental cost of privacy through data distribution. Existing approaches to using the Internet's computational resources either assume reliable and trustworthy underlying machines [7], [21], [27], only store data privately and rely on trusted entities to compute [3], [42], [46], [48], or are theoretical constructs (e.g., quantum computing [19] and homomorphic encryption [25]) that, to date, have not produced implementations efficient enough for practical use [26].

We evaluate sTile in three ways: First, we formally prove that sTile systems preserve data privacy as long as no adversary controls more than one-half of the cloud. Second, we empirically demonstrate sTile's feasibility by deploying an open-source, publicly available prototype implementation [14] on three distinct networks, including the globally distributed PlanetLab testbed [36]. Third, we formally analyze the communication and computation costs induced by sTile, bound them, and empirically verify those bounds. We have previously discussed sTile's ability to handle faults and malicious attacks [13], [15], and do not focus on that dimension here. This paper extends our earlier work [16] with sTile's tile architecture and tile algorithms details

- *Y. Brun is with the School of Computer Science, University of Massachusetts, 140 Governors Drive, Amherst, MA 01003-9264. E-mail: brun@cs.umass.edu.*
- *N. Medvidovic is with the Computer Science Department, University of Southern California, 941 Bloom Walk, Henry Salvatori Computer Science Center 338, Los Angeles, CA 90089-0781. E-mail: neno@usc.edu.*

Manuscript received 15 July 2012; revised 28 Oct. 2012; accepted 23 Jan. 2013; published online 13 Feb. 2013.
Recommended for acceptance by S. Distefano, A. Puliafito, and K.S. Trivedi. For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSCSI-2012-07-0170.
Digital Object Identifier no. 10.1109/TDSC.2013.13.

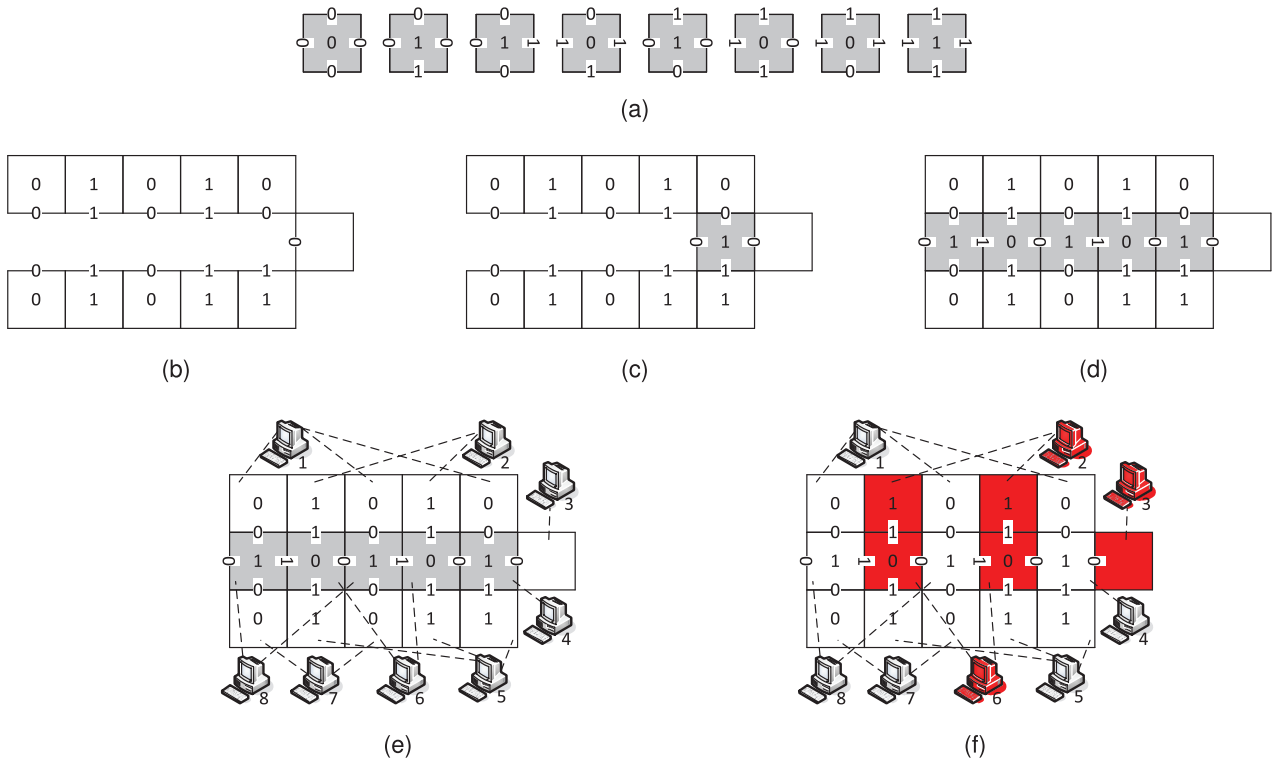

Fig. 1. An adding tile assembly with (a) eight tile types. A seed crystal (b) encodes the inputs, $10 = 1010_2$ and $11 = 1011_2$. The first attaching tile (c) adds the least significant bit of each input. The middle row of the final crystal (d) encodes the output $21 = 10101_2$. sTile deploys (e) software objects encoding individual tiles onto nodes. Even if an adversary compromises a significant fraction of the nodes (f), the probability that it can recover the private data is extremely low.

and an evaluation of sTile's scalability and efficiency in solving real-world problems on today's hardware. sTile significantly outperforms existing cryptography-based privacy techniques, such as homomorphic encryption [25].

