

# Elusive Interfaces: A Low-Cost Mechanism for Protecting Distributed Object Interfaces

Kay H. Connelly  
University of Illinois  
[hane@cs.uiuc.edu](mailto:hane@cs.uiuc.edu)

Andrew A. Chien  
University of California, San Deigo  
[achien@cs.ucsd.edu](mailto:achien@cs.ucsd.edu)

## Abstract

*We describe Elusive Interfaces, a new mechanism for protecting distributed interfaces from attacks. Elusive Interfaces can be used to ensure privacy of remote method invocation (RMI) messages and for authenticating the issuer of an invocation. Elusive Interface exploits a space of message configurations by using message transformations to conceal the actual communication. We describe this space and characterize its size for realistic distributed object interfaces. We analyze the effect of the various transformations on two types of attacks: eavesdropping and unauthorized invocations. We describe a sample application and examine the effects of applying the transformations to its interface. Specifically, the keyspace for our sample application exceeds  $2.7 \times 10^{15}$  formats that one particular message may take. We describe a prototype implementation of Elusive Interfaces, and evaluate the approach empirically. Performance experiments confirm that an Elusive Interface system can provide enhanced security at a moderate overhead (0.6 milliseconds per null RMI), and performs 11 to 56 times faster than Triple-DES, 168 bit-key encryption. With the recent advances in high-speed networks and the significant reduction of communication overheads experienced by user-level messaging layers, low-overhead security mechanisms are essential. The low-overhead incurred by our Elusive Interface prototype makes it ideal for such high-speed networking environments.*

## 1 Introduction

Until recently, distributed applications had very simple topologies. Most of the applications followed a simple client-server model, rather than more general component-based architectures. A typical system consisted of a few non-distributed, statically located servers and several clients. The communication traversing the network was sparse, with all of the heavy computation performed within the server, hidden from the rest of the world. Further, the communication channels between clients and servers were fairly short-lived. A connection would be acquired, some tasks performed,

and then the connection would be terminated. Because this style of communication is coarse-grained, an eavesdropper could not gain very much information about the internal state of either the client or the server.

With the rise of better distributed programming tools (Java, CORBA, DCOM), the style of distributed applications is changing dramatically. The communication topologies of these component-based distributed applications are more complex, with many smaller objects located on physically disparate machines cooperating to provide services that were once grouped together in a monolithic server. Since services are provided by many objects coordinating together, there are longer-lived connections between these objects in order to amortize connection setup overhead. These objects use exported interfaces that are documented by Interface Definition Languages (IDLs), making the flow control of the application which used to be internal, publicly available to attackers. With this finer granularity of network traffic, an outsider has a much greater insight into the state of the application. Indeed, many of these component-based applications are commodity-off-the-shelf (COTS), allowing anyone to purchase the application and search for security weaknesses in the privacy of their own home.

In addition, high-speed networks are becoming a reality, with bandwidth exceeding gigabits per second. Simultaneously, there have been breakthroughs in user-level messaging layers[1], reducing the per message communication overheads to tens of microseconds. Security overheads that appear moderate over TCP will be the dominant factor in these environments, making the reduction of these overheads critical. While one approach to reducing this overhead will be hardware deployment, it will be difficult for such solutions to become so pervasive that inter-operability is not a major problem. An end-to-end software solution is necessary for heterogeneous systems.

## **1.1 Approach: Elusive Interfaces**

With the new style of dynamic, component-based, distributed applications, more information is transmitted on the network, giving attackers more opportunities for devising a successful attack.

Further, the increase of network traffic and inter-object dependencies makes the use of a high-cost cryptographic algorithm prohibitive for performance reasons. Finally, applications are becoming more adaptive, to both the static hardware and system configuration as well as to run-time conditions. This necessitates an exploration of new, low-cost security methods that can protect this new style of application and adapt along with it.

