# Outsourcing Power System Simulations

Yue Tong, *Student Member, IEEE,* Jinyuan Sun, *Member, IEEE,* Kai Sun, *Member, IEEE* Pan Li, *Member, IEEE*

***Abstract*—The advancement of cloud-computing technologies opens new possibilities to outsource to the third-party cloud the computation-intensive and time-consuming dynamic simulations needed in power grid system research and operations. Outsourcing makes it possible to conduct dynamic simulations much faster and with lower cost than to keep all computations local. On the other hand, outsourcing, however, also gives rise to the risk of information leak, as the outsourced simulation contains sensitive information, such as critical operational parameters and projected states of the power grid. In this paper, a novel secure outsourcing scheme, combining disguising technique and code obfuscation, was proposed to enable efficient outsourcing while preserving the confidentiality of the information. It was shown that our scheme can limit the adversary's capability to obtain the sensitive information in the context of outsourcing of power system dynamic simulations.**

***Index Terms*—security, data confidentiality, scientific computation, cloud computing, outsourcing, disguising technique, code obfuscation**

## I. Introduction

Dynamic simulations play critical roles in power system research and operations. They are used to predict the dynamic behaviors of power systems under contingencies such as generator tripping, line switching, and short circuit [2]. In real practice, due to the heavy computational burden of simulating large scale power systems, dynamic simulation is currently conducted offline on hourly or daily basis, making it hardly useful in practice to respond to emergent contingencies.

Recently, outsourcing the dynamic simulations to the cloud has emerged as a promising solution to the problem mentioned above. Pilot studies done in [6] showed that by outsourcing the heavy computational burden to the cloud, it is possible to conduct power system simulations, not only much faster but also with less cost. According to [6], an N-1-1 contingency analysis with 4,100 scenarios, which would have taken 1,700 hours at a commodity laptop, or 40 hours at the internal computing cluster of 40 cores, now needs a running time of only 1.5 hours with 150 Amazon EC2 nodes for a total monetary cost of about $60.

Despite that cloud computing has demonstrated its tremendous potential, the fact that outsourcing requires dynamic simulation to take place in an external and a potentially malicious facility (the cloud) gives rise to concerns about the information security during the course of outsourcing, considering that both information required by and the results produced from the dynamic simulation are sensitive and private to the power grid owners. These concerns have restrained the utility companies from taking advantages of the aforementioned outsourcing paradigms. Instead, they would opt for conservative measures such as keeping all the computations local in exchange for the absolute data privacy assurance. The primary goal of this paper is to explore the possibility to outsource the computation overhead without giving away data privacy.

Yue Tong, Jinyuan Sun, and Kai Sun are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, 37921 USA e-mail: {ytong3, jysun, ksun}@utk.edu

P. Li is with the Department of Electrical and Computer Engineering, Mississippi State University.

## II. Related works

Our work falls under the broad topic of non-interactive outsourcing of scientific computation. Early research by [14] showed that any function can be securely evaluated by using garbled circuit. More recently, with the breakthrough of fully homomorphic encryption, homomorphic encryption appears as the most straightforward solution to computation outsourcing. Informally, homomorphic encryption allows one (the cloud in our case) to execute encrypted software over encrypted data to generate an encrypted result, which can be decrypted only by outsourcers later. The problem is, however, even the state-of-the-art fully homomorphic encryption [5] is too inefficient for practical uses. Relatively more efficient somewhat homomorphic encryption [7] only supports a very limited number of additive or multiplicative operations. As a result, the prospect of directly applying homomorphic encryption to secure the outsourcing of dynamic simulation is remote. Aside from homomorphic encryption, cryptographic tools are also invented to facilitate the outsourcing of specific computations like ranked keyword search [11], multiparty back-propagation[15], and ridge regression [8]. However, they focus on specific applications and cannot be readily applied to dynamic simulations. Additionally, these schemes utilize heavy cryptographic primitives like public key cryptography and pairing-based cryptography, which introduce significant computation overhead.