The rest of this paper is structured as follows: Section 2 explains sTile through an example. 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 Motivating Example: Addition

sTile preserves privacy by breaking a computation into small pieces and distributing those pieces onto a large network. Each piece is so small that it is prohibitively difficult for an adversary to collect enough pieces to reconstruct the confidential data. In this section, we describe sTile with an example of distributing an addition computation.

To describe adding using sTile, we explain three separate elements of our solution: the addition tile assembly, the distribution process, and the source of privacy.

2.1 The Addition Tile Assembly

A tile assembly is a theoretical construct, similar to cellular automata. It consists of square *tiles* with static labels on their four sides. Tiles can *attach* to one another or to a growing crystal of other tiles when sufficiently many of their sides match.

Fig. 1a shows eight different *types of tiles* used for addition. These tile types are the program—the tile assembly encoding of the algorithm for adding two integers, in binary, one bit at a time. (We designed these tiles through a process similar to programming in assembly. We do not focus on this process here, but describe in Section 3.2 an automated compilation procedure that allows a developer to use our technique without having to design new tile assemblies.) Fig. 1b shows a *seed crystal* that encodes an input: 10 (1010 in binary) in the top row and 11 (1011 in binary) in the bottom row. When an instance of a tile from Fig. 1a matches the seed crystal on three sides, that tile instance attaches to the crystal. Fig. 1c shows the seed with a single attached tile. Note that this tile adds the least significant bit of each input: $0 + 1 = 1$, displayed in the center of the newly attached tile. The label on the west side of the newly attached tile is the carry bit: 0. The tiles execute full adder logic to add the bits, one at a time, eventually producing the sum $21 = 10101_2$ in the middle row of Fig. 1d.

2.2 The Distribution Process

sTile uses the theoretical tile assembly to decompose a computation into small parts. Each small part represents a tile. Fig. 1e shows how the $10 + 11$ execution might be deployed on eight network nodes. Each node only deploys tiles of a single type, designated by the client machine (described in Section 3.2.1). The client sets up a seed on the network by asking nodes that can deploy tiles of appropriate types to deploy instances of those types (described in Section 3.2.1). Each node knows only the tile instances it is

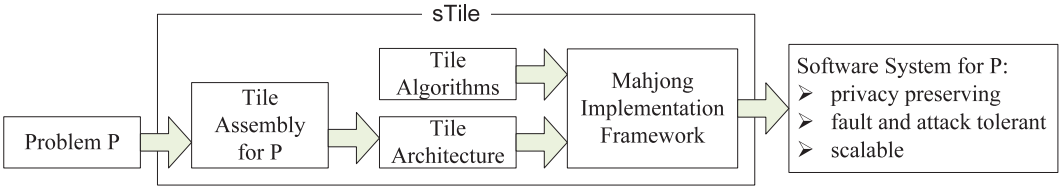

Fig. 2. A high-level overview of sTile.

deploying and maintains references to the geometrically adjacent tile instances on other nodes. Next, tiles with an empty adjacent location coordinate with their crystal neighbors to recruit matching tiles to attach (described in Section 3.2.3). This process uses a secure multiparty computation algorithm to ensure crystal neighbors do not learn each other's data [47]. (While a single seed is sufficient for addition, for NP-complete problems, sTile employs distributed seed replication, described in Section 3.2.4.) Each of these steps replies on an algorithm that ensures the tiles are deployed uniformly randomly on the available nodes (described in Section 3.2.2). Once the execution finishes, the tiles in the middle row report the solution to the client, indicating the node IDs of their crystal neighbors, which the client uses to reconstruct the output.

2.3 The Source of Privacy

Each tile instance is aware of only a single bit of the input, output, or intracomputation data, and not of the bit's global location. An adversary may attempt to reconstruct the confidential data from the nodes it controls. For example, Fig. 1f shows an adversary that has compromised three nodes (2, 3, and 6), and now has access to the data in five tiles. However, this adversary can only tell that there are some 0 and 1 bits scattered throughout the input, the computation, and the output, but not how many and not their relative positions. In fact, in this example, no three nodes contain the entire input (nodes 1, 2, 5, and 7 deploy the input). The adversary may recover partial information about the frequencies of 0s and 1s, and may be able to reconstruct small parts of the input and output (e.g., if it is lucky enough to control adjacent tiles). However, as we show in Section 5, as long as the adversary controls less than half the network, the probability that it can reconstruct the input is prohibitively small. In the $10 + 11$ example, it is easy to see that controling half the nodes translates to controlling roughly half the tiles, which is unlikely to be sufficient to reconstruct the input or the output.

3 sTile

sTile is a technique for designing, implementing, and deploying software systems that distribute computation onto large, insecure, public networks. sTile's primary concern is to perform computation while preserving the privacy of the involved data. Fig. 2 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. We have developed multiple tile assemblies, specifically for sTile. These assemblies solve NP-complete problems [10], [11], [12] and factor integers [9]. In this paper, we use one of those assemblies to demonstrate how sTile can solve 3-SAT. The nature of NP-complete problems allows for polynomial-time translations among them, so it is possible to use sTile for all NP-complete problems without designing new assemblies, although we do not discuss that approach here. Tile assemblies are Turing-universal [37], so future extensions of sTile can be made to perform arbitrary computations and to automatically compile programs into tile assemblies.

