Breaking the Barriers: High Performance Security for High Performance Computing

Kay Connelly
Indiana University
150 S. Woodlawn Ave.
Bloomington, IN 47405
(812) 855-0739
connelly@indiana.edu

Andrew A. Chien
University of California, San Diego
9500 Gilman Drive, Dept. 0114
La Jolla, CA 92093-0114
(858) 822-2458
achien@cs.ucsd.edu

ABSTRACT

This paper attempts to reconcile the high performance community's requirement of high performance with the need for security, and reconcile some accepted security approaches with the performance constraints of high-performance networks. We propose a new paradigm and challenge existing practice. The new paradigm is that not all domains need long-term forward data confidentiality. In particular, we take a fresh look at security for the high-performance domain, focusing particularly on component-based applications. We discuss the security and performance requirements of this domain in order to elucidate both the constraints and opportunities. We challenge the existing practice of high-performance networks sending communication in plaintext. We propose a security mechanism and provide metrics for analyzing both the security and performance costs.

General Terms

Distributed Computing, High Performance, Security.

1. INTRODUCTION

Over the past decade, high performance networks of workstations have replaced supercomputers for scientific parallel computations. As these clusters have become easier to manage and use, distributed applications outside of parallel scientific codes have targeted this platform as well. Search engines, airline reservation systems and command-and-control systems are just a few such applications. The combination of low-cost and high-performance execution has made such systems desirable to a wide variety of industries. In particular, developments in user-level communication layers have enabled applications to access the raw performance of such networks. Applications achieve peak bandwidths over 1Gbps, and latencies on the order of 10 to 20 microseconds.

With the focus centered on performance, there has been little research into security for high performance systems. Much of the security-related work in the high performance computing (HPC) community addresses how to securely communicate to high-performance applications from the wide-area (i.e.: how to retrieve remote data sets or how to securely start a remote high-performance application) [1, 6].

New Security Paradigms Workshop '02, September 23-26, 2002, Virginia Beach, Virginia.

But, beyond simple logins and access rights associated with those logins, there are few security mechanisms being regularly employed within the high-performance clusters, themselves. In terms of the communication going over the high performance network, the standard practice is to have *no* security. All data is sent in plaintext. The main goal is to keep the communication overhead to a minimum. Grafting an existing encryption mechanism onto the communication path is not seriously considered due to the relatively high overheads. A typical symmetric key encryption algorithm incurs an overhead of *milli*seconds, which is two to three orders of magnitude greater than the network latency in high-performance networks. Using such a mechanism would take the "high performance" out of HPC. Now that industry and the military are seriously pursuing high performance clusters as an environment to run their distributed applications, the HPC community must revisit the issue of security.

Distributed components are quickly becoming the programming model of choice for distributed high performance applications [16]. In this type of model, the functionality of the application is encapsulated in multiple components and spread over the network. In order for the application to make any forward progress, components must interact with other components via remote procedure calls (RPC). Thus, the state of the execution of the application can be pieced together with these RPCs.

One noticeable effect of low-latency communication is that the ideal balance of computation and communication changes dramatically from traditional TCP/IP over Ethernet. On the slower networks, a component must compute a lot and communicate rarely in order to achieve its peak performance. If it doesn't have enough computation to keep it busy while waiting on the results from an RPC, then it becomes idle waiting on the network. In the high performance domain, communication is many orders of magnitude faster. Thus, components can have much less computation, and still not block on the network. This results in applications that have many fine-grained components (as opposed to fewer, larger components). Finer-grained applications have more RPCs, making a more detailed state-reconstruction possible.

There are two important security attack scenarios for high performance component applications. The first is for an attacker to send RPC messages to various components in order to change the execution of the application. The attacker may not need to reconstruct the current state of the application in order for such an attack to succeed. Since RPCs are currently sent in plaintext with no authentication mechanisms, this attack is feasible as long as the locations of the components are accessible. The second scenario consists of an attacker eavesdropping on the communication and determining when the application is in a vulnerable state. The attacker can then attack the application, another application, or use the information to gain an advantage in the real world .

