Searching for High-Value Rare Events with Uncheatable Grid Computing

Wenliang Du[1] and Michael T. Goodrich[2]

[1] Department of Electrical Engineering and Computer Science, Syracuse University
wedu@ecs.syr.edu

[2] Department of Computer Science, University of California, Irvine
goodrich@acm.org

Abstract. High-value rare-event searching is arguably the most natural application of grid computing, where computational tasks are distributed to a large collection of clients (which comprise the computation *grid*) in such a way that clients are rewarded for performing tasks assigned to them. Although natural, rare-event searching presents significant challenges for a computation supervisor, who partitions and distributes the search space out to clients while contending with "lazy" clients, who don't do all their tasks, and "hoarding" clients, who don't report rare events back to the supervisor. We provide schemes, based on a technique we call *chaff injection*, for efficiently performing uncheatable grid computing in the context of searching for high-value rare events in the presence of coalitions of lazy and hoarding clients.

Keywords: Grid computing, cryptographic hash functions, obfuscation, security, cheating.

1 Introduction

Searching for high-value rare events is a natural use for grid computing environments, where computational tasks are farmed out to a collection of clients that comprise the computational *grid*. That is, such tasks naturally give rise to a large search space that can be explored in a parallel fashion to find a small handful of important inputs that deserve further attention. For example, rare-event searching forms the core of the most well-known grid computing applications, such as SETI@home, whose tasks involve processing large numbers of extraterrestrial signals for signs of intelligent life, and distributed.net, whose tasks involve performing search-space explorations that implement brute-force attacks on cryptographic algorithms.

Most grid computing environments provide an economic incentive for clients, which could involve something tangible, like digital cash, or something intangible, such as an identification of the fastest clients or the clients who have performed the most tasks. Searching for high-value rare events introduces additional economic complications in a grid computing environment, however. For example, such a system must contend with lazy clients, who can request payment for simply saying "I didn't find anything" and be assured that their answer is almost certainly correct. In addition, a system must contend with coalitions of hoarding participants, who may auction or ransom the rare events they find rather than reporting them back to the supervisor of the grid environment[1]. Here *supervisor* refers to the server that assigns the tasks to the grid participants.

The risks posed by coalitions of cheaters in grid computing environments are real and immediate. SETI@home's director, Dr. David Anderson, is quoted [8] as stating that security has been a major problem, requiring roughly fifty percent of the project's resources. For example, he mentioned that the SETI@home software had been hacked to make it look like more work had been performed (to improve leader board rankings).

Although some well-known rare-event searching tasks, such as looking for extraterrestrial intelligence or breaking cryptographic functions, are arguably of philosophical or recreational interest, some natural grid-computing searching tasks may involve identifying rare events that have significant economic value. For example, a drug company may wish to use a computational grid to identify promising drugs for treating certain diseases (e.g., by performing molecule docking simulations for a large number of candidate drug configurations). Unfortunately, coalitions of greedy, hoarding clients in a computational grid performing such a search could pose a serious obstacle for the drug company. Knowing that a drug is effective against a certain disease via grid computing, the coalition could sell the result to another drug company. This of course assumes that the participants can interpret the results.

In some computations, although the supervisor can also disguise the input so that participants cannot make direct uses of the rare events, this does not fully protect the grid supervisor from coalitions of hoarding clients. For, even if a participant cannot directly use an identified rare-event result, the coalition may be able to derive economic benefit through other means, such as ransoming the result back to the supervisor or buying stock in the supervisor's company (in the case when a rare-event discovery could boost the stock value of the company).

We provide schemes for efficiently performing uncheatable grid computing for searching for high-value rare events. Our techniques, which we call *chaff injection* methods, involve introducing elements to task inputs or outputs that provide rare-event obfuscation. Output-chaff injection applies to contexts where rare-events are not immediately identifiable by clients (as in the distributed.net application of breaking cryptographic functions). It involves the use of one-way hash functions applied to computation outcomes in a way that defends against coalitions of hoarding clients. Input-chaff injection applies to contexts where rare-events are easily identified by clients (as in the SETI@home application of finding patterns of intelligence in extraterrestrial signals). It involves the injection of a number of obfuscated inputs that will test positive as rare events. We show that distributing such inputs efficiently in an unpredictable way provides defense against coalitions of hoarding clients at a cost that is lower than existing input duplication and distribution methods.

Organization. The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 formulates the problem. Section 4 presents our general chaff injection approach for protecting rare events. Section 5 describes an input-chaff injection scheme, while Section 6 describes an output-chaff injection scheme. Finally we draw the conclusion and lay out our future work in Section 7.

[1] The participant who auctions or ransoms the rare events must be anonymous; otherwise the supervisor can get the correct results by redoing the tasks assigned to this identified participant.