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. 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 a tile assembly. Tile assemblies are extensively studied mathematical objects [1], [6], [37], [41], [44]. Our own prior work has developed the notion of efficient computation with tile assemblies and constructed efficient assemblies to add and multiply integers [8], factor integers [9], and solve NP-complete problems [10], [11], [12]. More generally, the tile assembly model is Turing universal [6], [37].

Tile assemblies are theoretical objects that have no notion of privacy, although it is their basic structure that allows sTile to preserve privacy. sTile is a reification of a tile assembly as a distributed software system. 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. We now formally define the tile assembly model and describe the tile assembly we developed to solve 3-SAT [11], though the reader need not master this formalism to appreciate sTile.

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 set of tile types in a tile assembly encodes the "program" the

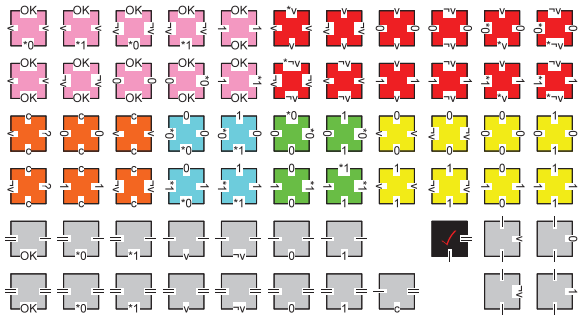

Fig. 3. A tile assembly that solves 3-SAT consists of 64 tile types.

tiles will execute. For example, the eight tile types from Fig. 1a encode integer addition. The placement of a set of tiles on a 2D 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 crystal neighbors exceeds the current *temperature*. Starting from a *seed crystal*, tiles may attach to form new crystals. Sometimes, several tiles may satisfy the conditions necessary to attach at a position, in which case the attachment is nondeterministic. A tile assembly $\mathbb{S}$ computes a function $f : \mathbb{N}^n \rightarrow \mathbb{N}^m$ if there exists a mapping i from $\mathbb{N}^n$ to crystals and a mapping o from crystals to $\mathbb{N}^m$ such that for all inputs $\vec{\alpha} \in \mathbb{N}^n$, $i(\vec{\alpha})$ is a seed crystal such that $\mathbb{S}$ attaches tiles to produce a terminal crystal F and $o(F) = f(\vec{\alpha})$. In other words, if there exists a way to encode inputs as crystals 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 $\surd$ tile.

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 [8] that we later use as a subroutine in other assemblies [9].

Fig. 3 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 a single bit to their crystal neighbors. Other interfaces include special symbols such as v and $\neg$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 $\surd$ tile attaches to the crystal.

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 $\neg$v. We insert a special c tile between the clauses.

Fig. 4 shows the progress of the growth of a sample crystal of the tile assembly that solves 3-SAT. The example asks the question "Is $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ satisfiable?"

Fig. 4a shows part of the seed crystal of the computation. This seed encodes ϕ. For example, the three tiles near the right side with labels 0, 0, and v encode the literal x_0, which is the last literal of the last clause of ϕ. The rightmost column (seen easiest in Fig. 4e) encodes the fact that ϕ contains three variables $10? = x_2$, $01? = x_1$, and $00? = x_0$.

Fig. 4b shows the first three tiles (instances of tile types from Fig. 3) that attach to the seed. These tiles make the nondeterministic decision on whether to try $x_0 = TRUE$ or $x_0 = FALSE$. Note that these tiles' left interfaces encode 00v, indicating that the assembly has nondeterministically chosen $x_0 = TRUE$ (00$\neg$v would have indicated $x_0 = FALSE$).

Having selected the assignment for x_0, the assembly compares the rightmost literal in ϕ to that assignment. Fig. 4c 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 *. Fig. 4d 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.

In Fig. 4e, the assembly repeats the above steps to nondeterministically select assignments for x_1 and x_2, and compare 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 had at least one literal match the assignment. If it does, the special $\surd$ tile can attach in the top left corner of the crystal. If some clauses were not satisfied, no such tile could attach. The fact that Fig. 4e contains the $\surd$ 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 [11] for the formal proof that this assembly solves 3-SAT. As discussed above, 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 [10]. 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 [12]. We mention it here 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.

Fig. 4. 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 (Fig. 3) 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 black $\surd$ tile attaches in the top left corner.

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 simulates a tile assembly: Each computer pretends to be a tile (or many tiles), and communicates with other computers to self-assemble a solution to a computational problem. Each computer deploys tile components, each representing a tile in a tile assembly, and facilitates the proper communication channels and algorithms to allow the tile component self-assembly. Thus, a tile architecture is based on a tile assembly; the software system employing that architecture solves the particular computational problem that the tile assembly solves.

A developer interested in using sTile to construct distributed systems may follow one of two possible processes: The first approach would require a detailed understanding of tile assemblies; the second approach, which we adopt, insulates the developer from those details. Next, we outline the first approach and then elaborate the second.

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 from Section 3.1.