There are also many works on outsourcing of computations that do not heavily rely on cryptographic tools. Wang *et al.*[12] investigates the problem of cloud-based outsourcing of linear programming for large-scale systems. Their proposal, however, dealt with only linear programming while the computations in dynamic simulations are based on nonlinear differential algebraic equations. Atallah *et al.*[1] studies extensively the problem of secure outsourcing of scientific computations such as sorting, template matching, string pattern matching, and differential equations. However, dynamic simulation is not considered.

## III. Problem Statement

### A. System Model and Threat Model

*1) System Model:* Figure. 1 illustrates the conceptual system model, consisting of two entities: the user and the cloud. The user wishes to outsource to the cloud power system

978-1-4799-5952-5/15/$31.00 ©2015 IEEE

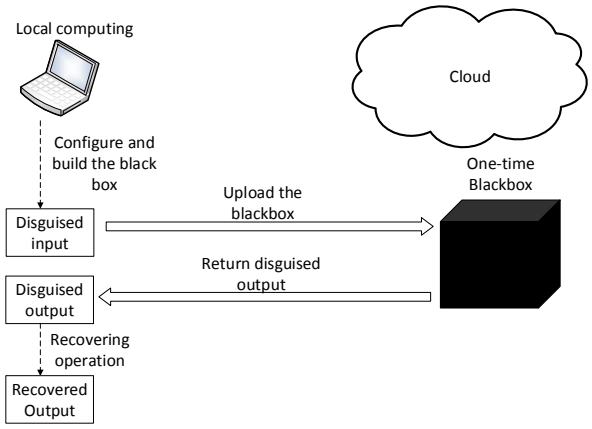



Fig. 1: Ideal system model for secure computation outsourcing

dynamic simulations with privacy preservation, by uploading a configured black box to the cloud. An ideal black box is a piece of executable software that encapsulates all the information it requires, except for the external input, to complete the computation while looking "unintelligible" to the cloud. The cloud performs all computations through the black box. To preserve the privacy of both the input and output values, the black box is constructed in such as way that it takes protected inputs and produce protected outputs, which will be recovered by the user later.

As data confidentiality is of the primary interest of this work, we enumerate in the following the private data involved in a typical dynamic simulation.

- **Power System Models:** these are the mathematical models, usually nonlinear differential algebraic equations, of machines and control systems, such as generators, governors, motor load.
- **Model data:** the specific parameters and configurations with respective to power system models, including the bus connectivity and information about where and when to apply faults. Model data are usually measured, validated or defined by utility planning engineers.
- **Operating conditions of the system:** the values of system states at the beginning of the simulation. They are updated based on the real-time data from the SCADA (Supervisory Control and Data Acquisition)/EMS (Energy Management System) in the control room.
- **Simulation results:** the output of the contingency simulation in the form of trajectories of state variables over the simulated time.

Data are divided into two categories: input data and output data. Input data are defined as the data submitted to the cloud so that the simulation can be conducted on the cloud end. Output data refer to any information that the simulation produces as the result of the dynamic simulation. Naturally, all the kinds of data listed above except for the simulations results belong to the input data. Simulation results are the only kind of output data. Both input data and output data are at risk of unauthorized disclosure if left unprotected.

*2) Threat model and security goals:* In the threat model, the adversary is a semi-honest cloud service provider, who is not entirely trusted by the user in the sense that the cloud service provider will honestly carry out the delegated computation tasks and deliver the result of the computations, but it is curious about the computation happening in the cloud, and may try to compromise the user's data privacy by eavesdropping any data stream that flows in and out of the cloud. The security goal of our work is to limit the adversaries' ability to compromise the users' data confidentiality, which, in this particular context, is defined as not any private data listed in Section.III-A1 can be gleaned from the data sent to and produced by the cloud.

### B. Background on Power System Dynamic Simulations

Dynamic simulation refers to a computer-based approach to study a system's dynamic behavior as a function of time [16]. The key idea is to describe the system with a set of mathematical equations, where the variables are the time-varying states of the system. The dynamic states are obtained by solving the equations over the simulated time. As the equations are usually nonlinear, for example, ordinary differential equations or partial differential equations, solving them requires numerical methods such as Euler method, Runge-Kutta method, trapezoidal method, and so forth. In the power system research and operations, dynamic simulations are used to simulate the time-varying trajectories of the state variables of a power system, such as machine speeds, rotor angles under certain initial conditions and disturbances.

