2012 IEEE Fifth International Conference on Cloud Computing

Keeping Data Private while Computing in the Cloud

Yuriy Brun
Computer Science & Engineering
University of Washington
Seattle, Washington, 98195, USA
brun@cs.washington.edu

Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, California, 90089, USA
neno@usc.edu

***Abstract*—The cloud offers unprecedented access to computation. However, ensuring the privacy of that computation remains a significant challenge. In this paper, we address the problem of distributing computation onto the cloud in a way that preserves the privacy of the computation's data even from the cloud nodes themselves. The approach, called sTile, separates the computation into small subcomputations and distributes them in a way that makes it prohibitively hard to reconstruct the data. We evaluate sTile theoretically and empirically: First, we formally prove that sTile systems preserve privacy. Second, we deploy a prototype implementation on three different networks, including the globally-distributed PlanetLab testbed, to show that sTile is robust to network delay and efficient enough to significantly outperform existing privacy-preserving approaches.**

I. Introduction

The emergence of cloud computing has allowed ubiquitous access to computation and data with higher availability and reliability than possible with personal machines and local servers. However, this transformation has created new challenges in computing. This paper addresses the challenge of *executing computations on untrusted machines in a trustworthy manner*. In particular, it focuses on preserving data privacy while solving computationally-intensive problems on untrusted machines, such as those in the cloud.

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 [34], in this paper, we present a prototype implementation that solves the NP-complete problem 3-SAT. Our approach is directly applicable to solving other NP problems, while future work is required to expand sTile to other computations.

sTile explores the fundamental cost of privacy through data distribution. Existing approaches to using the Internet's computational resources to perform NP-complete computations [9], [20], [26] have resulted in commercial enterprises deployed on over a million machines [3]. 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.

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, to empirically demonstrate sTile's feasibility, we deploy a prototype implementation on three distinct networks, including the globally-distributed PlanetLab testbed [33]. Third, we formally analyze the communication and computation costs induced by sTile, provide bounds on them, and empirically verify those bounds. We have previously discussed sTile's ability to handle faults and malicious attacks [15], [16], and do not focus on that dimension here. sTile significantly outperforms existing cryptography-based privacy techniques, such as homomorphic encryption [23].

The rest of this paper is structured as follows: Section II explains sTile through an example. Section III details the sTile approach. Section IV formally analyzes sTile's privacy-preservation. Section V discusses sTile's empirical experiments. Section VI positions our work in terms of related research. Finally, Section VII summarizes the contributions of the paper.

II. Motivating Example: Private Addition

sTile computes while preserving 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.