Making high-performance, distributed applications survivable involves:

- **protection of information flow:** Remote interfaces are documented by IDLs, allowing attackers to glean information about the internal state of application by eavesdropping.
- **protection of data:** Attackers can use the real-time data being passed within remote method invocations to determine information about the current state of an object, or the state of some other part of the system.
- **access control of interfaces:** Attackers can exploit weaknesses in an interface (buffer overflow, etc), if they are allowed to invoke the interfaces.
- **low cost solution:** The use of component-based applications greatly increase the network traffic. With the advent of low-overhead, user-level networking on high-speed networks (100Mb to 10Gb+), high-overhead security is not acceptable.

Our solution, *Elusive Interfaces*, addresses each of these issues. In *Elusive Interfaces*, the way in which a method is invoked over the network is constantly changing. Method IDs are mapped to other IDs, the order of parameters are scrambled, and fake parameters and method invocations are introduced as noise into network traffic. The way in which the interface is mutated is dependent on a secret key, which can be transmitted during the connection setup, or some other secret-sharing method[2]. Similar to cryptographic techniques, the *Elusive Interfaces* implementation is open to the public, while the key dictates the actual interface. Indeed, *Elusive Interfaces* is essentially another cryptographic technique, with different performance characteristics and a different size search space.

While traditional encryption mechanisms address some of the above needs, their cost is prohibitive in high-speed networking environments. In addition, *Elusive Interfaces* has other

advantages that are more in line with the dynamic application model we described earlier. In particular, Elusive Interfaces is:

- **adaptive:** In response to real-time system state, Elusive Interfaces can increase security/decrease performance during an attack and increase performance/decrease security under normal operation.
- **perceptive:** Elusive Interfaces can use its internal state to detect an attacker probing an interface. Once detecting a false invocation, it can notify an intrusion detection system.
- **flexible:** Elusive Interfaces is flexible in its view and management of cryptography. Instead of changing key-lengths to provide more or less protection (which can be difficult to do without a major software engineering effort), an Elusive Interface system can change the size of the keyspace by tuning a few parameters.

Elusive Interfaces is intended to be part of a dynamically adaptive security system, which will use Elusive Interfaces as just one of several mechanisms (i.e. Elusive Locations, booby-traps, encryption, etc...) in order to enhance survivability. If an intruder is detected in the system, an application may wish to pay the overhead of an encryption algorithm in order to keep the entire remote method invocation secret. Otherwise, under normal operating conditions, the cheaper Elusive Interface system may be used.

The primary contributions of this paper are threefold. First, we outline a novel solution for application security based on component-interfaces. We exploit a space of message transformations to conceal the actual communication. In addition, we exploit the structure of the messages to deliver these transformations at a low cost. Secondly, we provide an analytical characterization of the difficulty of attacking the Elusive Interface search space. We show that the number of mutated interfaces grows exponentially with the number of methods in the original interface, resulting in higher security for more complex interfaces. Finally, we describe a prototype implementation and compare it to a non-encrypted channel and a Triple-DES, 168 bit-key, encrypted channel. The asymptotic performance of our prototype is within 3% of the plain-text channel, and over 56 times

faster than the encrypted channel. We believe this substantiates the claim that Elusive Interfaces provides component-based security with minimal overheads, making it suitable for the emerging high-speed networks.

The remainder of the paper is organized as follows: Section 2 depicts a detailed example application that would benefit from Elusive Interfaces. Section 3 outlines several interface mutations and explores the keyspace for an Elusive Interface solution; while Section 4 applies this analysis to a real-world database server's interface. We present an Elusive Interface prototype and its performance in Section 5. Finally, we describe related work and conclude in Sections 6 and 7.

## 2 Motivating Example

An Aegis Cruiser distributed missile control application can demonstrate the necessity of each of the requirements listed in the previous section. Such an application may have a *drill* mode, which is used during a drill to test the ability of the military personnel to respond to an attack. During this drill mode, commands to launch missiles would be ignored, since a real attack is not underway. Being in the drill mode is part of the application's state.

An attacker could determine when the application is in drill mode simply by eavesdropping on the application's remote method invocations. Then the attacker could launch a real military attack, leaving the Aegis cruiser helpless to respond until the drill mode is over-ridden. Obviously, using the control flow of the application in this way gives the opposing side an advantage in battle.