## IV. Secure outsourcing schemes

### A. Scheme Overview

Our scheme is comprised of the following functions.

- $\mathsf{ProbEnc}(\mathbf{\Phi}, \mathbf{\Psi}) \rightarrow \mathbf{\Psi}'$. This function takes as input $\Psi$, the original simulation in the form of source code, and $\mathbf{\Phi}$, the spline functions used to disguise the simulation results. The function applies the disguising technique to the state variables by modifying simulation source code as explained in the next section. The function then conducts code obfuscation to the modified source code in order to generate the encrypted simulation source code, which is output as $\mathbf{\Psi}'$.
- $\mathsf{SimulationExec}(\mathbf{\Psi}') \rightarrow \mathbf{S_{disguise}}$. This function takes as input and executes the source code of an encrypted simulation, and produces the disguised simulation result $\mathbf{S_{disguise}}$.
- $\mathsf{ResultRec}(\mathbf{S_{disguise}}, \mathbf{\Phi}) \rightarrow \mathbf{S}$. This function takes as input $\mathbf{S_{disguise}}$ and $\mathbf{\Phi}$. It recovers the original simulation result by transforming the disguised result back to the original one.

In accordance with the system model illustrated in Section III-A1, the user performs SplineGen and ProbEnc, which are discussed in Section IV-B and Section IV-C, respectively, to produce the encrypted simulator. It then uploads the encrypted simulator to the cloud. The cloud performs $\mathsf{SimulationExec}(\mathbf{\Psi}')$ and returns the disguised trajectories to the user. Finally, the user calls ResultRec to recover the right trajectories from the disguise ones.

*B. Protecting the simulation results with the disguising technique*

In outsourcing, the simulation result, the trajectories of system variables, is first been by the cloud once it is calculated even before the simulation results are returned to the user. So a trajectory must be protected in such a way that when a point is calculated out, its exposure to the cloud does not reveal the point's real values during and after the simulation to the cloud. To this end, considering the form of the output data, we adopt the disguising technique to mask the trajectories while the simulation is still in progress. That is, original trajectories are transformed into another form so that their actual values remain secret to the cloud. Later, only the one who knows how the transformation can undo the applied transformation and recover the original trajectories. We elaborate this idea as follows.

*1) Using injective mappings to disguise the DAE's solution:* Dynamic simulations, in essence, solve a set of differential-algebraic equations (DAE from now on) in order to get the trajectories of state variables that describe the system's behavior. In order to disguise the trajectories, the original DAEs need to be transformed accordingly in the first place.

More specifically, DAEs can be generally represented as

$$\mathbf{F}(\dot{\mathbf{x}}(t), \mathbf{x}(t), t) = 0$$

where $\mathbf{x}$, *i.e.*, the solution of the DAEs, is a time-varying vector holding $n$ state variables. To hide the original solution, *i.e.*, the trajectory $\mathbf{x}(t)$, we define an injective mapping $\mathbf{sm} : \mathbf{x_{disguised}}(t) \rightarrow \mathbf{x}(t)$ such that $\mathbf{sm}[\mathbf{x}](t) = \mathbf{x_{disguise}}(t)$. The disguised DAEs, as a result of the replacing $\mathbf{x}(t)$ with $\mathbf{x_{disguise}}(t)$ will then be outsourced in place of the original DAEs as:

$$\mathbf{F_{disguised}}((\dot{\mathbf{x}}_{\mathbf{disguised}}(t), \mathbf{x_{disguised}}(t), t) = 0$$

which is equivalent to:

$$\mathbf{F}(\mathbf{sm}^{-1}[\dot{\mathbf{x}}_{\mathbf{disguised}}](t), \mathbf{sm}^{-1}[\mathbf{x_{disguised}}](t), t)) = 0$$

where $\mathbf{sm}^{-1}$ denotes the inverse function. Since $\mathbf{sm}$ and $\mathbf{sm}^{-1}$ are generated and kept secret by the user, after the cloud returns the disguised solution $\mathbf{x_{disguised}}(t)$, only the user can recover the original trajectories by computing $\mathbf{x}(t) = \mathbf{sm}^{-1}[\mathbf{x_{disguised}}](t)$.