2 Related Work

2.1 Uncheatable Grid Computing with Lazy Participants

Although it is an issue grid practitioners have had to contend with for some time [8], uncheatable grid computing was introduced by Golle and Sutbblebine [5]. In their paper, they advocate double-checking computational grid results by probabilistic redundant execution of the tasks (SETI@home employs a simple double-redundancy scheme [8]). A redundancy-based approach can reduce the incentive for isolated lazy cheaters when the number of potentially lazy cheaters is small, but it is less effective against large coalitions of lazy cheaters and potentially ineffective against hoarding cheaters. Indeed, we are not aware of any prior work that addresses the risks posed by hoarding cheaters.

Subsequent work on uncheatable grid computing has focused on stronger schemes to defeat lazy cheaters, that is, participants claiming credit for work left undone. Golle and Mironov propose a *ringer* scheme [4], which can protect against coalitions of lazy cheaters provided that the computational tasks all involve the Inversion of a One-Way Function (IOWF), f, for a given value y, as in the distributed.net attacks on cryptographic functions (but not the SETI@home application or other grid applications that don't involve IOWFs). In the ringer scheme, during the initialization stage for each participant, the supervisor randomly selects several inputs x_i that will be assigned to that participant and computes $f(x_i)$ for each one. Then, in addition to the value, y, that the supervisor wishes to invert, the supervisor also sends to that participant all the "ringers" the supervisor has computed for him. The participant must report the pre-images of all the ringers (as well as the pre-image of y if he was lucky enough to discover it). That is, the participant needs to compute f on x for all x in his input domain D and return the pre-image of y if found, and he also has to return all the ringer pre-images he finds. By remembering the ringers for each participant, the supervisor can easily verify whether each participant has found all his ringers or not. If he has, then the supervisor is assured with reasonable probability that the participant has indeed conducted all his computations. Golle and Mironov also discuss some ways to augment this basic ringer scheme – using bogus ringers to prevent participants from knowing the total number of ringers planted and magic sets to prevent direct images from being sent to the participants.

Szada, Lawson, and Owen [13] further extend the ringer scheme to deal with lazy cheaters for other classes of computations, including optimization and Monte Carlo simulations, by proposing effective ways to choose ringers for these computations. It is unknown whether the schemes proposed in [13] can be extended further to arbitrary computations.

Du *et al.* [3] propose a different approach to achieve uncheatable grid computing to defeat lazy participants. In their scheme, the supervisor randomly selects and verifies some samples from the task domain assigned to a participant. To prevent the participant from cheating on those samples, the participant must commit a digest of his/her entire computation results before being checked. Du *et al.* describe a commitment-based sampling scheme based on Merkle hash trees.

While the existing studies on uncheatable grid computing [3–5, 13] prevent participants from claiming credit for work they have not done, our study focuses on preventing participants from lying about the rare events they may have found, that is, defeating hoarding cheaters. The previous schemes do not seem useful against hoarding cheaters, particularly in the case of rare-event searching. Participants in a ringer scheme are not induced to report the desired inverse of the one-way function[2], and participants in the commitment-sampling scheme can simply lie about the commitment of a discovered rare event (which is unlikely to be chosen by the supervisor as a test).

As it turns out, our schemes provide some protection against lazy cheaters. In fact, even though previous works use scoring computations like SETI@home as motivating examples, our input chaff injection scheme is the first scheme for uncheatable grid computing that applies to applications that involve a scoring function that is computed in floating point and may have device-dependent rounding errors. These rounding errors don't significantly impact the correctness of requested computations but they prevent previous checking schemes, like ringers and commitment sampling, from validating otherwise acceptable computations. In the cases of other grid computing applications (with device-independent computations), protection against lazy cheaters can be strengthened by combining our schemes with one of the existing ringer or commitment-sampling techniques. Nevertheless, for the remainder of this paper, our primary emphasis will be on ways to defend against hoarding cheaters in grid computing applications.

2.2 Other Related Work

An alternative way to defeat cheating of many kinds is to use tamper-resistant software for all participants. Code obfuscators would be used in this context to convert programs to equivalent programs that are much harder to understand and reverse-engineer. Thus, it becomes hard for malicious attackers to modify the programs to accomplish what they want. However the tamper-resistant approach is only heuristically secure, and many tamper-resistant schemes cannot withstand attacks from coalitions of determined attackers [11].

The problem of uncheatable grid computing is close to another body of literature: the problem of malicious hosts in the study of mobile agents [15, 18]. Several practical solutions have been proposed for such problems, which include remote auditing [11, 14], code obfuscation with timing constraints [7], computing with encrypted functions [12], and replication and voting [17]. The major difference between the mobile-agent work and our grid-computing work is the threat model. The mobile-agent work assumes a malicious cheating model, i.e., a malicious host can do whatever it takes to cheat, including spending more CPU cycles than the honest behavior. Our study focuses on a different model, however, in which it is irrational for a participant to cheat with a cost more expensive than honest behavior.