Instead, we recommend an automated compilation procedure that allows the developer to create a sTile-based

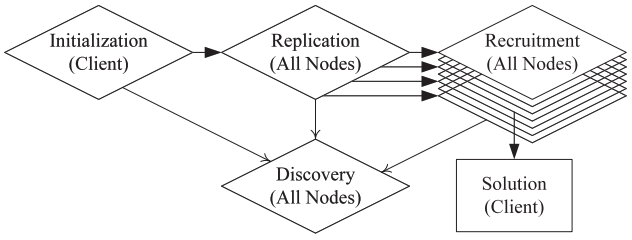

Fig. 5. Overview of tile architecture node operations.

system without ever understanding the details of tile assemblies. The compilation reduces [39] the problem for which the developer wishes to build a sTile-based system to a problem with a known tile assembly [10], [11], [12]. 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 [39] 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 [38]; and 3) in addition to ease of use, compilation masks the problem the user is solving. While this is a beneficial side effect of 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 contain a large number of components; on the other hand, as with traditional software systems, those components are of a much smaller number of different *types* (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 [40] 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, 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 one in Fig. 4a. Fig. 5 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 $\surd$ 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 underlying computation.

We elaborate on these operations next. We also discuss what happens when sTile is unable to find a solution to a computational problem.

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.

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 Section 3.1, 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 network neighbors directly and may query its network neighbors for their lists of network neighbors. 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 network neighbors, then their network neighbors, and so on, until at least one node of each type assignment in the tile type map knows the tile type map and its own tile type description. Per the *coupon collector* problem [34], a system with k tile types takes, 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 network 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 network 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 network neighbors will have to be sent the $128k$ bits, and $(\frac{128kp}{k} = \Theta(p))$. Because the diameter

of a network of N nodes with randomly distributed connections is $\Theta(\log N)$ [34], the tile type map and the tile types will propagate through the network in $\Theta(\log N)$ time.

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 of its crystal (seed) neighbors. 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. Thus, every suitable computer has an equal chance of being returned, in the long run, which in turn guarantees that all nodes on the network perform a similar amount of computation. The algorithm uses a property of random walks to ensure uniform-randomness.

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, 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 network neighbors and then gets help from those network neighbors (by requesting their network 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 the 3-SAT example with $k = 64$, 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 [34]. 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.

We now help clarify the preprocessing and discovery operations with the use of an example. Suppose the network in Fig. 6 represents the connectivity of six nodes that all map to the same tile type. In creating its node table, A first checks its network 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 network neighbor list by querying one of its network neighbors for its network 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 network 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.

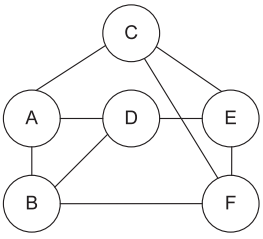

Fig. 6. A network with six nodes. We assume that every node in our underlying network has p network neighbors (here $p = 3$).

Theorem 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 [34], 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. □

Note that the random walk theorem [34] holds for all graphs with nodes with three or more graph neighbors, so this result is directly applicable to all reasonable distributed networks.

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 Section 3.1 that solves 3-SAT), a tile component that has both an upper and a left crystal neighbor recruits a new tile to attach to its upper left. Fig. 7 indicates several places in a sample crystal where tile components are ready to recruit new tiles. A recruiting tile component X (highlighted in Fig. 7), 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 crystal neighbor's left interface and X's left crystal neighbor's top interface. If those interfaces match Y's right

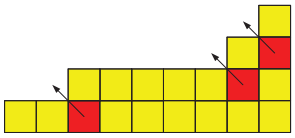

Fig. 7. Tile components that have both an upper and a left crystal neighbor (highlighted in the diagram) can recruit new components to attach to their upper left.

and bottom interfaces, respectively, then Y can attach. At that point, X informs Y of the IPs of its two new crystal neighbors, and those crystal neighbors of Y's IP. Note that X can perform this operation without ever learning its crystal neighbors' interfaces by using Yao's garbled protocol [47].

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

In the 3-SAT example from Section 3.1, the crystal recruits 310 tile components (nonclear tiles in Fig. 4e).

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 crystal neighbors. X lets its crystal neighbors know that Y is X's replica (by sending Y's IP to X's crystal neighbors). Those crystal 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 crystal neighbors about Y. We analyze these steps in Section 4.5.

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 network neighbors, and they forward that packet to their crystal neighbors, and so on. Since the diameter of a large connected network of N nodes with randomly distributed connections is $\Theta(\log N)$ [34], the "STOP" message will propagate in $\Theta(\log N)$ time.

3.2.5 Solution Reporting

One tile type, the black $\surd$ tile in Figs. 3 and 4e, includes in its encoding the identity (IP address) of the client. Recall that the black $\surd$ tile only attaches to a crystal when that crystal finds a solution. When that happens, the node deploying the black $\surd$ tile informs the client that the Boolean formula is satisfiable. While 3-SAT is a decision problem (i.e., the answer is either "yes" or "no"), the client may wish to also learn the Boolean assignment that satisfies the formula. To do so, the client may ask the node that notified it of the solution for its crystal neighbors' identities (IP addresses), and those for their crystal neighbors' identities, to reconstruct the entire crystal responsible for finding the solution. The client can then query the nodes that deploy the tiles encoding the assignment for their tile types. The cost of reconstructing the entire crystal is no more than contacting, and getting a response from each of the nodes deploying tiles in that crystal (310 tiles for the 3-SAT example from Fig. 4). However, since only part of the crystal is responsible for the assignment, it can be retrieved even more efficiently. To ensure privacy, all the relevant communication must be encrypted, and each involved node must verify the identity of the client. These requirements can be handled using standard public key encryption and authentication techniques, which we do not describe here.

While a satisfying assignment has a black $\surd$ tile-deploying node to report the solution, 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 $(1 - e^{-1})$. 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 Fig. 4, 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 on a network.

The open-source Mahjong framework [14] 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.