*2) Choice of injective mappings:* To achieve the best hiding without incurring too much computational overhead, we note that the secret injective mapping $\mathbf{sm}$ should satisfy the a number of properties summarized as follows.

1) No ambiguity. This property requires that the mapping, $\mathbf{sm}$, should be one-on-one. That is, given the disguised result, there is only one corresponding original result.
2) Differentiable. The mapping, which intends to replace the initial state variable, needs to be differentiable because the solution of DAEs involves the numerical integration of a state variable's derivative. Otherwise, singularity may occur, leading to numerical instability in solving the disguised DAE.
3) Random. If the mapping is deterministic or predictable, the adversary can guess the original result by merely looking at the disguised trajectories, and, thus, there is no effect of hiding at all.
4) Efficient to evaluate. The outsourcing scheme should not incur much overhead by evaluating the extra mappings.

Inspired by [1], we find that adding a well-chosen cubic spline functions to the original state variables in the DAEs is a suitable choice of secret injective mapping. That is: $\mathbf{x_{disguise}}(t) = \mathbf{x}(t) + \mathbf{g}(t)$, where $\mathbf{g}(t)$ are randomly generated spline functions. A cubic spline function [9], is defined as a piecewise cubic polynomial, which joins in the knots obeying continuity conditions. Knots refer to as points where the two neighboring pieces connect. A cubic spline function stipulates that neighboring parts are continuous at their sharing knot and have the same second and first order derivative. To obtain a random spline function, we first set a number of randomly generated knots and then use efficient cubic spline interpolation to connect these knots.

A well-chosen spline functions can satisfy all properties listed above.

First of all, adding a spline function to the original state variable is a one-on-one mapping, so one can always recover the original result by removing the effect of spline functions from the disguised result without ambiguity. Second, each piece is a cubic polynomial. Neighboring pieces must have the same second-order derivative, so, by definition, splines function are differentiable at any point in their domain. Thirdly, spline functions are based on a large number of randomly generated knots. A cubic spline function with $m$ knots involves $(m+1$ reginons$) \times (4$ parameter per region$) - (m$ knots$) \times (3$ constraints per knot$) = m+4$ parameters. As a result, in order to determine a cubic spline function with $m$ knots, one needs to guess all the $m+4$ random values (64-bit double type). If $m$ is large enough, this will become extremely difficult. Fourthly, spline functions are highly configurable. We can configure them such that they have approximately the same range as the original trajectories to ensure the best disguising effect. Lastly, a spline function comprises piecewise polynomials, which are efficient to evaluate. In our design, by limiting the degree of the spline function to three, we can incur only a small computation overhead.

*3) Implementation of the disguising technique using spline functions:* To disguise a state variable, it requires replacing all references to the state variable and all references of state variable's derivative, with their respective injective mapping, *i.e.* adding the corresponding random spline function in the source code of the simulator program. Here, we exemplify the source code modification with MATLAB based PST.

In the source code of PST, state variable $mac\_spd$ is a $n$-by-$t$ matrix denoting the speeds of all the $n$ machines in the power system over $t$ time instant. To disguise $mac\_spd$, reference like $mac\_spd(i, j)$, which indicates the $n$-th machine's speed at the $j$th instant, and $mac\_spd(i, :)$, which refers to the $i$th machine's speed over the simulated time, are to be replaced by $mac\_spd(i, j) + disg\_mac\_spd(i, j)$, and $mac\_spd(i, :) + disg\_mac\_spd(i, :)$, respectively. In addition, reference of the state variable's derivative like $dmac\_spd(i, j)$ should be replaced with $dmac\_spd(i, j) + disg\_dmac\_spd(i, j)$. Here, $disg\_mac\_spd$ denotes the spline functions used to

disguise $mac\_spd$, where the prefix $disg$ stands for disguising, and $disg\_dmac\_spd$ is the derivative of $disg\_mac\_spd$. In practice, $disg\_mac\_spd$ is implemented as a function $disg\_mac\_spd(k,t)$ that takes three parameters: $k$ the machine's number, and $t$ the time instant. The function outputs the value of the $k$-th spline function at time instant $t$. Similarly, $disg\_dmac\_spd(k,t)$ outputs derivative of the $k$-th function in $disg\_mac\_spd$ at time instant $t$.