Various cryptographic protocols, such as Private Information Retrieval (PIR) [2] and Probabilistically Checkable Proofs (PCP) [16] can also be used to achieve uncheatable grid computing. However, their expensive computation cost makes them inappropriate choice for grid computing in practice. We are interested in this paper on solutions that could be of efficient use in practice.

[2] Even if the hash value y would be randomly mixed with the ringer set for each participant, two cooperating cheaters could easily separate y from their respective ringers.

3 Problem Definition

In this section, we formally define the problem of rare-event searching with uncheatable grid computing, including definitions of the kinds of cheating we wish to defend against.

3.1 Model of Grid Computing

We consider a grid computing environment in which *untrusted participants* are taking part. The computation is organized by a *supervisor*. Formally, such computations are defined in our model by the following elements:

- **A task function $f : X \mapsto T$ defined on a finite domain X.** The goal of the computation is to evaluate f on all $x \in X$. For the purpose of distributing the computation, the supervisor partitions X into subsets. The evaluation of f on subset X_i is assigned to participant i.
- **A screening function S.** The screener is a function that takes as input a pair of the form $((x, f(x)); y)$ for $x \in X$, and returns a string $s = S((x, f(x)); y)$, where y represents the criterion. S is intended to screen for "valuable" outputs of f that are reported to the supervisor by means of the string s.

In the case of rare-event searching, we are interested in finding a small (possibly singleton) previously-unknown subset R of X such that $f(x)$ is some desired value, for each $x \in R$. Minimally, the screening function S should catch all $x \in R$.

3.2 Models of Cheaters

A participant can choose to cheat for a variety of reasons. We categorize the cheating using the following three models. We assume each participant is given a domain $D \subset X$, and his task is to compute $f(x)$ for all $x \in D$. From now on, we use D as the domain of f for a given participant.

- *Lazy Cheater Model:* In this model, the participant follows the supervisor's computations with one exception: for $x \in \check{D} \subset D$, the participant uses $\check{f}(x)$ as the result of $f(x)$. Function $\check{f}$ is usually much less expensive than function f; for instance, $\check{f}$ can be a random guess. In other words, the participant does not compute the required function f on inputs $x \in \check{D}$. The goal of the cheating participant in this model is to reduce the amount of computations, such that it can maximize its gain by "performing" more tasks during the same period of time. If the participants are getting paid, the cheating participant might be guided by the lure of money. This type of cheating behavior is a cheating on the task function f.
- *Hoarding Cheater Model:* In this model, the participant conducts all the required computations. However, the participant will keep the computation results if the results are valuable. For example, if the computation is to search for a rare event, a "lucky" participant who has found a rare event might report a negative result because of the value of such a result. This type of cheating behavior is a cheating on the screening function S.

– *Malicious Cheater Model:* In this model, the behavior of the participant can be arbitrary. For example, a malicious participant might have calculated function f on all $x \in D$, but when it computes the screener function S, instead of returning $S((x, f(x)); y)$, it might return a random value. In other words, the participant intentionally returns wrong results to the supervisor, for the purpose of disrupting the computations. A malicious cheater may be a competitor, or a non-serious participant playing pranks.

Defending against the lazy cheaters is the main purpose of the schemes proposed in [3–5, 13]. In this paper, we focus primarily on defeating the second type of cheaters – hoarding cheaters. Namely, we want to prevent the participant who has found high-value rare events (members of R) from lying about the discovery. Moreover, we even want to prevent participants from determining which are the high-value rare inputs, that is, we would like to prevent participants from learning R with any certainty.

For the sake of simplicity, we assume that participants, including the hoarding participants, honestly conduct all the tasks assigned to them (i.e., they do not cheat on f). This honesty behavior can be guaranteed by the uncheatable grid computing schemes proposed previously [3–5, 13]. In other words, by combining our scheme with the schemes such as ringers, we can successfully defeat both lazy cheaters and hoarding cheaters.

3.3 Types of Rare Events

Recall that in the screener function S, there is an certificate y, which defines the criterion to justify whether a rare event is found or not. We categorize grid computing problems into the following two types, and we will describe solutions for each of them:

1. **Obvious Rare Events (ORE):** In some grid computing scenarios, the criterion y for identifying rare events is obvious. That is, given x and $f(x)$, a participant can easily determine if x is a desired rare-event input. For example, in the Hamiltonian-cycle problem, a rare event is a permutation of the vertices that forms a Hamiltonian cycle, and verifying this fact is straightforward based on the well-known definition of Hamiltonian cycles. Indeed, every member of the complexity class NP has an effective verification algorithm, by definition. Likewise, the SETI@home application has obvious rare events – input signals that have high correlations scores for patterns of intelligence.
2. **Camouflaged Rare Events (CRE):** In some grid computing scenarios, given x and $f(x)$ alone, it is not at all clear whether x is a rare event or not. For example, to find the inverse of a one-way hash value y, i.e., to find x that generates $y = h(x)$, where h is the one-way hash function and each participant is given the value of y. Since y provided to the participants might not be the actual valuable one, even if a participant finds the corresponding x, it will not be able to decide whether this is the valuable rare event.