The contributions of this paper include:

- A discussion of specific security and performance needs of high performance applications.
- An approach for protecting tightly-coupled, high-performance, component communication.
- Definition of security and complexity metrics to analyze this approach.
- A characterization of the security achieved by this approach. For a modest sized component, this approach provides a brute-force search space of 10^{28}. A known-plaintext attack requires at least 20 plaintext/ciphertext pairs.
- A proof-of-concept prototype that adds less than 10% to the message latency.

Section 2 describes the shift in the way we must think about security for high-performance systems. Section 3 gives one possible approach to satisfying the security and performance requirements of this domain and introduces the metrics we use to analyze the approach. We apply these metrics to three particular security techniques in Section 4. Section 5 describes an initial prototype with performance numbers which demonstrate that this approach is promising in terms of performance. Finally, we describe some related work in Section 6, and conclude in Section 7.

2. PARADIGM SHIFT

When looking at security for specialized domains such as high-performance component applications, we cannot naively apply existing security solutions without potentially sacrificing the benefits of that domain. Instead, we must evaluate the needs of the system. There are security needs, but there are other needs, such as performance, reliability and usability.

In the case of HPC, the driving force is performance. Existing security mechanisms simply incur too much overhead for them to be adopted by the HPC community. Thus, we have an additional restriction on security mechanisms, in that they must have a low overhead. "Low overhead", of course, is a fuzzy term. For now, let it be sufficient that the overhead incurred by the security mechanism must be the same order of magnitude as the latency of the message sent in plain text. In the case of 100 Mb switched Ethernet sending small messages, this means that the security mechanism may incur an overhead up to 100 microseconds in order to satisfy the performance constraint.

Now, let us turn to the security needs of high-performance component applications. The bulk of the communication in this type of application is temporary data or information related to the control flow of the application. For example, in the case of scientific, parallel applications, the data traversing the network might be intermediate values in a computation. In the case of a command-and-control application, the data may consist of sensor-values or simple Booleans to enable and disable various resources. The risk of the communication being exposed is not that the data is valuable, but that the data may indicate that the application is in a weakened state, making it vulnerable to a specific attack. It does not matter if an attacker is able to determine the current state of the application in a few hours, minutes or seconds, as the application will have moved on to another state. This is an important change in perspective: long-term forward security is not the ultimate goal. The goal is to protect the data long enough for the application to change state, and to do so with low overhead.

While any sensitive data which needs long-term forward security must use a traditional encryption mechanism, the bulk of the communication in our target applications consists of these intermediate, or short-term, values; and thus, they have a shorter cover time than traditional data. This gives us a new opportunity when designing a security mechanism. In order for an application programmer to be able to determine if the cover time is long enough for their particular application, it will be necessary to precisely quantify the cover time provided by any proposed mechanism. In the most naïve attack, the cover time is roughly proportional to the size of the brute-force search space. In a more sophisticated attack on the state of the security module, we must determine the frequency with which the module must be reconfigured with a new secret key. This frequency must be low enough that the overhead of transmitting the keys does not dominate.

A key question for a given application is: how long does the cover time need to be? The security requirements depend on the frequency of the state changes. For a loosely-coupled application, communication (and thus state changes) are infrequent, necessitating a longer cover time. For tightly-coupled applications, communication and state changes are frequent, requiring a much smaller cover time. In essence, there is a range of communication patterns and security requirements.

Previous work [7] provides a high-performance communication library which allows the application programmer to turn communication security (DES) on and off. We believe, however, that more than two modes are needed in order to get the HPC community to actually *use* the provided security. For example, triple DES, which incurs one-way overheads on the order of a few milliseconds for 4k messages, could be used for loosely coupled applications. Single DES, which incurs overheads on the order of 500 microseconds could be used for applications which are somewhere in the middle of the tightly-to-loosely coupled continuum. And finally, instead of the plaintext option, another mechanism should be available which incurs virtually no overhead and provides very short term security for tightly coupled applications.

There already exist security mechanisms for the medium to loosly-coupled applications. The rest of this paper explores

one possible approach for a security mechanism specifically designed for tightly-coupled components.

3. APPROACH

Existing encryption mechanisms apply operations such as substitution and transposition in an iterative fashion on the data. For every iteration, data is read from a buffer, transformed in some way, and copied to another buffer. Analysis of messaging layers shows that buffer copies are one of the major sources of overhead to avoid [3, 9]. Indeed, zero-copy messaging layers have become the accepted norm in the HPC community.