More specifically, for example, we replace $mac\_spd(i,:)$ with $mac\_spd(i,:) + disg\_mac\_spd(i,:)$ by three steps: 1) extracting the name of the variable, "$mac\_spd$" and its associated indices, "$(i,:)$". 2) adding the disguising prefix to the variable name to get the name of the disguising term $disg\_mac\_spd$, 3) supplying the same indices to the disguising term $disg\_mac\_spd(i,:)$, and 4) replacing $mac\_spd(i,:)$ with $mac\_spd(i,:) + disg\_mac\_spd(i,:)$. The same procedures apply to replacing an arbitrary state variable and its derivative with their respective disguising terms.

It is worth noting that one problem we encounter when we try to programmatically modify the source code is that the resulted source codes after applying the replacing method may no longer be valid. For example, the initial source code looks

$mac\_spd(:,j) = mac\_spd(:,k) + h\_sol * dmac\_spd(:,k)$

After disguising $mac\_spd$, it becomes:

$mac\_spd(:,j) + disg\_mac\_spd(:,j) = mac\_spd(:,k) + disg\_mac\_spd(:,k) + h\_sol * (dmac\_spd(:,k) + disg\_dmac\_spd(:,k))$

The newly generated code is no longer valid nor executable since the equal sign "=" means assigning the value of the right operand to the left operand. However, the left-hand side of the new code is no longer a variable. In this example, the problem could be solved by rewriting the new code to:

$mac\_spd(:,j) = -disg\_mac\_spd(:,j) + mac\_spd(:,k) + disg\_mac\_spd(:,j) + h\_sol * (dmac\_spd(:,k) + disg\_dmac\_spd(:,k))$

To solve the problem, when an "=" is scanned in the source code, we move all terms but the original state variable from the left-hand side to the right-hand side. We illustrate our source code modification algorithm in Algorithm. 1, where the symbol "||" means concatenating.

### C. Ensure input privacy through code obfuscation

In the previous section, we use the injective mappings to mask the output to maintain the output confidentiality. However, there is yet nothing so far to prevent the adversary from peeking into the source codes and discovering where and how the disguise techniques. In that case, the adversary is free to remove the mask and recover the initial source code. More importantly, regardless of whether the disguised technique has been applied to disguise the output, the source code also contains all the input data discussed in Section. III. To overcome this, we propose to make use of code obfuscation to enhance the security protection of the input data.

Code obfuscators are widely used in practice for the intellectual property protection. Collberg [3] gives a formalized notion of an obfuscating transformation:

**Definition** (obfuscating transformation) Let $P \rightarrow P'$ be a transformation of a source program $P$ into a target program

**Data**: The source code;
$\{\mathbf{Vars}, \mathbf{dVars}\}$ names of state variables and their derivatives to be disguised
**Result**: Modified source codes
**forall the** $S$ *in* $\{\mathbf{Vars}, \mathbf{dVars}\}$ **do**
  **forall the** *occurrences of "$S$(" in the source code* **do**
    Initialize character $current\_character \leftarrow$ the character next to "$S$(";
    Initialize integer $unmatched\_parenthesis \leftarrow 1$;
    Initialize character $start\_of\_indices \leftarrow current\_character$;
    **while** $unmatched\_parenthesis$ *is not 0* **do**
      **if** $current\_character$ *is "(*" **then**
        increment $unmatched\_parenthesis$;
      **if** $current\_character$ *is ")*" **then**
        decrement $unmatched\_parenthesis$
      move $current\_character$ to the next;
    **end**
    Initialize character $end\_of\_indices \leftarrow current\_character$ ;
    Initialize character string $indices \leftarrow$ substring between $start\_of\_indice$ and $end\_of\_indice$ (inclusive);
    `/* determine if the current reference is on the left-hand side of a equals sign ("=") */`
    $current\_character \leftarrow$ the next character after $end\_of\_indices$;
    Initialize character $operator \leftarrow current\_character$;
    `/* Initialize the string to replace the original reference */`
    Initialize character string $rep\_str \leftarrow$ an empty string;
    **if** $operator$ *equals "=*" **then**
      $rep\_str \leftarrow S||indices||$"$=-disg\_$"$||S||indices$;
      replace the reference with $rep\_str$
    **else**
      $rep\_str \leftarrow$ "("$||S||indices||$"$+disg\_$"$||S|indices||$")";
      replace original $S||indices$ with $rep\_str$;
    **end**
  **end**
**end**