Mahjong's implementation uses Prism-MW [33], a middleware platform that provides explicit implementation-level constructs for declaring components, interfaces, interactions, network communication, and so on.

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 and automates the remaining steps of building a sTile-based system. The Mahjong extends PrismMW with 29 objects and 2,900 lines of Java code.

4 Computational Feasibility

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 that distribute distinct NP-complete problems on physical networks. Further, we have built Simjong, a discrete-event simulator that distributes Mahjong computations onto a simulated network of virtual nodes while controlling the network message delays. Simjong's goal is to accurately simulate Mahjong distributions on networks up to 1 million nodes.

Section 4.1 presents the Mahjong-based implementations and Simjong. Section 4.2 details our experimental setup. Sections 4.3, 4.4, and 4.5 discuss our experiments testing the scalability, robustness to network delay, and efficiency, respectively. Finally, Section 4.6. summarizes the potential threats to the validity of our evaluation.

4.1 sTile-Based Implementations

We have built and made public [14] two Mahjong-based implementations, for 3-SAT and SubsetSum, as well as Simjong-based simulations of the same systems.

Simjong [14] 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, chosen at random from some distribution, or proportional to the geographic distance between locations assigned to each virtual node. Simjong's network model is based on ns-2 [23], simplified to abstract away the exact topology of the 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.

4.2 Experimental Setup

We use three distributed networks for our experimental evaluation: 1) a private heterogeneous cluster of 11 Pentium 4 1.5-GHz nodes with 512 MB of RAM, running Windows XP or 2000; 2) a 186-node subset of USC's Pentium 4 Xeon 3-GHz High Performance Computing and Communications cluster [28]; and 3) a 100-node subset of PlanetLab [36], 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:

- $\mathcal{A}$: 5-number 21-bit SubsetSum problem,
- $\mathcal{B}$: 20-variable 20-clause 3-SAT problem,
- $\mathcal{C}$: 11-number 28-bit SubsetSum problem, and
- $\mathcal{D}$: 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 $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. 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 continued 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. In the Simjong experiments reported here, we simulated the first 10^{-4} percent of the seeds to estimate the time required to complete the entire computation. PlanetLab in particular presented a unique opportunity to test sTile as well as a number of challenges. PlanetLab is distributed on 1,090 nodes at 485 locations around the world. Almost half of PlanetLab's nodes are typically unresponsive; of the responsive nodes, some are heavily loaded or exceedingly slow. Because of these well-known issues with PlanetLab [22], we were unable to repeat our experiments as many times as on the other networks. In the end, PlanetLab did demonstrate useful numerical trends.

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.	
$\mathfrak{B}$	186	116 min.	1.90
PlanetLab	50	9.2 min.	
$\mathfrak{C}$	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. 8. The effect of doubling the network size on the system's execution time. The speedup ratio is the factor of speed improvement over the network of half the size.

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 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 100 ms for all packets). 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} percent fraction is accurate.

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. Fig. 8 shows a cross section of the results of our scalability experiments, which confirm this hypothesis. For example, executing $\mathcal{D}$ 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 Latency

Intuitively, high-network latency should adversely affect the speed of sTile-based systems: If the tile attachments happen sequentially, the latency affects every attachment and greatly slows down the overall computation. This intuition holds for the addition example from Section 2. However, in the case of NP-complete computations, this intuition is false. In such computations, many of the subcomputations (tile attachments) happen independently, in parallel. Each node in our experiments deployed millions of lightweight tiles, and whenever a sTile packet traveled between nodes, those nodes handled other tiles rather than waiting idly for the network communication to arrive. As a result, the throughput of sTile-based systems is not affected by the network latency.

We have previously formally verified that network latency has little to no effect on sTile-based system speed. We have further demonstrated this property empirically, by solving the same problem on physical networks of varying latency, and by controlling the latency in Simjong [16].

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 required 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 (Fig. 4a) and no more than $3nm\lg^2 n$ recruited tiles (nonclear tiles in Fig. 4e). 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 (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 [12]. 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

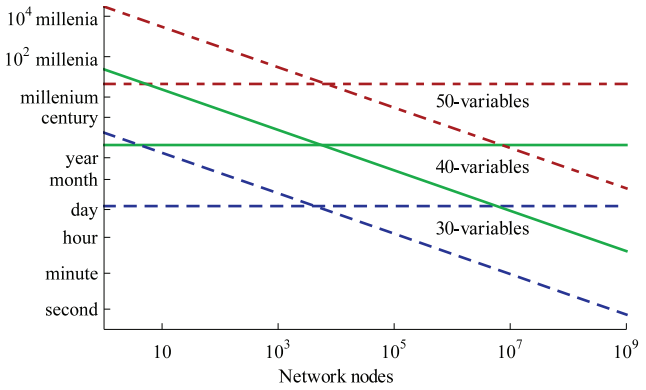

Fig. 9. Expected execution times for single-computer (horizontal lines) and sTile-based (diagonal lines) solutions for 30-, 40- and 50-variable, 100-clause 3-SAT problems.

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.

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 2.4-GHz machine running Windows XP and Sun JDK 6.0 by executing several million benchmark tests and averaging their running times. We found that $r \approx 3.6 \times 10^{-7}$ seconds (≈ 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 (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.

Fig. 9 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 4,000 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 four 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 [43]) would take seven days.

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. 10. Comparison of execution time for solving $\mathcal{D}$ as measured by Simjong and estimated by (1).