Even if the missile control application is constructed carefully to not transmit information about being in drill-mode, another application could expose the vulnerability within data it transmits. As an example, imagine that a notification program, such as Unix's *wall*<sup>1</sup>, is used to inform certain personnel of an impending drill. This data can be used to determine the time of the drill; and, as in the previous example, an attacker can physically attack the ship while its offensive powers are disabled.

---

<sup>1</sup> *wall* is a simple notification program that writes a message to the console of all users logged on to a particular computer.

As an example of an attack that involves access control, suppose the object responsible for launching missiles exports a method which will crash the object if invoked incorrectly<sup>2</sup>. If this object is running on a computer with the Windows operating system, an attacker could utilize the fact that the win32 subsystem uses a first-available algorithm to assign process ids in order to replace the object with one of his/her own. The attack would consist of the following steps:

- 1) gain access to the computer (any non-privileged access will do),
- 2) obtain the process id for the missile launching object
- 3) crash the missile launching object by exploiting the vulnerability in the exported method
- 4) start as many processes as necessary until it one has the appropriate process id.

If the object's identity within the distributed application is a function of the process id, then by following the above steps, an intruder can replace the missile launching object with one of its own, and the distributed application will not detect the swap! Note, this attack is unique to the win32 subsystem since it re-assigns process ids using the first available. Other operating systems, such as Unix, assign process ids in order up to  $2^{32}$  before rolling over and starting at the beginning, making this attack infeasible. As is apparent, successful attacks involve combining information about an application and the system on which it runs. Distributed applications are particularly susceptible to these types of attacks since the more complicated the system, the more opportunities for error arise.

### **3 Elusive Interface Framework**

Elusive Interfaces addresses each of the problems encountered by the example application in Section 2. In particular, Elusive Interfaces exploits the similarity of method invocations to make individual method invocations look virtually identical. Since an attacker cannot determine the actual method invocations that are being transmitted, they can tell very little about the internal state of the distributed application by analyzing communication between objects. In addition, if fake remote

---

<sup>2</sup> While this may seem like a contrived example, it is, in fact, a reality. In September of 1997, an Aegis cruiser was left dead-in-the water for over 2 1/2 hours after a user inputted a zero, resulting in a divide by zero error.

method invocations are transmitted, as well as fake data sent within the real remote method invocation messages, then it will be difficult for an attacker to determine the legitimate data in real-time. If the format of a remote method invocation message is determined at run-time by a session key known only to the two communication objects, then an attacker, who does not know the run-time session key, will be unable to sensibly invoke the interfaces. Finally, Elusive Interfaces does not perform bit-wise operations on the data stream like standard encryption mechanisms. Instead, it exploits the structure of the data by simply altering the order in which the data is (un)marshalled. While exploiting this structure allows for easier decrypting of the data, we believe (and will show) that the keyspace is still large-enough to prevent decoding *in real-time*.

### **3.1 Dimensions of Mutation**

There are several dimensions that can be permuted in order to create a mutated interface. This section discusses some of those possible dimensions. We classify them in terms of mutations that preserve the size of the remote method invocation, and those that do not. By mutating among multiple dimensions, the system makes decrypting an interface by examining snooped messages difficult. In addition, the interface that is being used will constantly be changing. Section 3.2 assigns metrics for evaluating the strength of the system based on the dimensions that are mutated and the frequency of the mutations. Finally, the sequence in which the mutated interfaces are used is another dimension that makes it difficult for an outsider to determine the current interface.

#### ***3.1.1 Size preserving***

- **Method offset value.** To identify a method, a system typically has a one-to-one mapping between methods and indices in an array<sup>3</sup>. One mutation is to change that mapping.

---

<sup>3</sup> Java RMI uses method offsets. The example program in section 4 is CORBA based, and uses strings of method names. It is simple to map these strings to integers, and perform the same transformations that are used for method offsets.