Thus, when developing a security mechanism for tightly-coupled, high-performance applications, it is necessary to avoid buffer copies whenever possible; making an "iterative" approach undesirable.

Instead, our approach applies traditional security techniques such as transposition, substitution and data padding while the message is being marshaled onto the wire. We apply these operations on the primitive data types (i.e, bytes and words) in the RPC marshalling layer. This allows us to avoid all buffer copies, and to capitalize on the marshaling infrastructure that already exists, adding what we anticipate to be a modest amount of overhead.

In addition, much of the computation in the techniques we propose can be done before the message becomes available from the application. This allows our system to pre-compute the more time-consuming algorithms during any CPU idle time, significantly reducing the communication latency experienced by the application.

3.1 Metrics

In order to determine the success of this approach, we must analyze the level of security as well as the implementation complexity for any possible algorithms that combine transpositions, substitutions and data padding.

We define two metrics for security:

1. **S** is the size of the brute-force search space. Given **S**, an application developer may determine if it is sufficiently large enough for their particular application.
2. **M** is the number of plaintext/ciphertext pairs necessary in order to determine the internal state of the security module. The security module sends a new key before **M** messages is sent. Of course, key transmission overhead must not dominate. **M** must be large enough that a sufficient amount of communication may occur before a new key must be securely transmitted.

In addition, we define one metric for implementation complexity:

1. **C** is the complexity of the algorithm, normalized to some base operations: basic compute, memory load and store operations, as well as a basic random number generation operation. The symbols used to represent each of these operations in equations will be: **op, ld, st** and **rand**, respectively.

We expect all possible algorithms to have a tradeoff between security and complexity. The more secure, the more complex; and thus, the slower the performance. The key is to provide a precise analytical model of security and complexity so that an application developer may determine if the approach is suitable for their application and deployment environment.

4. DESIGN

In this section, we describe how three basic security techniques (substitution, transposition and data padding) may be applied in the RPC marshalling layer. These operations are used to make all of the RPC messages look the same in terms of their structure, so that any particular RPC message could be invoking any of the methods on the destination server.

4.1 Substitution

Substitution replaces each character in the plaintext with a different character in the ciphertext. Conceptually, substitution is implemented with substitution tables, which enables the individuals with access to the tables to encode/decode messages one character at a time. The substitution table is the "secret" which must be kept from adversaries. Historically, static substitution tables are used to determine the mapping, which means that the table does not change for some period of time. A major drawback of static substitution tables is that if an adversary obtains the plaintext and ciphertext of a message, he can easily reconstruct the table, making it possible to immediately decode all future messages. Another drawback is that they are susceptible to frequency analysis attacks, where the frequency of characters in the plaintext and ciphertext can be used to determine the substitution table.

Table 1: Security and performance metrics for sample algorithms, assuming reasonable use of registers to reduce memory load/store operations [4]. n is the number of items in the message, k is the number of bits used to hold the method offset, b is the number of random bits per random number used in SHUFFLE algorithm and p is the number of padding arguments

	S	M	C
Method offset substitution	2^k	2^k+1	Send: 4 **op** + 2 **ld** + 4 **st** + 1 **rand** Recv: 7 **op**+ 3 **ld** + 7 **st** + 1 **rand**
SHUFFLE	**n!**	$\mathbf{bn}\log_{n!}2$	**11n op** + **2n ld** + **2n st** + **n rand**
Padding	**(n+p)!/p!**	$\mathbf{b(n+p)}\log_{(n+p)!}2$	**11p op** + **2p ld** + **2p st** + **2p rand**

Since we anticipate an adversary being able to eventually decode RPC messages, it is inadvisable to use static substitution tables, as an adversary would be able to reconstruct the table over time. Instead, we use dynamic substitution tables [12]. Figure 1 shows how the table entries are altered every time they are used. Dynamic substitution tables not only prevent table reconstruction, but they also avoid frequency analysis attacks.