A. 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.

Figure 1(a) 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,

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.126

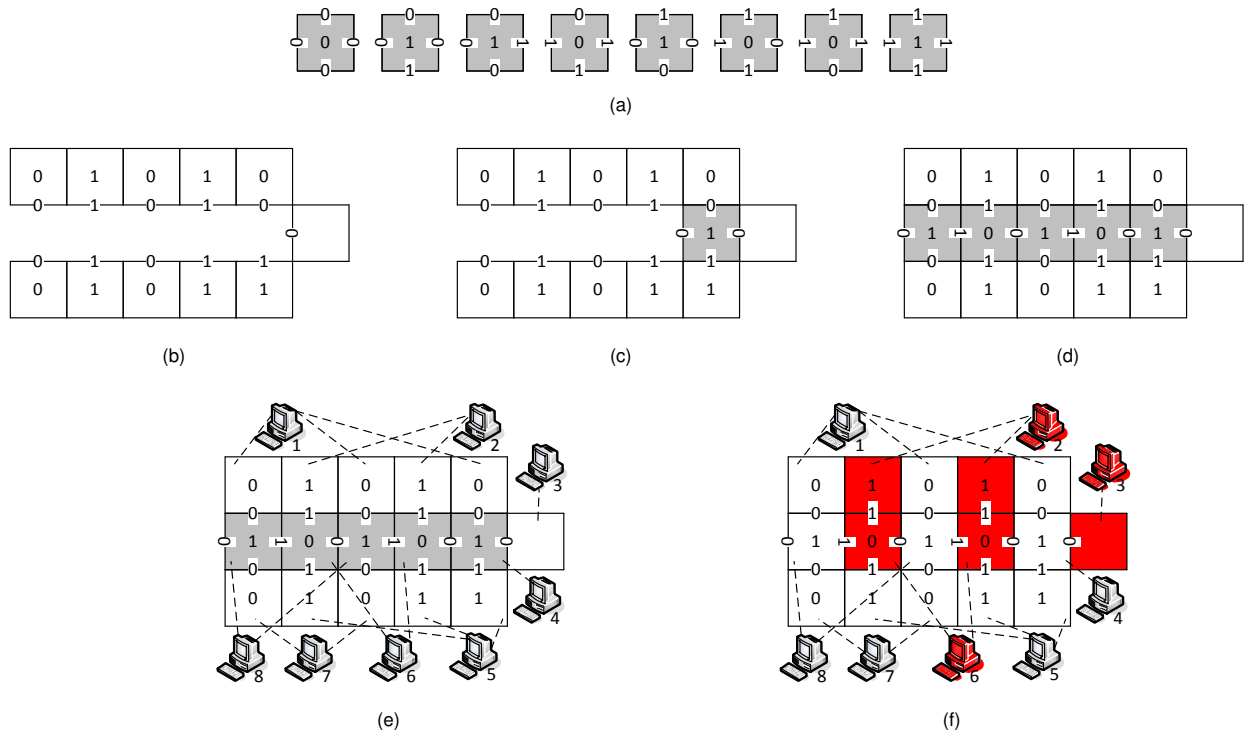

Figure 1. An adding tile assembly with (a) eight tile types. A seed crystal (b) encodes the inputs, $10 = 1010_2$ and $11 = 1011_2$. The first attaching tile (c) adds the least significant bit of each input. The middle row of the final crystal (d) encodes the output $21 = 10101_2$. sTile deploys (e)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

one bit at a time. Figure 1(b) 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 Figure 1(a) matches the seed crystal on three sides, that tile instance attaches to the crystal. Figure 1(c) 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 Figure 1(d).

We designed this addition tile assembly by hand. In practice, we do not expect sTile developers to program using tile assemblies. Section III will discuss compiling distributed systems without understanding tile assemblies.

B. The Distribution Process

sTile uses the theoretical tile assembly to decompose a computation into small parts. Each small part represents a tile. Figure 1(e) 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 III-B1). 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 III-B1). Each node knows only the tile instances it is deploying and maintains references to the geometrically adjacent tile instances on other nodes. Next, tiles with an empty neighbor location coordinate with their neighbors to recruit matching tiles to attach (described in Section III-B3). This process uses a secure multi-party computation algorithm to ensure neighbors do not learn each other's data [41]. (While a single seed is sufficient for addition, for NP-complete problems, sTile employs distributed seed replication, described in Section III-B4.) Each of these steps replies on an algorithm that ensures the tiles are deployed uniformly randomly on the available nodes (described in Section III-B2). Once the execution finishes, the tiles in the middle row report the solution to the client, indicating their neighbor's tiles' nodes' IDs (e.g., IPs), which the client uses to reconstruct the output.

C. 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, Figure 1(f) 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

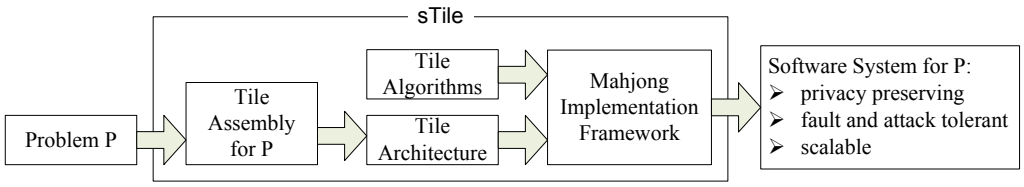

Figure 2. A high-level overview of sTile.

(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 will show in Section IV, as long as the adversary controls less than half the network, the probability it can reconstruct the input is prohibitively small. In the $10+11$ example, it is easy to see that controlling half the nodes translates to controlling roughly half the tiles, which is unlikely to be sufficient to reconstruct the input or the output.