To prevent participants from keeping the rare events, it is necessary to hide the criterion y, so that the participants cannot identify whether an input is a rare event or not.

Hiding y for the CRE problem is relatively easier than hiding y for the ORE problem, because y is given by the supervisor in CRE problems, while y is a public knowledge in ORE problems. In the remainder of this paper, for each type of rare events, we describe our solutions for performing uncheatable grid computing in the presence of hoarding cheaters.

4 Protecting Rare Events via Chaff Injection

In this section, we give a high-level description of *chaff injection* for protecting high-value rare events in grid computing. Like a true rare event (a member of R), a piece of chaff is also a rare event, but it is generated by the supervisor. Participants cannot distinguish chaff from an actual rare event. Only the supervisor knows which one is chaff and which one is an actual rare event he/she is searching for. We use the term *semi-rare* events to refer to sets that consist of the chaff and the actual rare events.

The main idea of chaff injection is the following:

1. The supervisor injects a number of chaff items into the computations
2. Participants report to the supervisor all the semi-rare events they find
3. The supervisor filters the results and discards the chaff.

If some participants decide not to return the chaff they have found, the supervisor can easily catch such cheaters. As long as the number of chaff items is sufficiently large, participants will have no incentive to hide semi-rare events, because the results might be useless chaff and their cheating can be caught. On the other hand, chaff should not be too dense, because receiving and processing those chaff consume bandwidth and CPU cycles of the supervisor. Since the supervisor has a high bandwidth connection and a large number of available CPU cycles, introducing millions of chaff items does not introduce much burden to the supervisor, but this amount is already large enough for participants to lose their cheating incentives.

4.1 Our Approaches to Chaff Injection

Two approaches can be used to introduce new chaff, which we respectively refer to as *input chaff* injection and *output chaff* injection.

One technique involves the replacement of the criterion y with a new criterion $\hat{y}$ so many non-rare events also become a semi-rare events, while maintaining the condition for defining actual rare events. Since the criterion y is used at the output, we call this chaff approach the *output chaff injection* scheme. Formally speaking, the screen function should satisfy the following predicate:

$$S((x, f(x)); y) = 1 \longrightarrow S((x, f(x)); \hat{y}) = 1.$$

The predicate states that if an event x is a rare event, then x should also be a semi-rare event. This way, we guarantee that the set of semi-rare events is a superset of rare events.

The output chaff injection scheme is suitable for CRE problems, because the criterion y is a secret information that is known only to the supervisor. In ORE problems, where y is public knowledge, replacing y with a new criterion $\hat{y}$ cannot hide y. Therefore, we use the output chaff scheme for CRE problems. There are two different ways to transform a criterion y in this case, however, one through *expansion* and the other one through *reduction*.

For ORE problems, where y cannot be disguised, to inject chaff, we turn to the input. In this case we replace the input domain X with $X \cup C$, where C is a set of chaff items that are guaranteed to test positive for the task function f. Depending on the particular application, we may assign a participant a set of inputs that include items from both X and C, or we may probabilistically choose to send a participant a set of inputs exclusively from X or C. The key property that we must maintain for input chaff is that it must be computationally expensive for a participant to determine if his input domain includes chaff or not. The efficiency challenge for an input-chaff injection scheme is to define the size of C to be large enough for rare events to become semi-rare while also keeping the size of C to be a small fraction of the size of X.

5 Obvious Rare Events with Input Chaff

In this section, we present *input chaff injection* schemes to protect high-value rare events in grid computing for ORE problems. Because the criterion y is already publicly known in this case; obfuscating y cannot prevent participants from knowing whether a rare event has occurred. We provide solutions to three types of computations, while solutions to generic computations are still an open problem.

5.1 Graph Isomorphism

As an illustration of how input-chaff injection can obfuscate rare events for hoarding cheaters, in this section we present an uncheatable grid computation for a problem often used to illustrate the concept of zero-knowledge proofs – the graph isomorphism problem. In the graph isomorphism problem, we are given two graphs $G = (V, E)$ and $H = (V, F)$, each having n vertices and m edges, and we are asked if there is a permutation map π, on V, so that $(a, b) \in E$ if and only if $(\pi(a), \pi(b)) \in F$. We view π as a permutation map because G and H have the same vertex sets; their lists of edges are different (but possibly isomorphic). The graph isomorphism problem is not known to be in P, nor is it known if the graph isomorphism problem is NP-complete. It is widely believed to be a difficult problem, however.