- **Method offset range.** In addition to changing the mapping, the values of the offsets do not need to be zero-based and contiguous, making it more difficult for an attacker to guess at an offset.
- **Method offset location.** The method offset is located in a static location in the remote method invocation message that traverses the network. Another technique for mutating an interface is to change the location of the method offset.
- **Parameter location and organization.** Similar to the method offset location approach, another way to mutate interfaces is to change the location of all of the parameters in the method invocation. For complex data structure parameters, the way in which the parameters are un/marshaled could be varied, making it more difficult to extract the parameters from snooped messages. Indeed, portions of multiple complex data structures could be interleaved.
- **Parameter Encryption.** Individual parameters may be encrypted, using different keys if desired. While this would add to the overhead, it would make it more difficult for a snooped message to be decoded and the interface pieced together.

### ***3.1.2 Non-size preserving***

- **Parameter buffering.** Fake parameters can be included in method invocations to hide the real ones. Likewise, real parameters may be embedded inside fake ones. The goal is to make all method invocations look the same with respect to the number and type of parameters.

### ***3.1.3 Time***

An important aspect to Elusive Interfaces is that the interface which is being used may change over time. There are several possibilities to coordinate interfaces changes. The in depth analysis of these possibilities will be covered in a future paper. For now, we can think of changing interfaces in terms of stream ciphers. With stream ciphers, the key which encrypts/decrypts the data changes

dynamically in response to the actual data stream. Similarly, the interface which is being used in an Elusive Interface connection may change in response to the actual method invocations.

### 3.2 Keyspace

This section quantifies the size of the keyspace<sup>4</sup> for an application interface which is mutated among each of the various dimensions discussed in the Section 3.1. Without loss of generality, we assume that all data types are reduced to a set of primitive data types, such as integers, characters and booleans. For example, a complex number,  $\mathbf{c}$ , can be described as two integers,  $\mathbf{a}$  and  $\mathbf{b}$ . So, if  $\mathbf{c}$  is an argument to a function, the analysis treats it as two completely separate pieces of data:  $\mathbf{a}$  and  $\mathbf{b}$ . This simplification will automatically enable the parameter organization to be altered as discussed in Section 3.1.1.

There are three types of data that make up an Elusive Interface function call. The method id ( $\mathbf{m}_i$  for  $1 \leq i \leq r$ , where  $r$  is the number of methods in the interface), the parameters ( $\mathbf{p}_{ij}$  for  $0 \leq j \leq n$ , where  $n$  is the number of parameters for method  $\mathbf{m}_i$ ) and the dummy parameters ( $\mathbf{d}_{ik}$  for  $0 \leq k \leq q$ , where  $q$  is the number of dummy parameters for method  $\mathbf{m}_i$ ).

For this section, we will use the following interface,  $\mathbf{M}$ , as a simple example:  $\mathbf{M} = \{add(x,y), sub(x,y), eq(x,y)\}$ . Without any interface mutations  $r = 3$ ,  $n = 2$  and  $q = 0$ , for our example interface<sup>5</sup>.

On the server object, there is an array of methods, as seen in Figure 1a. When another object wants to  $add(3,4)$  on the remote server object using Java RMI, what actually gets sent over the wire is the offset of the add method in the array, followed by the two arguments to the function: (1,3,4). So, a method invocation can be represented by an ordered tuple, where the arity of the tuple is one more than the number of parameters.

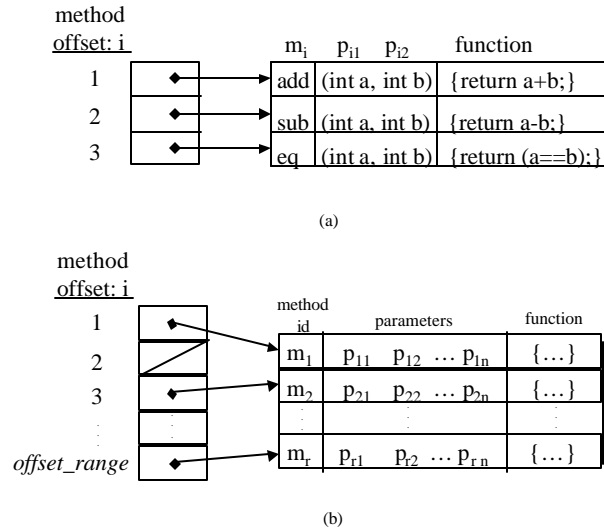
Figure 1b gives the graphical representation of the server-side data structures for a more generic interface, with the method offset mapping and method offset range mutations in place.