III. sTile: An Approach to Preserving Privacy

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. Figure 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 [12], [13], [14] and factor integers [11]. 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. Finally, tile assemblies are Turning-universal [34], so future extensions of sTile can be made to perform arbitrary computations and to automatically compile programs into tile assemblies.

This paper focuses on 3-SAT to demonstrate using sTile to solve NP-complete problems. Thus, we describe how sTile uses a tile assembly for 3-SAT (described in Section III-A) to create a tile architecture (described in Section III-B). sTile then uses that tile architecture and the tile algorithms (also described in Section III-B) to compile a Mahjong-based implementation of the system (described in Section III-C), which is the software system used to distribute the 3-SAT computation onto a cloud, in a privacy-preserving manner.

To build sTile systems, a developer does not need to understand the underlying theoretical model described in Section III-A. Due to space limitations, our description of this model is at a high level; the reader can refer to [13] for details and proofs that the theoretical model solves 3-SAT. The process we describe here allows the developer to automatically compile a tile assembly description (such as the 3-SAT assembly [13]) into a distributed, privacy-preserving software system.

A. Computing with Tiles

A key component of sTile is the tile architecture, which is based on a tile assembly, an extensively studied mathematical object [34], [36], [38]. As mentioned, our own previous work has developed tile assemblies to efficiently solve NP-complete problems [12], [13], [14]. Others have shown that tile assemblies are Turing-universal [34], [38], [1]. 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.

The set of tile types in a tile assembly encodes the "program" the tiles will execute. For example, the eight tile types from Figure 1(a) encode integer addition. The assembly that solves 3-SAT has 64 tile types [13]. Figure 3 shows one possible example execution (crystal) of that tile assembly. The clear tiles on the bottom and right edges encode the input: $(x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$. The shaded tiles then attach, growing the crystal to nondeterministically select a truth assignment and compute whether that assignment satisfies the formula. The crystal in Figure 3 depicts an assignment that satisfies the formula, indicated by the ✓ tile in its top left corner. To solve 3-SAT, many such crystals must self-assemble in parallel, each exploring a different assignment nondeterministically.

Next, in Section III-B, we describe how sTile uses a network of computers to reify tile assemblies, resulting in a software system. In subsequent sections, we argue that such systems are efficient and possess properties that are important on large public networks such as clouds.

B. Tile Architecture and Algorithms

A sTile system is a software system that uses a network of potentially untrusted computers to solve a computational problem in a privacy-preserving manner. Intuitively, the cloud will simulate a tile assembly: each computer in the cloud will deploy tile instances, and will communicate with other computers to self-assemble a solution to a computational problem following the rules of the tile assembly. Thus, a tile architecture is based on a tile assembly; the software system employing