Suppose, then, that we are given two such graphs G and H (and that this appears to be a hard instance of the graph isomorphism problem). Suppose further that we are interested in performing a brute-force search for an isomorphism between G and H using a computational grid. Of course, the straightforward way of solving this problem is to give each participant the following tasks:

– Each participant is given, as input, G and H and a family Π of permutation maps on V. For each $\pi \in \Pi$, the participant should check if π maps each edge in G to an edge in H and that each edge in H has an edge mapped to it by π. If π is determined to define an isomorphism from G to H, this fact is to be reported back to the supervisor.

Thus, an exponential-sized search space is divided among the set of participants in a way that requires only polynomial space and bandwidth. The problem is that an isomorphism mapping from G to H is obvious; hence, it is easily hoarded.

To obfuscate this obvious rare event, we introduce some input chaff (and a further obfuscation of non-chaff inputs). Let $\mathcal{P} = \{\Pi_1, \Pi_2, \ldots, \Pi_k\}$ be the collection of permutation sets that are to be distributed to the participants, where we assume that the sets are represented in a way that allows composition with a permutation π to define a legitimate set of permutations in a way that does not reveal π. In addition, let $\{\pi_1, \pi_2, \ldots, \pi_{k+r}\}$ be a set of random permutations and let j be an index into this set (initially, $j = 1$). For a probability p set by the supervisor (e.g., $p = 1/100$ or $p = 1/1000$), the supervisor performs the following computation for each participant:

1. The supervisor generates a random bit with probability p. If this bit is 0, then the supervisor assigns the given participant, G and H, and the next set Π_i of permutation maps to as above, except that the supervisor disguises H and Π_i by applying π_j to both (this will not change the computation for the participant). The supervisor then increments i and j.
2. Otherwise, if the random bit is 1, then the supervisor randomly chooses a permutation π from Π_i and constructs a permutation G' of G using π. Then the supervisor sends the participant, G and G' and the family Π_i, after applying π_j to G' and Π_i. The supervisor then increments j (but not i).

Note that roughly p of the inputs are now guaranteed to test positive as isomorphisms between the two graphs given to the participants. Moreover, each participant is given the graph G and a graph that looks different for each participant but which will prove to be isomorphic to G with probability p.

Security. The security of the above chaff-injection scheme is derived from the inability of participants from determining whether they have been given chaff input or not. In this case, the argument is simple: for a participant to distinguish the chaff from a true input, they must be able to tell the difference between G' and H. But doing so would require them to solve essentially the same graph isomorphism problem the grid is collectively trying to solve. Note in addition that colluding cheaters may be able to determine that some of them were assigned the same permutation set Π_i, but, again, without solving the graph isomorphism problem itself, they will not be able to tell which of these tasks are chaff and which (if any) are a true input set. Each participant is given a different looking graph to test against G (even the ones assigned H); hence, colluding participants cannot determine which of them are given chaff and which are not (without solving the graph isomorphism problem itself, that is).

Analysis. The expected additional work involved in using this input chaff injection scheme is equal to p times the work needed by the original solution, which is not resistant to hoarding cheaters. That is, if we let e denote the additional work introduced by input chaff injection, then

$$E(e) = k \cdot p,$$

where k is the number of subproblems defined by the supervisor based on the size of her participant base (e.g., $k = 100,000$ or $k = 1,000,000$ are realistic values). Thus, if $p = 1/10$ or $p = 1/100$, this scheme introduces a small computational overhead of 10% or 1% to the process. For example, if $p = 1/100$ and $k = 1,000,000$, then $E(e) = 10,000$. But even with small values for p, rare events are significantly obfuscated. For, even if G and H are isomorphic, we have the following:

$$\mathrm{Pr}(\text{a participant finding an isomorphism that is not chaff}) = 1/e.$$

For example, if $e = 10,000$, then there is only a one in ten thousand chance that a discovered isomorphism is not chaff; hence, a 99.99% chance that a hoarding participant will be caught in this scheme (using these example parameters). Note that our chaff injection approach is therefore more efficient than task duplication approaches to uncheatable grid computing [5], which have a computational overhead of 46% to 243% for reasonable protection against lazy cheaters (while providing only modest protection against hoarding cheaters of rare events).

5.2 Data Filtering Problems

Having given a "warm-up" example of input-chaff injection, we turn in this section to a more practical set of grid computations – *data filtering* problems. In data filtering problems we are given a large set X of data instances and a Boolean filtering function f. The supervisor is interested in all the elements x of X such that $f(x) = 1$. Usually, the function f will involve some internal scoring function on each input x along with a threshold value such that if x scores above this value, then x is considered rare and interesting. This class of problems includes the SETI@home application, where X consists of extraterrestrial signals that are scored against what are considered to be patterns of intelligence. Likewise, this class of problems includes the drug screening example mentioned in the introduction.

By their very nature, it is not obvious which of the inputs in X will score positive for the filter f (for otherwise there would be no motivation for us to go to the trouble of using a grid computing environment to solve this problem). For example, a casual examination of the signals that have scored highest so far in the SETI@home scoring function does not yield any obvious patterns; to the naked eye they all appear as noise. Thus, for data filtering applications such as this, employing an input chaff injection scheme is easy.