**Algorithm 1:** Source code modification algorithm

$P'$. $P \rightarrow P'$ is an *obfuscating transformation*, if $P$ and $P'$ have the same observable behavior. More precisely, in order for $P \rightarrow P'$ to be a valid obfuscating transformation, the following conditions must hold.

- If $P$ fails to terminate or terminates with an error condition, then $P'$ may or may not terminate.
- Otherwise, $P'$ must terminate and produce the same output as P.

Various obfuscation techniques have been proposed and used in reality. These include layout transformation, control

transformation, ordering transformation. They have different levels of potency and can be applied independently. We refer the reader to [3] for more details on code obfuscation.

In our implementation, we use MATLAB's $pcode$ functionality as a surrogate of the general-purpose code obfuscator. The $pcode$ functionality is provided in MATLAB. It takes the MATLAB source code and transforms it into the preparsed and encoded version, so the resulting code becomes intelligible but functionally equivalent to the original code. Apparently, the $pcode$ satisfies the definition of code obfuscation. Note that $pcode$ is used here to represent the general obfuscation process. More sophisticated and secure obfuscations can be chosen depending on the level of protection. To our best knowledge, there is not yet any known active attack against the $pcode$. In addition, when the general-purpose secure obfuscation such as [4] and [10] becomes mature and ready for practical uses, we can quickly replace $pcode$ employed above in the dissertation with provable-secure obfuscation without changing the system model illustrated in Fig. 1.

## V. Evaluations

### A. Complexity analysis

We based our analysis on the following assumptions.

- The simulation to in involves $N$ state variables and $s$ time steps.
- The original simulator uses the fixed time step trapezoidal method for numerical integration, which is also commonly used by default in most commercial dynamic simulation software packages like PSS/E.
- Each spline function contains $k$ random knots.
- Evaluating a spline function at a certain point is $O(1)$.
- Disguising one state variable in the source code is $O(1)$.
- Obfuscating a program with a code obfuscator is $O(1)$.

*1) Local end overhead:* The local end performs SplineGen, ProbEnc, ResultRec. Based on our assumptions, in Big-O notations, the complexity of ProbEnc is $O(N)$ since it handle $N$ state variables, with each taking constant time; SplineGen is also of $O(Nk)$ complexity since it generates a spine function for $N$ state variables. Each requires Lagrange interpolation over $k$ knots, which is of $O(k)$. ResultRec is $O(Ns)$, as it needs to recover $N$ state variables' trajectories, and each of the trajectories is $s$ long. Overall, the local overhead is of $O(Nk+N+Ns) = O(Ns)$ complexity, as $s$ is much greater than $k$. As seen, the complexity is predictable and independent of the complexities of power system models of the power system thanks to offloading most computation burden to the cloud.

*2) Cloud end overhead:* The computational cost that is pertaining to solving DAEs dominates the computational overhead in the cloud end. With fixed-step trapezoidal method, the per step complexity to calculate all the $N$ state variables is given by $O(N\rho)$, where $\rho$ indicates the average complexity of evaluating the functions of state variables in the power system model. Accordingly, the complexity required by solving for the $N$ state variables' trajectories over $s$ time steps is $O(Ns\rho)$. With massive and complex system models, *i.e.*a large $\rho$, the computation complexity of the cloud end is enormous and unpredictable.

### B. Experiment results

We implemented our secure outsourcing scheme and tested whether the scheme would hide the simulation output data and whether we can recover the original trajectories from the disguised ones. As a cloud computing application, our implementation involves two parts: the cloud end and the client end. The cloud end is a service running in the cloud that receives and executes MATLAB codes sent from the client side and returns the result back upon completion. The client end consists of the source code modifier and code obfuscator. Source code modifier is a python program that first inserts the codes responsible for generating disguising spline functions to the initial source code and then runs Algorithm.1. The code obfuscator makes use of the MATLAB built-in function $pcode$ to obfuscate the modified source code.