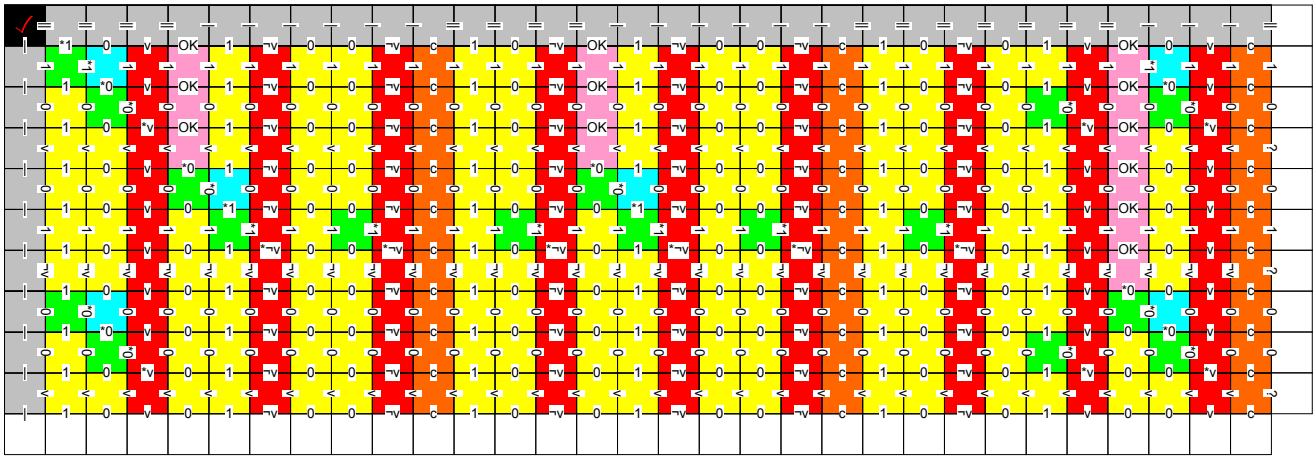

Figure 3. An example crystal of a 3-SAT-solving tile assembly. Here, the clear tiles along the bottom encode the input Boolean formula $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$. This crystal represents one possible nondeterministic execution that checks whether the assignment $x_0 = x_2 = \mathit{TRUE}$ and $x_1 = \mathit{FALSE}$, encoded in the clear tiles on the right, satisfies ϕ. Because it does, the ✓ tile attaches in the top left corner.

that architecture solves the particular computational problem the tile assembly solves.

The components of the tile architecture are instantiations of the tile types of the underlying assembly. A sTile 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 in the cloud will contain these components, and components that are adjacent in a crystal can recruit other components to attach, thus dynamically completing the architectural configuration [35] 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 architectural configuration. The input consists of a seed crystal, such as the clear tiles along the right and bottom edges in Figure 3. Figure 4 summarizes the algorithms a sTile system follows to find a solution. During *initialization*, the system sets up a single input seed crystal on the cloud. The seed then *replicates* to create many copies, and each of the copies *recruits* tiles to assemble larger crystals and eventually produce the solution. The solution tile components (e.g., the ✓ component for the 3-SAT assembly) then report their state to the client.

We now elaborate on these operations.

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.

A tile type map maps node IDs (e.g., all 128-bit IP addresses) to tile types. It determines the type of tile components each computer 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 3-SAT, there are 64 different tile types, so the tile type map would partition 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 the provisioned subset of the nodes in the cloud, is small. For 3-SAT, the map is sixty four 128-bit numbers. The client node, and subsequently the rest of the network, distribute the map by using a standard gossip protocol, which takes $\Theta(\log N)$ time, for a network of N nodes.

The client is also responsible for creating the first seed on the cloud. For each tile in the seed, the client selects a node that deploys that tile type (as described next), 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 less network communication than the distribution of the tile type map.

2) Discovery: 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.

Each node keeps a *node table* of three IP addresses of other nodes that deploy each component type. When queried for a

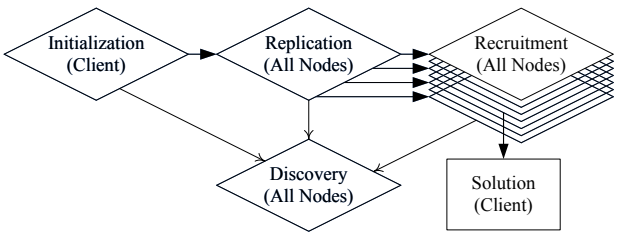

Figure 4. Overview of sTile algorithms.

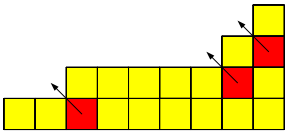

Figure 5. Tile components with both upper and left neighbors (highlighted in the diagram) can recruit new components to attach to their upper left.

node of a given type, the node will select and return one of the three entries at random, and replace it's three entries with the selected node's list. This table is small ($64 \times 3 = 192$ IPs for 3-SAT), and the query procedure takes $\Theta(1)$ time. Repeated queries emulate a random walk on a graph on which each node has three random neighbors. Such a walk mixes rapidly, which implies that after $\Theta(\log N)$ queries, every IP is equally likely to be returned [31] (formal proof omitted for space).