---

<sup>4</sup> The keyspace is "the number of distinct key-selected transformations supported by a particular cipher"[3].

<sup>5</sup> Every method in this simple interface has the same number of arguments. This is generally not the case with real applications, making the analysis more complex.

Specifically, there are *offset\_range* entries in the offset array, but only *r* actual methods. Therefore, some of the entries in the offset array are null.



**Figure 1:** On the server object, there is an array of pointers to methods. A remote invocation message has the offset of the desired method. (a) shows the server for the example interface, ***M***. (b) shows a more generalized server with *r* methods in the interface, *n* parameters for each method and *offset\_range* entries in the method array.

Our focus is on two types of attacks: active attacks, where the attacker invokes a remote method, and eavesdropping, where the attacker learns import information just by monitoring the network traffic. This section analyses the space for both types of attacks. As with traditional encryption, the larger the keyspace, the longer it will take an attacker to crack the secret key, and thus decode the communication, with brute-force.

To prevent the active attack, it is desirable to maximize the number of possible formats that a method invocation may take. For example, if we map the offset anywhere from 1 to 4, then the intruder has four different messages which it may have to send in order to actually invoke *sub(x,y)*: (1,x,y), (2,x,y), (3,x,y) and (4,x,y). We will call the number of formats that a message may take, ***NumFormats(m)***. For this simple example, ***NumFormats(m<sub>2</sub>) = 4***.

To prevent a passive attack, we must maximize the number of method invocations that a particular message may represent. Again, if the only mutations we allow is that the message offset may be mapped anywhere from 1 to 4, then a particular message, say (2,x,y) can map to *add(x,y)*,

$sub(x,y)$  or  $eq(x,y)$ . We will call the number of methods that a particular message may map to **NumMethods( $m_i$ )**. For this simple example, **NumMethods( $m_i$ )** = 3. Note that **NumFormats( $m_i$ )** is not necessarily the same as **NumMethods( $m_i$ )**. This is because that while a method invocation can be mapped to any of the offsets in the offset-range, we are not adding actual methods to the interface, so a particular message's offset can only map to one of the actual methods<sup>6</sup>.

Finally, since the interface mutations will be changing over time, the number of possible interfaces plays a role in the degree of security. If there are twice as many possible interfaces, then an attacker must spend approximately twice as long eavesdropping before it can determine the pattern that is being used to cycle through the interfaces. We will call the number of unique interface sets **NumSets( $M$ )**.

Case: Mutation	NumFormats( $m_i$ )	NumMethods( $m_i$ )	NumSets( $M$ )
0: No Elusive Interfaces	1	1	1
1: Method Offset Value	$r$	$r$	$P(r, r) = r!$
2: Method Offset Range	$offset\_range$	$r$	$P(offset\_range, r) = \frac{offset\_range!}{(offset\_range - r)!}$
3: Tuple Organization	$P(n+1, n+1)$ $= (n+1)!$	$P(n+1, n+1)$ $= (n+1)!$	$(n+1)P(n, n)^r$ $= (n+1)(n!)^r$
4: Method Offset Range and Tuple Organization (case #2 and case #3)	$offset\_rangeP(n+1, n+1) =$ $offset\_range(n+1)!$	$rP(n+1, n)$ $= r(n+1)!$	$(n+1)P(offset\_range, r)P(n, n)^r =$ $(n+1)\frac{offset\_range!}{(offset\_range - r)!}(n!)^r$
5: Method Offset Range, Tuple Organization and Parameter Buffering (buffers not checked)	$offset\_rangeP(n+q+1, n+1) =$ $offset\_range\frac{(n+q+1)!}{q!}$	$rP(n+q+1, n)$ $= r\frac{(n+q+1)!}{(q+1)!}$	$(n+1)P(offset\_range, r)P(n+q, n)^r =$ $(n+1)\frac{offset\_range!}{(offset\_range - r)!}\left(\frac{(n+q)!}{q!}\right)^r$
6: Method Offset Range, Tuple Organization and Parameter Buffering (buffers checked)	$offset\_rangeP(n+q+1, n+q+1)$ $= offset\_range(n+q+1)!$	$rP(n+q+1, n+q)$ $= r(n+q+1)!$	$(n+1)P(offset\_range, r)P(n+q, n+q)^r$ $= (n+1)\frac{offset\_range!}{(offset\_range - r)!}((n+q)!)^r$

**Figure 2:** Equations for the keyspace of an Elusive Interface system.

<sup>6</sup> Of course, if the Elusive Interface system sends completely fake messages, then those fake messages do not map to real methods, increasing the search space.