Dynamic substitution requires output from a pseudo random number generator (PRNG) every time the table is used. Depending on the performance of the PRNG, this could make applying substitutions to every piece of data in the message quite expensive. One piece of data that must be substituted, however, is the method identifier. For this discussion, let us assume that the method identifier is an offset into an array, as is the case in Java's RMI (Remote Method Invocation) layer. If the method identifier is not substituted, but simply placed into a different location in the message using transposition, then it will be fairly trivial for an attacker to determine the remote method being invoked[1]. Thus, while we may want to examine applying substitutions on all data in the message, it is absolutely necessary to apply a substitution on the method identifier.

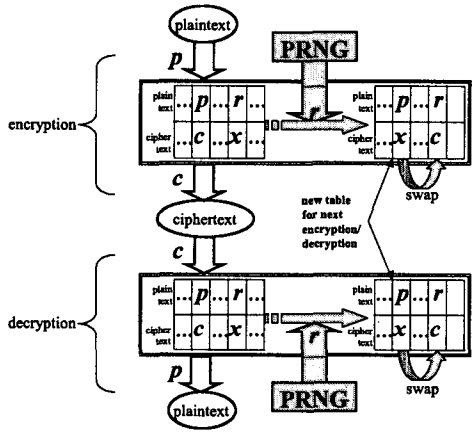

Figure 1: Dynamic Substitution

An additional benefit to substituting the method identifier is that probes of the network can be detected. Specifically, the range of numbers to which the method offsets are mapped should be significantly larger than the actual number of methods in the interface. Thus, if an adversary attempts to probe the network with some random values just to see what happens, it is likely that the probe message will contain a method offset value which does not map to an actual method. For the sample component described later in this section, almost 84% of all possible method offsets do not point to a real method. Once a probe is detected, the security system may notify an intrusion detection system and take evasive actions.

Table 1 gives the values of the security and performance metrics. The search space, **S**, equals $2^{\mathbf{k}}$, where **k** is the number of bits used to encode the method offset. Assuming the PRNG is good (i.e. it does not get into a short cycle), the minimum number of messages to determine the random numbers used, **M**, is $2^{\mathbf{k}}+1$. The implementation complexity, **C**, equals 4 **op** + 2 **ld** + 4 **st** + 1 **rand** on the send side, and 7 **op** + 3 **ld** + 7 **st** + 1 **rand** on the receive side. As an optimization, the random number may be generated in advance.

Table 2 gives the values of the metrics for a sample component which has 41 methods but uses 8 bits to encode the method offset using dynamic substitution. This results in **S** = 256 and **M** = 257. While the search space is not large for this particular technique, the number of messages before an intruder may predict the internal state is more than sufficient. For example, our implementation's secure key-exchange takes on order of 108 milliseconds, but provides enough bits of entropy to reseed the PRNG 16 times. Thus, a key must be exchanged every 4096 messages, resulting in a 15% overhead if the component is communication bound.

4.2 Transposition

Transposition does not change the values of the data being sent, but changes the order in which they appear in the message. A particular order is called a permutation.

Transposition can be applied in the RPC layer simply by changing the order in which data is marshaled onto the wire. In order to disperse complex data structures throughout the message, the order should be changed on the primitive data (i.e. bytes, or words). Once an order is decided, it costs very little to alter the marshaling calls to adhere to that order. Indeed, the most time consuming aspect of transpositions at this level is determining the desired permutation of the message.

There are a variety of algorithms in the literature which could be used to determine a permutation based on random numbers [5, 8, 10, 11, 13, 14]. It is necessary to analyze any possible algorithm in terms of the security and complexity metrics introduced in Section 3.1. To give an example of what is feasible, we briefly describe an algorithm based on the SHUFFLE algorithm [5, 8]. In the SHUFFLE algorithm, an array of data is manipulated, resulting in a permutation of the original array. In our modified algorithm, we shuffle an array of *positions* (1 through **n**, where **n** is the number of data items), and use the position permutation to drive the data marshaling order. This allows us to determine the permutation before the data is available. Thus, if we incorporate data padding as described in the next subsection to make all of the messages the same length, the permutation algorithm may be computed in advance during CPU idle time, reducing the message latency experienced by the application.