3) Recruitment: The seed crystal grows into a full assembly by recruiting tile attachments. Every tile component that has both an upper and a left neighbor recruits a new tile to attach to its upper left corner. Figure 5 indicates several places in a sample crystal where tile components are ready to recruit new tiles. A recruiting tile component X uses node discovery to pick a potential attachment node of each tile type and sends those nodes attachment requests. An attachment request consists of the X's upper neighbor's left interface and left neighbor's top interface. If those interfaces match the right and bottom interfaces, respectively, of the potential attachment tile Y, Y attaches. X then informs Y of the IPs of its two new neighbors, and those neighbors of Y's IP. X can perform this operation without ever learning its neighbors' interfaces by using Yao's garbled protocol [41].

Each component's recruitment is a five-step process: (1) X asks N (its upper neighbor) to encode its left interface, (2) N asks W (X's left neighbor) to encode its top interface, (3) W responds to X, (4) X sends attachment requests to a set of potential attachments Y, and (5) those Ys reply to X. We will analyze these five steps in Section V-D.

In the 3-SAT system, the successful crystal recruits 310 tile components (non-clear tiles in Figure 3). An unsuccessful crystal, which we discuss further in Section III-B5 can recruit fewer, but no more than 310 tiles.

4) Replication: Whenever cloud nodes have extra cycles they are not using for recruitment, they replicate the seed. Using node discovery, each node X selects another node Y that deploys the same tile type as itself, and sends it a replication request consisting of up to two IP addresses of X's neighbors. X informs its neighbors that Y is X's replica (by sending Y's IP address to X's neighbors). Those neighbors, when they replicate using this exact mechanism, will send their replicas' IP addresses to Y. Thus, the entire seed replicates. Each component's replication is a three-step process: (1) X sends a replication request to Y, (2) Y replies to X, and (3) X tells its neighbors about Y. We will analyze these steps in Section V-D.

At the start of the computation, the seeds replicate exponentially. When there are sufficiently many seeds to keep the nodes occupied recruiting, replication naturally slows down because replication only occurs when cycles are not occupied by recruitment. 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 sends a small "STOP" packet to all its neighbors, which they forward to their neighbors, and so on. Since the diameter of a large connected network of N nodes with randomly distributed connections is $\Theta(\log N)$ [31], the "STOP" message will propagate in $\Theta(\log N)$ time.

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. When there is no satisfying assignment, no crystal can claim to have found the solution. Rather, the exhaustive exploration of all assignments finds the solution. After exploring 2^n randomly-selected assignments, the certainty that the formula is unsatisfiable is at least $\left(1-e^{-1}\right)$. As more assignments are explored, the certainty grows exponentially quickly towards 1. As an example, after exploring 80 assignments for the example 3-SAT problem from Figure 3, the probability that the answer is not found is less than $e^{-10} < 10^{-4}$. This amounts to fewer than 25,000 tile components, which even a single computer can easily deploy.

C. Mahjong Implementation Framework

The final element of sTile is the Mahjong implementation framework. The framework [10] 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 III-B, and outputs (i.e., deploys) a complete sTile software system. Mahjong 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 adapt Mahjong to distinct networks and a developer unfamiliar with the project was able to deploy it without consultation with the authors.

IV. Privacy Preservation

In this section, we formally argue that sTile systems preserve privacy. That is, we prove 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 Figure 3) 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 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 neighboring tiles and not of its 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. Theorem 1 bounds the probability that an adversary can use this scheme to learn the input.

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

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 Figure 6 graphically illustrates that the probability of input reconstruction drops exponentially in c. Let us examine a sample scenario. 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. Let us also suppose a powerful adversary has gained control of ≈125,000 machines ($\frac{1}{8}$ of the network). The adversary will be able to reconstruct the seed with probability $1-\left(1-2^{-114}\right)^{2^{38}} < 10^{-22}$. As the input size increases, this probability further decreases. The probability decays exponentially for all $c < \frac{1}{2}$.