To inject input chaff into the set of tasks, the supervisor needs only to have a set of instances Y such that determining if any member y_i is not in X is at least as difficult as computing $f(y_i)$. (The supervisor may not need to explicitly construct Y if she has a way of choosing elements from Y probabilistically.) Then the supervisor can randomly inject members of Y into the task sets $D \subset X$ for each participant (with some probability p) to provably obfuscate the rare events. For example, a true input x in the SETI@home application could be transformed into chaff simply by adding a pattern of intelligence to it.

Note that this chaff injection scheme has a finer granularity than our scheme for graph isomorphism, that is, in this case, each participant is likely to have some semi-rare chaff events to report, whereas in the graph isomorphism solution each participant was given an input that was entirely valid or entirely chaff. In any case, the security and analysis of this scheme for data filtering are similar to those given for the graph isomorphism solution given above.

5.3 Searching for Solutions to NP-Complete Problems

In this section, we present a chaff-injection solution for grid computations that involve searches for solutions to NP-complete problems. We begin with a template for building such schemes and we then give a solution for a well-known NP-complete problem – 3SAT. Since all NP problems can be reduced to 3SAT in polynomial time, this gives a chaff-injection scheme for any NP-complete problem. Of course, in practice, it may be more efficient for a supervisor setting up a grid search for solutions to a specific NP-complete problem to use our template and example as a guideline in setting up a specific scheme.

A Template for Chaff Injection for NP Solution Searching. By their nature, solutions to difficult instances of NP-complete problems are rare events, and, with the practical importance of many of these problems, such solutions may have economic value. Let us therefore provide a template for chaff injection for NP solution searching. Every language L in NP has a polynomial-time verification algorithm, A, that takes as input a problem instance x and a polynomial-sized certificate y, such that $x \in L$ if and only if A accepts x for some y (e.g., see [6]). Thus, we can set up a naive grid computation by dividing the set Y of all certificates into as many subsets as participants and asking each participant i to evaluate $A(x, y)$ for all y in his $Y_i \subset Y$. Of course, accepting inputs for A are obvious in this naive computation.

To set up a grid computation that defends against hoarding cheaters, we assume the existence of two functions, α and β, for L. The function α takes an input instance x and certificate set Y_i and produces a pair (x', Y_i') such that $A(x, y) = A(x', y')$, for each $y \in Y_i$ and $y' \in Y_i'$, but determining this fact is computationally difficult (e.g., at least as hard as the graph isomorphism problem). The function β takes a certificate set Y_i and returns a problem instance z for L such that $A(z, y) = 1$ from some $y \in Y_i$, but determining this fact is computationally hard (ideally, there should be only one accepting certificate in Y_i and locating it should be difficult). Moreover, it should be computationally difficult to determine whether or not z is isomorphic to x with respect to A. The template for a chaff-injection scheme, then, is for the supervisor to set a probability p (e.g., $p = 1/10$ or $p = 1/100$) and perform as follows for each participant:

1. The supervisor generates a random bit b with probability p. If $b = 0$, then she applies $\alpha(x, Y_i)$ to construct an isomorphic pair (x', Y_i') that is equivalent with respect to A to (x, Y_i). She then sends (x', Y_i') to the participant and increments i.
2. Otherwise, if $b = 1$, then the supervisor applies β on Y_i to generate at hard positive instance z of L on Y_i. She then applies α to the pair (z, Y_i) to yield an isomorphic pair (z', Y_i'), and she sends (z', Y_i') to the participant and does not increment i.

The participant in this scheme has no efficient way of determining if they got an input isomorphic to x or z; hence, each participant is induced to perform their computation. But roughly p of the inputs are now guaranteed to be rare events; hence, each rare event should be reported or a cheater will be discovered with high probability. The analysis and security of this scheme are therefore similar to that given for the graph isomorphism problem above.

A Specific Example for 3SAT. Since any NP-complete problem can be reduced to another NP-complete problem, we only need to describe our solution to one NP-complete problem. We choose the 3-CNF satisfiability problem, because it is traditionally the problem to which other problems in NP are reduced.

A *CNF formula* is a Boolean expression in a conjunctive normal form, i.e., each conjunct in the formula is a *clause*, which is itself the disjunction of literals. Each *literal* is either a *variable*, or the complement of a variable. A *3-CNF* formula consists of clauses each containing exactly 3 literals. A formula is *satisfiable* if there exists at least one assignment for those variables that will satisfy the formula. We use n and m to represent the number of variables and the number of clauses, respectively.