To confirm these results, we compared the execution times measured by Simjong with the estimates from (1) for $\mathcal{D}$. Fig. 10 shows the comparison. We have consistently found that (1) was within 8 percent 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 [12]). 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. 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. While we have addressed part of this issue since Mahjong-based systems only require the underlying nodes to be able to execute JVMs, some engineering challenges may remain in deploying such systems on multiple clouds.

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 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

In this section, we formally argue that sTile systems preserve privacy. Specifically, we 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.

sTile's privacy preservation comes from each tile being exposed only to a few intermediate bits of the computation (see Fig. 4) and the tiles' lack of awareness of their global position. 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 neither 1) a node deploying a single tile, nor 2) a node deploying multiple tiles can know virtually any information about the input; moreover, 3) controlling enough computers to learn the entire input is prohibitively hard on large networks.

1. Each tile type in an assembly encodes at most one bit of the input. A special tile encodes the solution, but has no knowledge of the input. A node that deploys a single tile is only 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 crystal-neighboring tiles and not of its global position. Thus, a node deploying several noncrystal-neighboring tiles 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 crystal-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's input from parts of the crystal that consist of tiles deployed on the compromised nodes. Theorem 2 bounds the probability that an adversary can use this scheme to learn the input.

Theorem 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 system is $1-(1-c^n)^s$.*

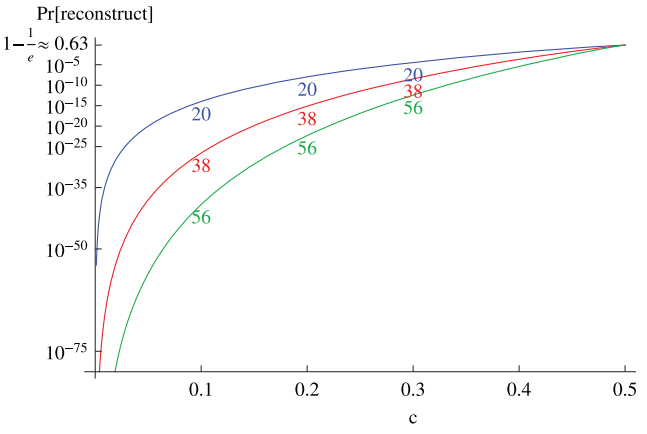

Fig. 11. The probability that an adversary controlling c fraction of the network can reconstruct an entire 20-, 38-, and 56-bit input.

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 $1-(1-c^n)^s$. □

The log-scale plot in Fig. 11 graphically illustrates that the probability of input reconstruction drops exponentially in c. Suppose we deploy a sTile system on a network of $2^{20}\approx 1{,}000{,}000$ machines to solve a 38-variable 100-clause 3-SAT problem. Also, suppose a powerful adversary has gained control of $\approx 125{,}000$ machines ($\frac{1}{8}$ of the network). The adversary will be able to reconstruct the seed with probability $1-(1-2^{-114})^{2^{38}}<10^{-22}$. As the input size increases, this probability further decreases. The probability decays exponentially for all $c<\frac{1}{2}$. An adversary who controls exactly half the network has a $1-\frac{1}{e}\approx 63\%$ chance of learning the input, which is why our technique is geared toward large public networks and networks comprised of multiple clouds.

The same analysis and exponential probability drop-off apply to reconstructing fractional parts (e.g., one-third) of the input. It is somewhat simpler to reconstruct small (e.g., 3-bit) fragments of the input, but the information contained in those fragments is greatly limited and cannot be used to reconstruct larger fragments [17].

One possible challenge to privacy preservation on large networks is botnets. However, no single botnet comes close to controlling a significant fraction, $(\text{say, more than}\frac{1}{1{,}000})$, of the Internet [20]. As the size of the underlying network grows, for any fixed-size botnet, the probability that botnet can affect a sTile system drops exponentially. Further, combining multiple clouds in a single sTile computation ensures that no single cloud provider has control of a large

fraction of the network, and thus data can be kept private from the cloud providers themselves.

Finally, each tile component handles at most a single bit of the input. Theoretically, this is sufficient for solving NP-complete problems; however, in practice, handling more than a single bit of data at a time would amortize some of the overhead. Such a transformation would result in a tradeoff between privacy preservation and efficiency, as faster computation would reveal larger segments of the input to each node.

6 Related Work

Distributing computation. The growth of the Internet has made it possible to use public computers to distribute computation to willing hosts. This notion focuses the underpinning of computational grids [24]. Among systems that concentrate on distributed computation are BOINC systems [2] (such as SETI@home [30] and Folding@home [32]), MapReduce [21], and the organic grid [18]. A unique approach—FoldIt—uses the competitive human nature to solve the protein-folding problem [4]. These systems try to solve exactly the highly parallelizable problems toward which our work is geared, but unlike sTile, they do not preserve privacy.

NP-complete computation can be accelerated by developing faster algorithms for single machines and small clusters [5], [31], [35], [45]. Such work is complementary to ours since sTile is based on a Turing-universal computational model [6], [37] and can implement each of these advanced algorithms on large distributed networks.

Cloud privacy. Cloud computing has reemphasized the importance of data privacy, causing the emergence of numerous approaches for keeping data private on the cloud [3], [42], [46], [48]. Most such approaches concentrate on private data *storage* and user-authorized data *retrieval* and require some trusted agents [3], [42], [46] whereas our work concentrates on preserving privacy during *computation* and requires no trusted agents.

Privacy-preserving computation. 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 [19]. 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.