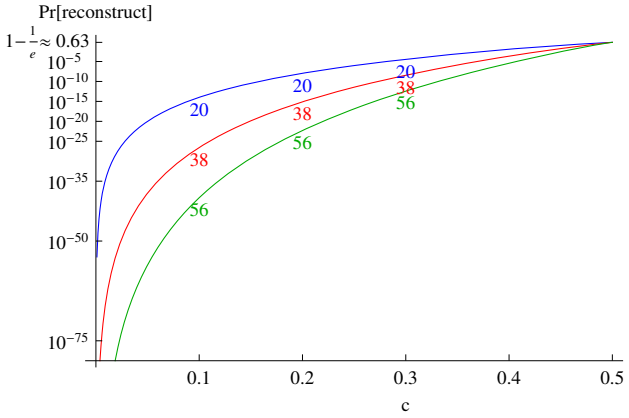

Figure 6. The probability that an adversary controlling c fraction of the network can reconstruct an entire 20-, 38-, and 56-bit input.

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., three-bit) fragments of the input, but the information contained in those fragments is greatly limited and cannot be used to reconstruct larger fragments [17].

V. Computational Feasibility

In order to demonstrate that sTile is a feasible solution for building software systems that distribute computationally intensive problems on clouds, we must show that (1) such systems are robust to network delay, and (2) real-world-sized problems can be solved on real-world-sized networks in reasonable time. In separate experiments not presented here, we also verified that the computational speed of sTile systems scales linearly with the number of nodes it is deployed on.

We have built two Mahjong-based implementations and a simulator called Simjong. The former distribute computation onto physical networks and the latter is a discrete-event network simulator that employs accurate models of network message delays. These implementations establish the correctness of our algorithms and demonstrate the distribution of a sTile system onto a physical network. Simjong's goal is to accurately simulate very large networks (e.g., 1,000,000 nodes).

A. Prototype sTile Implementations

We have built two instances of the Mahjong framework for NP-complete problems (3-SAT and SubsetSum) and a discrete-event simulator Simjong, with network-delay simulation capabilities. These implementations are available for download [10]. Simjong simulates delays for messages sent between nodes: a constant (e.g., 100ms), chosen at random from some distribution (e.g., Gaussian around 100ms with $\sigma = 20$ms), or proportional to the geographic distance between nodes. Simjong's network model is a simplification of the network simulator standard ns-2 [21] because it abstracts

away the exact topology of the network, which is not important for our needs.

B. Experimental Setup

We use 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 [27], whose nodes were distributed in several locations in a city; and (3) a 100-node subset of PlanetLab [33], 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 here 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 goal was to verify sTile's robustness to network delay. Our experiments had two independent variables — the communication delay, and the computation size — and one dependent variable — computation time.

First, to verify sTile's correctness, we solved over 100 SubsetSum and 3-SAT problems, including $\mathfrak{A}$, $\mathfrak{B}$, and $\mathfrak{C}$. As a rule of thumb, we chose problem instances to each execute in under 4 hours on our 186-node cluster. We verified that on all networks, the solutions were correct and the execution exhibited only the expected communication and connections.

C. Robustness to Network Delay

To measure the effect of network delay, we compared sTile execution times on equal-sized subsets of the three networks. We then compared Simjong execution times on six virtual networks of 1,000,000 nodes, with respective network delays of 0ms, 10ms, 100ms, 500ms, drawn from a Gaussian distribution around 100ms with $\sigma = 20$ms, and ones proportional to the geographic distance between randomly assigned world-wide locations (varying from 20ms to 500ms). Simjong deployed the first 10^{-4}% of the seeds to estimate the entire execution time.

Hypothesis: *Execution time is independent of network delay.*

The intuition behind our hypothesis is that each cloud node handles the deployment of thousands of lightweight tiles and whenever a packet travels between nodes, nodes handle other tiles rather than waiting idly for the network communication to arrive. Thus, network delay affects the latency but not the throughput of sTile systems. Figure 7 confirms the hypothesis. We found that the execution times were closely clustered and no statistical significance in the small variances.

Prob.	# of Nodes	Delay	Execution Time
Mahjong			
$\mathfrak{A}$	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.

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

D. Efficiency

The final claim we address in demonstrating sTile's feasibility is that real-world-sized problems can be solved on real-world-sized networks (such as clouds) in reasonable time. In particular, we posit that sTile systems can outperform solving the problem on a private machine.