[1] There are two reasons that permutation-only of the method identifier results in a trivial attack. First, many of the data values won't fit into the range of method identifiers, allowing an intruder to immediately eliminate them as possible method identifiers. Second, the number of data values that could possibly represent method identifiers will be dramatically less than all possible method identifiers, substantially reducing the search space (and thus search time).

Table 2: . This table lists the values of the security and performance metrics for a sample component which has 41 methods, with n = 20 , k = 8, b = 64 and p = 10 .

	S	M	C
Method offset substitution	256	257	Send: 4 **op** + 2 **ld** + 4 **st** + 1 **rand** Recv: 7 **op**+ 3 **ld** + 7 **st** + 1 **rand**
SHUFFLE	2.43×10^{18}	21	220 **op** + 40 **ld** + 40 **st** + 20 **rand**
Padding	7.30×10^{25}	18	110 **op** + 20 **ld** + 20 **st** + 20 **rand**

```
//param n: number of items to permute

int [] SHUFFLE(int n){
   float u;
   int k, current, tmp;

   int *items = malloc(n * sizeof(int));

   //initialize array of positions
   for(k=0; k < n; k++)
      items[k] = k;

   for(current=n-1; current > 1; current--){
      //generate random number between 0 & 1
      u = random(0,1);

      // make into int between 1 & current
      k = floor(current*u) + 1;

      // swap items[current] and items[k]
      tmp = items[k];
      items[k] = items[current];
      items[current] = tmp;
   }
   return items;
}
```

Figure 2: modified SHUFFLE algorithm

As the pseudocode shows in Figure 2, our modified SHUFFLE algorithm starts with an array the size of the number of items to be permuted, with each entry in the array initialized to its index in the array. Then, we set the current position to be at the end of the array. We randomly choose an index in the array between the beginning and the current position. Swap the value at the randomly chosen index with the value in the current position, then decrement the current position. Repeat until the current position is at the beginning of the array. Now, the value at index **x** in the array is the position in the message for the data normally sent in position **x**.

Table 1 shows the equations for the security and complexity metrics of our modified SHUFFLE algorithm. There are **n!** possible permutations of the message, where **n** is the number of items to be permuted in the message. Table 2 shows that for our sample component with the number of bits per random number, **b** = 64 and the number of data items, **n** = 20, **S** = 2.43×10^{18} (or approximately 2^{62}). On average, an adversary would have to be able to analyze 2^{61} states to find the actual state. If an attacker had a cluster of 1 GHz machines available to her and if each machine could analyze a state in 20 cycles, she would require over 45 billion nodes to decode the message in 1 second, or approximately 12.5 million nodes to determine the message in 1 hour.

To compute **M**, we determine how many sequences of random numbers could have resulted in a particular permutation. Then we can determine how many messages are needed to eliminate all but one sequence. When **b** random bits are used in each iteration of the loop, **M** is equal to $\mathbf{bn}\log_{\mathbf{n!}}2$. In Table 2, we see that **M** is 21 messages for the sample component. Using the key exchange and message latencies that we used in Subsection 4.1, this would result in a key exchange every 320 messages with an overhead of 69%. While this may appear large at first, we believe the overhead can be reduced by performing parts of the key exchange in the background before the key is needed.

Finally, we compute the complexity of the algorithm, assuming that the compiler can make judicial use of registers, avoiding memory load/store operations for temporary data like temporary variables and loop iterators. The complexity then becomes **11n op + 2n ld + 2n st + n rand**.

4.3 Data Padding

Data padding consists of adding data to a message. It is often used to ensure messages are a particular length, making the implementation of certain algorithms on the message easier. In addition, data padding has been used to avoid traffic analysis attacks, which are able to infer important information simply by knowing how much data is being sent [15].

For RPC communication, data padding can be used to avoid traffic analysis attacks which can determine the identity of the remote method simply by analyzing the length of the message and the values of the arguments being passed to the method. Adding padding data makes all of the messages look identical in terms of their length and the type of data that is being sent. Thus, to an eavesdropper, any message may be used to invoke any of the methods on the server.

Data padding can be applied to the message in the marshaling layer. For each piece of padding data, two things must be decided: its location in the message and its value. There are multiple approaches to determine both. To identify the most suitable, the security and performance metrics must be applied. Here, we have space to describe one possible approach.