Finding whether a 3-CNF formula is satisfiable is NP-complete (e.g., see [6]). To apply our grid computing template, therefore, we need to show how to construct the functions α and β for this problem. Fortunately, building α for a formula x and certificate set Y_i is relatively simple in this case: we simply rename all the variable names in x (and change their respective names in Y_i in a corresponding way), and randomly permute the clauses in x. Determining if the output formula x' is equivalent to x is at least as hard as the graph isomorphism problem. For β, we can employ any of a number of schemes for constructing hard instances of 3SAT. For example, Achlioptas, Kautz, *et al.* [1, 9] have an interesting tunable scheme for generating hard instances of SAT, and Massacci and Marraro [10] show how to construct a satisfiable Boolean formula such that finding the satisfying assignment is as hard as breaking the DES encryption algorithm. Combining these techniques allows us to create difficult instances of 3SAT that would match the statistical properties of the input x, allowing us to create hard, satisfiable formulas that are difficult to distinguish as not being isomorphisms of x.

The statistical properties of two isomorphic 3-CNF formula are similar (e.g., the number of clauses are the same, and the number of occurrences for each variables are the same), while the statistical properties of chaff might be different from the original input and other chaff. Therefore, two colluding participants can compare their formulae to find out whether they are likely to have the chaff or the original input. There are two approaches to defeat the colluding attacks. First, we can make the statistical properties of the chaff match that of input x; second, the chaff can also be isomorphic to each other, so they also have similar statistical properties. Therefore, by comparing the statistical properties of their inputs, colluding participants are still unable to tell whether they have the original input x or chaff.

6 Camouflaged Rare Events with Output Chaff

In this section, we present our *output chaff injection* schemes to protect high-value rare events in grid computing when rare events are camouflaged.

6.1 An Chaff-Injection Scheme via Criterion Expansion

In the following, we give a generic solution for CRE problems, i.e. the criterion y is only known to the supervisor. We assume that the screen function is defined as the following:

$$S(f(x), y) = 1 \text{ iff } f(x) == y.$$

For CRE problems, to obfuscate rare events, we introduce many other criteria. If an input x satisfies any of these criteria, it is considered as a semi-rare event. If the actual rare-event criterion is superficially indistinguishable from the introduced criteria, participants cannot tell whether a semi-rare event is the actual rare event or just chaff. The algorithms for CRE problems is described in the following:

1. The supervisor randomly selects m inputs $x_1, \ldots, x_m$ from the input domain X. Note that X is the global input domain, each participant only conducts tasks for a subset of X.
2. The supervisor generates m chaff by computing $c_i = hash(f(x_i))$, for $i = 1, \ldots, m$.
3. The supervisor sends the list $C = \{hash(y), c_1, \ldots, c_m\}$ to all the participants. C should be permuted to hide $hash(y)$.
4. For any input x assigned to each participant, the participant computes $hash(f(x))$ and compares the results with the list C. If a match occurs, the participant sends x back to the supervisor; otherwise x is discarded.
5. The supervisor can verify whether a returned x value is an actual rare event or chaff, by a simple lookup in C (say, by storing the elements of C in a hash table). The supervisor also checks whether the participant whose tasks include chaff has returned the chaff or not. This way, the cheater can be caught.

Security Analysis. When a participant finds a semi-rare event, because of the existence of chaff, the probability that this semi-rare event is an actual rare event is only $\frac{1}{m}$. Furthermore, because the participants all get the same list, i.e., the information about the rare events is the same, colluding does not bring any extra information to improve that probability. Incidentally, this scheme differs from the ringers scheme of Golle and Mironov [4] in two important ways. First, our output chaff injection scheme gives the same set of chaff to all participants, so collusion is useless in identifying chaff, unlike ringers. Second, unlike ringers, our output chaff cannot be distinguished from actual desired outputs (without performing the requested computation, that is).

In an actual grid computing application, of course, the rare event may or may not exist. Knowing whether a rare event exists or not might also be valuable, and needs to be protected. Our chaff scheme does prevent the participants from finding out this information. The only situation when the participants know that the rare event exists is when they have found the matches for all the elements in C. Because the chaff are selected uniformly randomly from the entire input domain, the chance that a group of cheating participants get all the chaff is p^m, where p is the portion of the cheating participants among all the participants. In practice, p is quite small. Therefore, p^m is negligible when m is large.

Computation Overhead. The extra computation cost added to this grid computing is that of computing a cryptographic hash function and the finding of matches among the list C (the lookup in C can be done using a non-cryptographic hash table). We have assumed that the cost of computing a hash function is negligible compared to the cost of computing the task function f. Moreover, finding the match can be quite efficient using the hash table data structure even if the list is long (i.e., the value of m is large). Therefore, the supervisor can afford to send a list containing tens of thousands chaff. As we have known, the longer the list is, the fewer incentives participants have.

6.2 Criterion Reduction for CRE Problems

Another way to increase the number of inputs that satisfy a criterion, and thereby introduce input chaff, is to reduce the amount of requirements that must be satisfied. For example, assume that the original criterion says that an input must satisfy five requirements to be considered as a rare event; we can reduce the number of requirements to three in the new criterion, thus increasing the number of satisfiable inputs.