Suppose Mahjong solves an n-variable, m-clause 3-SAT problem on N nodes. 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 3) and no more than $3nm\lg^2 n$ recruited tiles (non-clear tiles in Figure 3). 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 III-B4, 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 III-B3, each concluded by sending a single network packet; let the time for these operations be denoted as $5u$. Equation (1) describes this system's overall execution time.

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

$$2^n(n+3m)r \quad (2)$$

Now suppose a user wishes to solve the 3-SAT instance on a single computer using the same algorithm that explores 2^n possible assignments. 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. sTile's overhead is the ratio of (1) and (2). Assuming $m > n$ and $i = u = r$ (which we verified empirically but omit the evidence here) the over-

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.

Figure 8. Comparison of execution time for solving $\mathfrak{D}$ as measured by Simjong and estimated by Equation (1).

head is no greater than $\frac{8n\lg^2 n}{N}$. In other words, if the size of the public network exceeds $8n\lg^2 n$, Mahjong will execute faster than a single machine. For the sizes of problems we discuss next, that network size is several thousand nodes. We note that it is also possible to use a small private network, as opposed to a single private computer. Using M private nodes would at best improve the single-computer approach by a factor of M, and thus, in the worst case, would increase the overhead by that same factor.

We verified the accuracy of Equations (1) and (2) by comparing them to actual measured running times. Figure 8 highlights some of the results — Simjong running times and Equation (1) values for $\mathfrak{D}$. We consistently found that Equation (1) was within 8% of the Simjong-measured execution times. As expected, we found that for large networks, sTile systems outperform the single private machine approach. For example, solving a 38-variable, 100-clause 3-SAT instance on a single computer takes 3.3×10^7 seconds ≈ 1 year. Mahjong solves the same problem on a million-node network in 1.8×10^5 seconds ≈ 2.1 days.

E. 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 comparing sTile and other approaches to solving 3-SAT, we have used simple underlying algorithms (e.g., exploring 2^n assignments). Efficient SAT solving is a popular area of research and much faster algorithms exist. We have focused on simple algorithms only for clarity of explanation and ease of the prototype implementation. sTile can also use the more complex, efficient algorithms, although we omit that description here. For example, we have already made significant progress toward implementing a sTile system that explores only $O(1.8394^n)$ assignments to solve 3-SAT [14].

We have taken into account accurate models of network delay. However, we have assumed the volume of network traffic created by sTile systems will not affect message delivery. While our experiments suggest that Mahjong traffic volumes are not significantly larger than typical, this assumption may not hold for some networks or clouds.

VI. Related Work

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

A. Distributing 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 [22]. Among systems that concentrate on distributed computation are BOINC systems [3] (such as SETI@home [28] and Folding@home [30]), MapReduce [20], and the organic grid [18]. A unique approach — FoldIt — uses the competitive human nature to solve the protein-folding problem [5]. 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.

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 [29], [39], to using runtime information to dynamically improve the speed of *SAT* solvers [7], to leveraging local message-passing protocols such as MPI and OpenMP to use small clusters to linearly accelerate the computation of specialized problems [32]. This work is not in competition with our technique, but is rather complementary. The tile architecture is based on a Turing-universal computational model [8], [34] 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 [14]. 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.

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 [20] 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 [6]; as many as a thousand MapReduce jobs are

executed on Google's and Amazon's clusters daily [2], [20]; and Facebook uses a MapReduce system to process more than 15 terabytes of new data every day [43]. 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.

B. Cloud Privacy

Cloud computing has re-emphasized the importance of data privacy, causing the emergence of numerous approaches for keeping data private on the cloud [4], [37], [40], [42]. Most such approaches concentrate on private data *storage* and security policies around user-authorized data *retrieval* and require some trusted agents [4], [37], [40] whereas our work concentrates on preserving privacy during *computation* and requires no trusted agents. One approach [42] that begins to explore the notion of performing computation on private data uses attribute-based encryption and proxy re-encryption to distribute client data onto potentially untrusted cloud servers. In contrast, sTile computes more complex functions (ones that require knowing the entire input), requires no trusted agents, and does not disclose the data to the underlying agents.