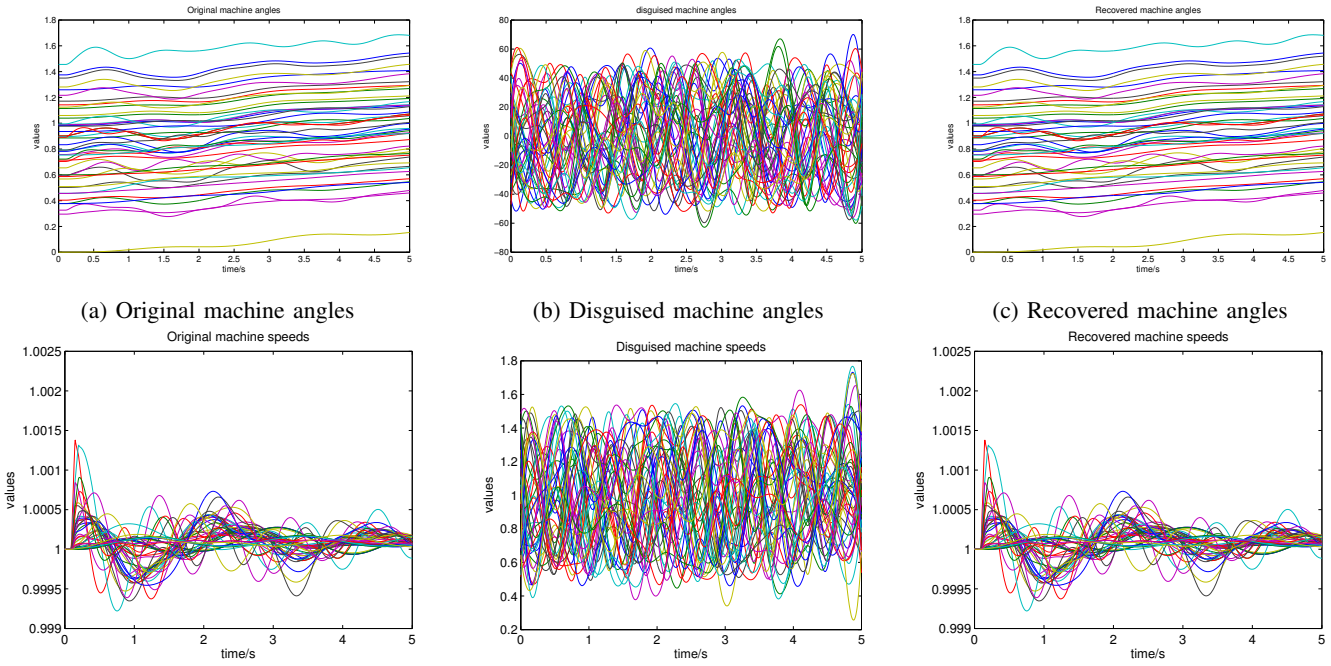
To demonstrate the correctness of our scheme, we tested it on the NPCC 48-machine, 140-bus power system model. The model represents the backbone system of the northeast region of the North American East Interconnection.

Figure. 2 delineates the original (unmasked), disguised (masked), and recovered trajectories of all the 48 generators' angles and speeds. It is shown that the original and the recovered trajectories are identical. Taking the difference of the two, which is a zero vector, verifies our observation. By that, the outsourcing scheme can disguise the original result and also recover the disguised result without any error.