1. The supervisor computes $h(y)$, and let $\hat{y}$ be the first k bits of the result, where k is a security parameter. The supervisor sends $\hat{y}$ to participants along with the task assignments.
2. For each assigned input x, a participant computes $f(x)$, and checks whether the first k bits of $h(f(x))$ equal $\hat{y}$. If true, the participant returns x and $h(f(x))$ to the supervisor; otherwise, discards x.
3. The supervisor verifies whether $h(f(x)) = h(y)$. If false, x is just chaff; else, x is a rare event.

Analysis. Just like the criterion-expansion scheme, we want to keep the number of semi-rare events sufficiently large, so participants will lose incentives to cheat. On the other hand, the number of semi-rare events must be kept within a range, otherwise, finding the actual rare events from the semi-rare events will consume too much time of the supervisor.

Because of the randomness of hash value, we can easily decide how many bits to disclose to the participants. Assume that the search space is 2^{64}, and hash values of outputs are uniformly distributed. Disclosing the first 44 bits of $h(y)$ will lead to a real-to-false ratio 1:2^{20}. This is big enough for the participant to lose incentives, but small enough for the supervisor to find out the actual rare events (2^{20} means only 1 million of hash-function evaluations).

Compared to the criterion-expansion scheme, the criterion-reduction scheme introduces computation costs on the server side, although they are manageable. Another difference is that in criterion reduction, controlling the ratio between the rare and semi-rare events becomes difficult when the hash values of outputs are not uniformly distributed (this can happen, for example, when many outputs have the same values).

7 Conclusion and Future Work

This paper provides a first step towards disguising high-value rare events in the context of grid computing with hoarding cheaters. The techniques we employed involved the injection of input and/or output chaff, depending on whether the rare events were obvious or not. There are still many interesting open problems that could be studied in the future, including:

- Can we develop a generic scheme for all ORE problems?
- The difficulty with the ORE problems is the publically known indicator of rare events. Can we turn an obvious indicator to a non-obvious indicator, i.e., transform ORE problems into CRE problems? If this can be done, we could use solutions to CRE to solve ORE problems.

Acknowledgment

The authors acknowledge the supports from the United States National Science Foundation under grants ISS-0219560, IIS-0312366, CCR-0225642, CCR-0311720, CCR-0312760.

References

1. Dimitris Achlioptas, Carla P. Gomes, Henry A. Kautz, and Bart Selman. Generating satisfiable problem instances. In *AAAI/IAAI*, pages 256–261, 2000.
2. C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. *Lecture Notes in Computer Science*, 1592:402–414, 1999.
3. W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *The 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 4–11, Tokyo, Japan, March 23–26 2004.
4. P. Golle and I. Mironov. Uncheatable distributed computations. *Lecture Notes in Computer Science*, 2020:425–440, 2001.
5. P. Golle and S. Stubblebine. Secure distributed computing in a commercial environment. In P. Syverson, editor, *Proceedings of Financial Crypto 2001*, volume 2339 of *Lecture Notes in Computer Science*, pages 289–304. Springer-Verlag, 2001.
6. M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, New York, NY, 2002.
7. F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. *Mobile Agents and Security, Lecture Notes in Computer Science,Springer-Verlag*, 1419:92–113, 1998.
8. L. Kahney. Cheaters bow to peer pressure. *Wired Magazine*, Feb. 15, 2001.
9. Henry A. Kautz, Yongshao Ruan, Dimitris Achlioptas, Carla P. Gomes, Bart Selman, and Mark E. Stickel. Balance and filtering in structured satisfiable problems. In *IJCAI*, pages 351–358, 2001.
10. F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem: The encoding of the Data Encryption Standard. *Journal of Automated Reasoning*, 24(1–2):165–203, 2000.
11. F. Monrose, P. Wykoff, and Aviel D. Rubin. Distributed execution with remote audit. In *Proceedings of ISOC Symposium on Network and Distributed System Security*, pages 103–113, February 1999.
12. T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts, springer-verlag. *Lecture Notes in Computer Science*, 1419:44–60, 1998.
13. D. Szajda, B. Lawson, and J. Owen. Hardening functions for large scale distributed computations. *IEEE Symposium on Security and Privacy*, 2003.
14. G. Vigna. Protecting mobile agents through tracing. In *Proceedings of the 3rd Workshop on Mobile Object Systems*, June 1997.
15. G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer, 1998.

16. R. Ostrovsky W. Aiello, S. Bhatt and S. Rajagopalan. Fast verification of any remote procedure call: short witness-indistinguishable one-round proofs for np. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 463–474, July 2000.
17. F. Schneider Y. Minsky, R. van Renesse and S. D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of Seventh ACM SIGOPS European Workshop,System Support for Worldwide Applications*, pages 109–114, Connemara, Ireland, September 1996.
18. B. S. Yee. A sanctuary for mobile agents. In *Secure Internet Programming*, pages 261–273, 1999.