C. Privacy-Preserving Computation

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 [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.

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 [23]. 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 [25]. 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 V, 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 [41]. Zero-knowledge compilers bridge that work closer to our approach by making Yao's protocol secure even if the parties cannot be trusted [24]. 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 III-B3.

VII. Contributions

Sensitive computation, such as tax calculations, is rapidly moving onto the cloud, raising the issue of data privacy of not only storage but also computation. In this paper, we have presented sTile, a technique for distributing computation onto the cloud while keeping the data private, both, from intruders and from the cloud nodes themselves. Here, we have focused on solving only NP-complete problems, and have demonstrated our approach through solving the well-know NP-complete problem 3-SAT. Future work remains on applying sTile to a larger class of problems, which is likely feasible because sTile's underlying theoretical model is Turing-universal.

We have evaluated sTile theoretically and empirically. First, we formally proved that sTile systems preserve privacy as long as no adversary controls one half of the cloud executing the computation. Second, we deployed a sTile prototype called Mahjong on three networks, including the globally-distributed PlanetLab testbed. We used these deployments to demonstrate that Mahjong (1) is robust to network delay, and (2) significantly outperforms existing privacy-preserving approaches, such as homomorphic encryption and computing on privately-owned, secure hardware.

Acknowledgments

We would like to thank Jae young Bang for his help deploying Mahjong-based implementations on PlanetLab.

References

[1] L. Adleman, J. Kari, L. Kari, and D. Reishus, "On the decidability of self-assembly of infinite ribbons," in *FOCS*, Ottawa, Ontario, Canada, 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 *IEEE/ACM GRID*, Pittsburgh, PA, USA, 2004, pp. 4–10.

[4] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved proxy re-encryption schemes with applications to secure distributed storage," *ACM TISSEC*, vol. 9, pp. 1–30, 2006.

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

[6] 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.

[7] A. Balint, M. Henn, and O. Gableske, "A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem," in *SAT*, Swansea, UK, 2009, pp. 284–297.

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

[9] 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 TPDS*, vol. 14, no. 4, pp. 369–382, 2003.

[10] Y. Brun, "Mahjong tile style implementation," http://csse.usc.edu/~ybrun/Mahjong.

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

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

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

[14] ——, "Efficient 3-SAT algorithms in the tile assembly model," *Natural Computing*, vol. in press, 2012, doi: 10.1007/s11047-011-9299-0.

[15] Y. Brun, G. Edwards, J. young Bang, and N. Medvidovic, "Smart redundancy for distributed computation," in *ICDCS*, Minneapolis, MN, USA, 2011, pp. 665–676.

[16] Y. Brun and N. Medvidovic, "Fault and adversary tolerance as an emergent property of distributed systems' software architectures," in *EFTS*, Dubrovnik, Croatia, 2007, pp. 38–43.

[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," in *ICAC*, New York, NY, USA, 2004, pp. 96–103.

[19] A. M. Childs, "Secure assisted quantum computation," *Quantum Information and Computation*, vol. 5, no. 456, pp. 456–466, 2005.

[20] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, San Francisco, CA, USA, 2004.

[21] S. Floyd and V. Paxson, "Difficulties in simulating the Internet," *IEEE/ACM TNET*, vol. 9, no. 4, pp. 392–403, 2001.

[22] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *Intl. Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[23] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM STOC*, Bethesda, MD, USA, 2009, pp. 169–178.

[24] O. Goldreich, S. Micali, and A. Wigderson, "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.

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

[26] A. S. Grimshaw, W. A. Wulf, and the Legion team, "The Legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, 1997.

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

[28] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home — massively distributed computing for SETI," *IEEE MultiMedia*, vol. 3, no. 1, pp. 78–83, 1996.

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

[30] 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.

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

[32] 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.

[33] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 59–64, 2003.

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

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

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

[37] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," *IEEE TPDS*, vol. 22, pp. 847–859, 2011.

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

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

[40] 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 Transactions on Smart Grid*, 2011.

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

[42] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *INFOCOM*, San Diego, CA, USA, 2010, pp. 534–542.

[43] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user MapReduce clusters," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, 2009.