The table in Figure 2 gives the equations for a variety of interface mutations. Each column is a particular equation, while each row has the equations for a particular interface mutation (or combination of mutations).

- **Case 0** represents the standard RMI interface without any mutations, resulting in all equations equal to 1.
- **Case 1** represents the scenario where the method offset mapping is changed, and the range of the method offset remains unchanged:  $r$ . Thus, every method invocation can take  $r$  forms (the method offset can be any one of  $r$  numbers), and every message can map to  $r$  different methods. The number of unique interfaces corresponds to the number of permutations of the numbers 1 through  $r$ :  $r!$ .
- **Case 2** not only changes the mapping of the method offset, but also increases the size of the offset range to  $offset\_range$ . Thus, the number of formats increases to  $offset\_range$ . However, the number of methods that a particular message may be remains  $r$ , since there are still only  $r$  methods in the actual interfaces. Finally, the number of unique interfaces is the number of permutations of  $r$  items, out of a total of  $offset\_range$  items.
- **Case 3** concerns only the order of the method offset and the arguments. There are  $n+1$  items being sent over the wire. So, the number of formats that a particular method make take is the number of permutations of  $n+1$  items. Similarly, the number of methods that a message may represent is the number of permutations of  $n+1$  items. The equation for **NumSets(M)** is not as intuitive. All of the methods in an interface instance must have the method offset located in the same place; otherwise, it would be very difficult for the server object to determine which method is actually being invoked. So,  $(1,x,y)$  should never be in the same interface instance as  $(x,2,y)$ . For every position in the tuple, there are  $n!$  permutations of the arguments *for each* of  $r$  methods. Thus, case #3 adds exponential complexity to **NumSets(M)**.
- **Case 4** combines cases 2 and 3: the method offset mappings change with a range of  $offset\_range$  and the order of the arguments and method offset changes as well. The complexity of the

mutations in case #2 and #3 grow independent of each other. Thus, to compute the complexity of this case, we simply multiply the corresponding equations of case #2 and #3.

- **Case 5** shows case #4 with parameter buffering, where the values of the dummy parameters are not checked. While  $sub(x,dummy1,y,dummy2)$  is technically a different permutation than  $sub(x,dummy2,y,dummy1)$ , an attacker does not need to distinguish between the two to effectively eavesdrop or invoke a method. We must look at the number of unique permutations of the location of the *real* data out of all  $n+q$  locations, where  $q$  is the number of dummy parameters. Thus, we take the equations for case #4 and replace all  $P(n,A)$  with  $P(n+q,A)$  and all  $P(n+1, A)$  with  $P(n+q+1, A)$ .
- **Case 6** is case #4 with parameter buffering where the values of the dummy parameters are checked by the server object. Thus, every instance of  $n$  should be replaced with  $n+q$ , where  $q$  is the number of dummy paramters.