## VI. Discussions

We understand that code obfuscation used in our scheme has its limitation in that given enough time and resources, a determined adversary will be able to reverse engineer any computer program protected by code obfuscation. However, we argue that, code obfuscation can still build up a strong defense against potential privacy breaches in the unique context of real-time power system simulations. In power system dynamic simulations, although the input and output information such as transient states of the power system is sensitive, its value of providing decision support for real-time power system operations diminishes quickly after the simulated period passes. A typical transient stability analysis simulates the system's behavior over a period of up to 20 seconds, as transient stability analysis does not involve dynamic models whose time constant is larger 20 seconds. Hence, data confidentiality is most valuable during this 20 seconds. In our ongoing work, we are conducting penetration testing where we simulate attacks to evaluate and measure the security of our secure outsourcing scheme. On the other hand, code obfuscation does show great usefulness when it comes to stalling the attacker. Collberg *et al.*[3] pointed out that applying obfuscating transformations can be done in polynomial time but removing them takes exponential time complexity. Wroblewski [13] conducted empirical studies which demonstrated that it takes 100 experienced crackers to work a year, 4 hours per day, to understand the meaning of a compiled and obfuscated program that contains 1,400,000 to 70,000,000 instructions protected by the opaque construct technique.

We also would like to point out that, using code obfuscation in tandem with other non-cryptographic strategies is also




(a) Original machine angles (b) Disguised machine angles (c) Recovered machine angles

(d) Original speeds (e) Disguised speeds (f) Recovered speeds

Fig. 2: Original, disguised, and recovered curves of machine angles and machine speeds

helpful to increase the strength of protection. For example, we can hide the authentic simulation among a group of randomly generated bogus but similar simulations. In order for a successful attack, the attacker will need to analyze all of the outsourced simulations, which significantly raises the difficulty to launch an attack.

## VII. Conclusions

In this paper, we presented a novel approach, which combines the disguising technique and the code obfuscation, to secure the outsourcing of non-linear power system dynamic simulations against potential privacy breaches. We implemented and tested the proposed schemes. It is shown that the trade-off between feasibility, efficiency, and security has been made.

## Acknowledgment

This work was partially supported by the NSF/DoE Engineering Research Center (ERC) under NSF Award EEC-1041877. In addition, this work was partially supported by NSF under grant CNS-1422665. The work of P. Li was supported by the U.S. National Science Foundation under grants CNS-1149786 and CNS-1343220. We would like to thank Bin Wang of Department of Electrical Engineering and Computer Science at the University of Tennessee for the helpful discussion on dynamic simulation of power system. We also thank Dr. Joe H. Chow and Felipe Wilches-Bernal of RPI for their supports on Power System Toolbox.

## References

[1] Mikhail J Atallah, K N Pantazopoulos, John R Rice, and Eugene H Spafford. Secure outsourcing of scientific computations. *Advances in Computers*, 54:215–272, 2001.

[2] Joe Chow and Graham Rogers. Power system toolbox. *Cherry Tree Scientific Software,[Online] Available: http://www. ecse. rpi. edu/pst/PST. html*, 2000.

[3] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[4] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 40–49, Oct 2013.

[5] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[6] Eugene Litvinov. Early experience with cloud computing at iso new england, 2014.

[7] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? *Proceedings of the 3rd ACM workshop on Cloud computing security workshop - CCSW '11*, page 113, 2011.

[8] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 334–348, May 2013.

[9] Christian H Reinsch. Smoothing by spline functions. *Numerische mathematik*, 10(3):177–183, 1967.

[10] A Sahai and B Waters. How to Use Indistinguishability Obfuscation: Deniable Encryption, and More. *IACR Cryptology ePrint Archive*, 2013.

[11] Cong Wang, Ning Cao, and Jin Li. Secure ranked keyword search over encrypted cloud data. *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 253–262, 2010.

[12] Cong Wang, Kui Ren, and Jia Wang. Secure and practical outsourcing of linear programming in cloud computing. In *2011 Proceedings IEEE INFOCOM*, pages 820–828. IEEE, April 2011.

[13] Gregory Wroblewski. *General Method of Program Code Obfuscation (draft)*. PhD thesis, Citeseer, 2002.

[14] Andrew C. Yao, Andrew C. Yao, Andrew C. Yao, and Andrew C. Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS '08. 23rd Annual Symposium on*, pages 160–164, Nov 1982.

[15] Jiawei Yuan and Shucheng Yu. Privacy Preserving Back-Propagation Neural Network Learning Made Practical with Cloud Computing. *IEEE Transactions on Parallel and Distributed Systems*, 99(1):1–1, January 2013.

[16] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.