A fully homomorphic encryption scheme encrypts a circuit (program) and then executes that circuit on a separate agent without disclosing the private data [25]. While theoretically exciting, in practice, this approach cannot be used today because of the exponential amount of computation and memory required to encrypt and decrypt. Using homomorphic encryption to perform a Google search, while keeping the query private, would require one trillion times as much computation as is needed today [26]. Homomorphic encryption is theoretically more powerful than sTile because it keeps the data private from the entire network (as opposed to subsets of the network). However, sTile is efficient enough to be used today.

Secure multiparty computation allows multiple computers, each with a part of an input, to compute a function of that entire input without sharing the parts [47]. Secure multiparty 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.

7 Contributions

sTile distributes computation onto large, insecure, public networks in a manner that ensures privacy preservation, fault and adversary tolerance, and scalability. We presented a rigorous theoretical analysis of sTile and formally proved that the resulting systems are efficient and scalable, and that they preserve privacy as long as no adversary controls half of the public network.

We deployed two sTile implementations on several networks, including the globally distributed PlanetLab [36], 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 a negligible 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 4,000 nodes, sTile outperforms optimized solutions that assume privately owned, secure hardware.

sTile explores the fundamental cost of achieving privacy through data distribution and bounds the extent to which a privacy-preserving system is less efficient than a nonprivate one. While that cost is not trivial, we have demonstrated that sTile-based systems execute orders of magnitude faster than homomorphic encryption systems, the alternative promising approach to preserving privacy.

Acknowledgments

The authors thank Jae young Bang for his help deploying Mahjong-based implementations on PlanetLab. This material was based upon work supported by the US National Science Foundation (grant 0937060 to the Computing Research Association for the CIFellows Project and grants 0905665 and 1117593), and by Infosys.

References

[1] L. Adleman, J. Kari, L. Kari, and D. Reishus, "On the Decidability of Self-Assembly of Infinite Ribbons," *Proc. 43rd Ann. IEEE Symp. Foundations of Computer Science (FOCS '02)*, pp. 530-537, Nov. 2002.

[2] D.P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," *Proc. Fifth IEEE/ACM Int'l Workshop Grid Computing (GRID '04)*, pp. 4-10, 2004.

[3] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage," *ACM Trans. Information and System Security*, vol. 9, no. 1, pp. 1-30, Feb. 2006.

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

[5] A. Balint, M. Henn, and O. Gableske, "A Novel Approach to Combine a SLS- and a DPLL-Solver for the Satisfiability Problem," *Proc. 12th Int'l Conf. Theory and Applications of Satisfiability Testing (SAT '09)*, pp. 284-297, 2009.