To determine the location of the padding data, we first append it to the original message. It then undergoes transposition along with the real data as described in the previous section. To determine the value, we could simply send a random number from the PRNG. More intelligent value choices can be made depending on the types of the real data in the message, making the values of the data statistically meaningless to an eavesdropper. For example, if a Boolean is sent, it would be wise to send the negation of that Boolean value as well. Then, an eavesdropper cannot easily determine the value of the Boolean.

Table 1 provides equations for our metrics based on sending random numbers as our padding data. It does not make sense to insert padding data without also permuting the message data, so the security metrics, **S** and **M**, contain equations which include the transposition algorithm. Thus, **S** is **(n+p)!/p!**,

where **p** is the number of pieces of padding data in the message. Similarly, **M** is $\mathbf{b(n+p)}\log_{(n+p)!}2$. The equation for **M** assumes that the values of the padding data are checked on the receiver side to ensure that they are accurate. This check aids in detecting probes of the network in a way similar to the one described in Subsection 4.1. The complexity of this algorithm is the complexity of the transposition algorithm (replacing **n** with **p**) plus one random number for each piece of padding, for determining the value of the pad: **11p op + 2p ld + 2p st + 2p rand**. Of course, the complexity metric does not include the time it takes to send the padding data (**p** times the data rate), but that should be considered as a cost.

As an optimization, all of the overheads associated with the permutation algorithm may be pre-computed. The data padding values, however, can only be determined once the identity of the remote method being invoked is known, as the number and type of pads is dependant on the remote method.

5. INITIAL EVALUATION

We have implemented a proof-of-concept prototype to demonstrate the feasibility of this approach. The prototype performs a diffie-hellman key exchange at connection setup time. Based on the key, it performs a simple method offset substitution, message permutation and data padding algorithm in the RPC layer. The purpose of the prototype is to prove the concept of implementing this scheme in the RPC layer, and therefore, it does not explore the many possible algorithms that could be used to apply the operations to a given method invocation.

Our prototype builds upon a high-performance Java implementation called Manta [17]. Our executing environment is an 8-node network of 1.5 GHz Pentium 4 boxes with 256 MB of RAM, running RedHat Linux 7.1. The cluster is connected with 100 Mb switched Ethernet.

Adding simple substitutions and transpositions to the RPC layer incurs an overhead of less than 10% of the original message latency, showing that this is a promising approach in terms of the necessary performance. For example, for an RPC with 64 data bytes, our mechanism adds 12 microseconds to the base plain-text latency of 152 microseconds.

In addition, since many of the algorithms that we are exploring require one or more random numbers, we have measured the time it takes to retrieve a random number from an implementation of Yarrow. One call to Yarrow took approximately 0.241 microseconds to retrieve 64 random bits. Thus, for our sample component used in Table 2, the time it would take to obtain all of the necessary random bits is less than 10 microseconds, which is substantially less than the maximum overhead requirement of 100 microseconds. It may be possible to further reduce that overhead by eliminating function calls and retrieving all of the random bits in one call.

We believe that further optimizations can be performed to effectively reduce the latency experienced by the application. Because much of the work in the algorithms we have presented does not depend on the actual data being sent, several key pieces may be computed in advance during CPU idle time. Possibilities include:

- random number generation
- computation of the message permutation
- diffie-hellman key exchange

6. RELATED WORK

Globus [7] is one of the only high-performance projects which has added an encryption option (DES) to their standard communication library. Because of the cost, it is not clear if this capability has been adopted by their users for communication within a high-performance cluster.

The Globus [7] and Legion [6] architectures contain mechanisms for users to specify the level of security required by each communication channel. While neither project provides a low-overhead/short-cover-time mechanism as we have described in this paper, both could include such a mechanism in the future. With the variety of communication patterns in high-performance environments, we believe the flexibility provided by these two architectures is essential for satisfying the necessary security and performance requirements.

A seminal work on Lightweight Remote Procedure Calls optimizes RPCs between processes on the same machine [2]. This concept could be incorporated in any high-performance messaging layer, disabling all security/encryption of the data when being sent to another processes on the same machine. The focus of our paper, however, is on RPCs that traverse the network.

7. CONCLUSIONS

Because of the performance requirements in high-performance distributed systems, it is not possible to simply retrofit existing security mechanisms and expect the HPC community to use them. This paper is a first attempt to construct a security solution based on the specific needs of high-performance component communication. We classified the data security needs and determined that much of the data transmitted over the network has a short cover-time requirement. We then presented an approach which capitalizes on the marshaling infrastructure in order to maintain a low overhead. We specified metrics for evaluating the approach, and analyzed three security techniques with these metrics. Finally, we described an initial prototype and its performance which indicates that this approach is promising for meeting the performance requirements of the high-performance domain.

8. ACKNOWLEDGMENTS

We would like to thank to Geetanjali Sampemane for providing feedback on an early draft of this paper, and the anonymous reviewers for their helpful comments.

This work is partially supported by the Pervasive Technology Labs at Indiana University, and supported in part by the Defense Advanced Research Projects Administration through United States Air Force Rome Laboratory Contracts AFRL F30602-99-1-0534 and the National Science Foundation thru NSF EIA-99-75020 Grads.

9. REFERENCES

[1] Allcock, Bill et. al. "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing". *IEEE Mass Storage Conference*, 2001.

[2] Bershad, Brian et. al. "Lightweight Remote Procedure Call". *ACM Transactions on Computer Systems*, Vol. 8, issue 1, pages 37-55, February 1990.

[3] Clar, David D., Van Jacobson, John Romkey, and Howard Salwen. "An analysis of TCP processing overhead". *IEEE Communications Magazine*, June 1989.

[4] Connelly, K. and A. Chien. "Elusive Interface Design and Analysis", Computer Science Department, Indiana University. April, 2002.

[5] Durstenfeld, Richard. "Algorithm 235: Random Permutation [G6]". *Communications of the ACM*. Vol. 7, page 420, 1964.

[6] Ferrari, Adam et. al. "A Flexible Security System for Metacomputing Environments". *High Performance Computing and Networking Europe*, April 1999.

[7] Foster, Ian, Nicholas Karonis, Carl Kesselman and Steven Tuecke. "Managing Security in High-Performance Distributed Computations", *Cluster Computing*, Vol 1, issue 1, pages 95-107, 1998.

[8] Knuth, Donald. 1997. *The Art of Computer Programming*, Vol 2, *Seminumerical Algorithms*. 3rd ed. Reading, Mass: Addison-Wesley.

[9] Kay, J. and J. Pasquale. "Measurement, Analysis and Improvement of UDP/IP Throughput for the DECstation 5000", *Proceedings of the 1993 Winter Usenix Conference*, San Diego, USA, pages 249-258.

[10] Plackett, R. "Random Permutations". *Journal of the Royal Statistical Society, Series B (Methodological)*. Vol. 30, issue 3, pages 517-534, 1968.

[11] Rao, C. "Generation of Random Permutations of Given Number of Elements Using Random Sampling Numbers". *Sankhya, A*. Vol. 23, pages 305-307, 1961.

[12] Ritter, Terry. "Substitution Cipher with Pseudo-Random Shuffling: The Dynamic Substitution Combiner". *Cryptologia* Vol. 15, issue 4, pages 289-303, 1990.

[13] Sandelius, Martin. "A Simple Randomization Procedure". *Journal of the Royal Statistical Society: Series B (Methodological)*. Vol. 24, issue 2, Pages 472-481, 1962.

[14] Sloane, N. "Encrypting by Random Rotations". *Cryptography: EUROCRYPT'82*. Lecture Notes in Computer Science Vol. 149, pages 71-128, 1983.

[15] Timmerman, Brenda. "A Security Model for Dynamic Adaptive Traffic Masking". *New Security Paradigms Workshop*, 1997.

[16] Tuecke, S. et. al. "Grid Service Specification". February 2002. http://www.globus.org/research/papers/gsspec.pdf

[17] Veldema, Ronald, Rob van Nieuwpoort, Jason Maassen, Henri E. Bal and Aske Plaat. "Efficient Remote Method Invocation". *Technical Report IR-450*, Vrije Universiteit Amsterdam, September, 1998.