[6] R. Berger, *The Undecidability of the Domino Problem,* no. 66. Am. Math. Soc., 1966.
[7] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive Computing on the Grid Using AppLeS," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 4, pp. 369-382, Apr. 2003.
[8] Y. Brun, "Arithmetic Computation in the Tile Assembly Model: Addition and Multiplication," *Theoretical Computer Science,* vol. 378, no. 1, pp. 17-31, June 2007.
[9] Y. Brun, "Nondeterministic Polynomial Time Factoring in the Tile Assembly Model," *Theoretical Computer Science,* vol. 395, no. 1, pp. 3-23, Apr. 2008.
[10] Y. Brun, "Solving NP-Complete Problems in the Tile Assembly Model," *Theoretical Computer Science,* vol. 395, no. 1, pp. 31-46, Apr. 2008.
[11] Y. Brun, "Solving Satisfiability in the Tile Assembly Model with a Constant-Size Tileset," *J. Algorithms,* vol. 63, no. 4, pp. 151-166, 2008.
[12] Y. Brun, "Efficient 3-SAT Algorithms in the Tile Assembly Model," *Natural Computing,* vol. 11, no. 2, pp. 209-229, 2012.
[13] Y. Brun, G. Edwards, J. Young Bang, and N. Medvidovic, "Smart Redundancy for Distributed Computation," *Proc. 31st Int'l Conf. Distributed Computing Systems (ICDCS '11),* pp. 665-676, June 2011.
[14] Y. Brun and N. Medvidovic, "Mahjong: A sTile Framework for Distributing NP-Complete Computations Onto Untrusted Networks in a Trustworthy Manner," http://www.cs.umass.edu/brun/Mahjong, 2013.
[15] Y. Brun and N. Medvidovic, "Fault and Adversary Tolerance as an Emergent Property of Distributed Systems' Software Architectures," *Proc. Second Int'l Workshop Eng. Fault Tolerant Systems (EFTS '07),* pp. 38-43, Sept. 2007.
[16] Y. Brun and N. Medvidovic, "Keeping Data Private While Computing in the Cloud," *Proc. Fifth Int'l Conf. Cloud Computing (CLOUD '12),* pp. 285-294, June 2012.
[17] M. Chaisson, P. Pevzner, and H. Tang, "Fragment Assembly with Short Reads," *Bioinformatics,* vol. 20, no. 13, pp. 2067-2074, 2004.
[18] A.J. Chakravarti and G. Baumgartner, "The Organic Grid: Self-Organizing Computation on a Peer-To-Peer Network," *Proc. First Int'l Conf. Autonomic Computing (ICAC '04),* pp. 96-103, 2004.
[19] A.M. Childs, "Secure Assisted Quantum Computation," *Quantum Information and Computation,* vol. 5, no. 456, pp. 456-466, 2005.
[20] D. Dagon, G. Gu, C. Lee, and W. Lee, "A Taxonomy of Botnet Structures," *Proc. 23rd Ann. Computer Security Applications Conf. (ACSAC '07),* pp. 325-339, Dec. 2007.
[21] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Symp. Operating System Design and Implementation (OSDI '04),* Dec. 2004.
[22] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera, and J. Lepreau, "Flexlab: A Realistic, Controlled, and Friendly Environment for Evaluating Networked Systems," *Proc. Fifth Workshop Hot Topics in Networks (HotNets V),* pp. 103-108, Nov. 2006.
[23] S. Floyd and V. Paxson, "Difficulties in Simulating the Internet," *IEEE/ACM Trans. Networking,* vol. 9, no. 4, pp. 392-403, Aug. 2001.
[24] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int'l J. High Performance Computing Applications,* vol. 15, no. 3, pp. 200-222, 2001.
[25] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," *Proc. 41st Ann. ACM Symp. Theory of Computing (STOC '09),* pp. 169-178, 2009.
[26] A. Greenberg, "IBM's Blindfolded Calculator," *Forbes Magazine,* July 2009.
[27] A.S. Grimshaw, W.A. Wulf, and the Legion Team, "The Legion Vision of a Worldwide Virtual Computer," *Comm. ACM,* vol. 40, no. 1, pp. 39-45, 1997.
[28] "High Performance Computing and Communications," http://www.usc.edu/hpcc, 2013.
[29] Javelin Strategy & Research "2010 Identity Fraud Survey Report,"http://www.marketresearch.com/product/display.asp?productid=2592343, 2010.
[30] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home—Massively Distributed Computing for SETI," *Computing in Science and Eng.,* vol. 3, no. 1, pp. 78-83, Jan./Feb. 1996.
[31] O. Kullmann, "New Methods for 3-SAT Decisions and Worst-Case Analysis," *Theoretical Computer Science,* vol. 223, pp. 1-72, 1999.
[32] S.M. Larson, C.D. Snow, M.R. Shirts, and V.S. Pande, *Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology.* Horizon Press, 2002.
[33] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems," *IEEE Trans. Software Eng.,* vol. 31, no. 3, pp. 256-272, Mar. 2005.
[34] R. Motwani and P. Raghavan, *Randomized Algorithms.* Cambridge Univ. Press, 1995.
[35] A. Nakano, R.K. Kalia, P. Vashishta, T.J. Campbell, S. Ogata, F. Shimojo, and S. Saini, "Scalable Atomistic Simulation Algorithms for Materials Research," *Scientific Programming,* vol. 10, no. 4, pp. 263-270, 2002.
[36] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," *ACM SIGCOMM Computer Comm. Rev.,* vol. 33, no. 1, pp. 59-64, 2003.
[37] R.M. Robinson, "Undecidability and Nonperiodicity for Tilings of the Plane," *Inventiones Math.,* vol. 12, no. 3, pp. 177-209, 1971.
[38] S.M. Rubin, *Computer Aids for VLSI Design.* Addison-Wesley, 1994.
[39] M. Sipser, *Introduction to the Theory of Computation.* PWS Publishing, 1997.
[40] R.N. Taylor, N. Medvidovic, and E.M. Dashofy, *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, 2009.
[41] H. Wang, "Proving Theorems by Pattern Recognition," *II. Bell System Technical J.,* vol. 40, pp. 1-42, 1961.
[42] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 5, pp. 847-859, May 2011.
[43] Wikipedia, "SETI@home," http://en.wikipedia.org/wiki/SETI@home, 2008.
[44] E. Winfree, "Simulations of Computing by Self-Assembly of DNA," Technical Report CS-TR:1998:22, California Inst. of Technology, Pasadena, CA, 1998.
[45] G.J. Woeginger, "Exact Algorithms for NP-Hard Problems: A Survey," *Combinatorial Optimization - Eureka, You Shrink!,* vol. 2570/2003, pp. 185-207, 2003.
[46] Z. Yang, S. Yu, W. Lou, and C. Liu, "P^2: Privacy-Preserving Communication and Precise Reward Architecture for V2G Networks in Smart Grid," *IEEE Trans. Smart Grid,* vol. 2, no. 4, pp. 697-706, Dec. 2011.
[47] A.C.-C. Yao, "How to Generate and Exchange Secrets," *Proc. 27th Ann. IEEE Symp. Foundations of Computer Science (FOCS '86),* pp. 162-167, Oct. 1986.
[48] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving Secure, Scalable, and Fine-Grained Data Access Control in Cloud Computing," *Proc. IEEE INFOCOM,* pp. 534-542, 2010.

Yuriy Brun received the MEng degree from the Massachusetts Institute of Technology in 2003 and the PhD degree from the University of Southern California in 2008. He completed his postdoctoral work in 2012 at the University of Washington as a CI Fellow. He is currently an assistant professor in the School of Computer Science at the University of Massachusetts. His research focuses on software engineering, distributed systems, and self-adaptation. He is a member of the IEEE, the ACM, and the ACM SIGSOFT. More information is available at http:// www.cs.umass.edu/brun/.

Nenad Medvidovic received the PhD degree from the University of California, Irvine, in 1999. He is a professor in the Computer Science Department at the University of Southern California. He received the US National Science Foundation CAREER Award. His research focuses on the software architectures of large, distributed, mobile, and embedded systems. He is a member of the IEEE, the IEEE Computer Society, the ACM, and the ACM SIGSOFT. More information is available at http://sunset.usc.edu/neno/.