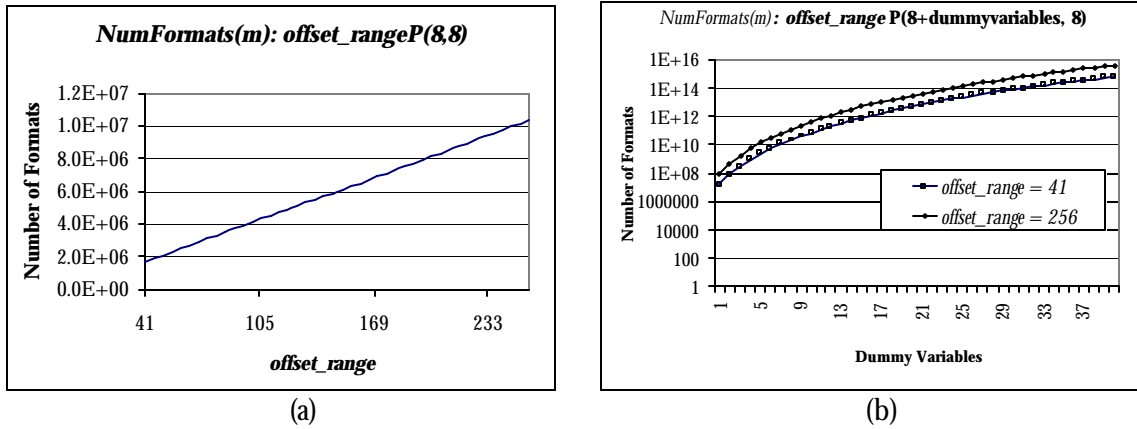
To summarize, **NumFormats(m)** and **NumSets(M)** scale with the size of *offset\_range*, while **NumMethods(m)** does not. If the primary attack you are trying to defend against is eavesdropping, inflating the method offset range artificially does not help. In addition, all of the equations scale with the number of dummy parameters ( $q$ ). We explore the relative effects of these two variables in our discussion of the sample database application in Section 4.

## 4 Example Database Application

In order to give a concrete example of the Elusive Interface keyspace for an application, we have chosen the European Molecular Biology Laboratory's (EMBL) Nucleotide Sequence Database (NSDB) [4]. Essentially, this database collects genome information. Every entry contains a DNA or RNA sequence, as well as additional information about the sequence such as references to the sequence in the rest of the database, the type of molecule that it is, et cetera.

The methods provided by this interface must be invoked on specific instantiations of 4 different types of objects. By examining TCP traces, we learned that the client application sends a 7-

byte object identifier to indicate which object is being accessed, followed by the method identifier and any arguments to that method. Thus, the minimum invocation of a method without any arguments can be sent with 8 bytes. There are 6 methods in this interface which require arguments: 2 methods take a 7-byte object identifier, 1 method takes two unsigned longs (4 bytes total), and the remaining 3 methods take strings. By analyzing some actual values of strings from our trace, we determined that a good maximum value for strings for this application is 30 characters. Thus, the maximum size of an interface message could be set at 38 bytes, and any message with less than 38 bytes could be buffered with dummy data.



**Figure 3:** (a)  $\mathbf{NumFormats(m)}$  for case #4 of the NSDB program, with  $n+1 = 8$  and with an  $offset\_range$  from 41 to 256. (b)  $\mathbf{NumFormats(m)}$  for case #5 of the NSDB application with the number of dummy variables ranging from 1 to 40. This is a logarithmic scale.

Figure 3a shows how the value of  $offset\_range$  effects  $\mathbf{NumFormats(m)}$  for case #4. In this figure, we vary  $offset\_range$  from the actual number of methods in the interface (41) up to the value of two hexadecimal digits (256), which is what the transfer protocol uses to encode the method offset. It is possible to extend the value of  $offset\_range$  beyond 256, and the graph shows that the equation for  $\mathbf{NumFormats(m)}$  will scale linearly with  $offset\_range$ .

Figure 3b is a logarithmic scale of  $\mathbf{NumFormats(m)}$  for case #5, where the number of byte-sized dummy variables changes along the X-axis. There are two lines, one for  $offset\_range = 41$ , and another for  $offset\_range = 256$ . As you can see, the number of dummy variables has a larger effect on the complexity of the problem than the size of  $offset\_range$ .

By buffering all of the method invocation messages to a maximum size of 38 bytes, the number of formats that a particular message may take for  $offset\_range = 41$  is  $4.3 \times 10^{14}$ , and for  $offset\_range = 256$  is  $2.7 \times 10^{15}$ .

## 5 Empirical Evaluation

We implemented an Elusive Interface prototype in order to analyze the overheads of several of the message transformations. Currently, the prototype supports changing the method offset value, increasing the method offset range and changing the message organization as described in cases #0-4 in Section 3. For the experiments, the  $offset\_range$  is set at 256. The prototype does not yet implement parameter buffering.

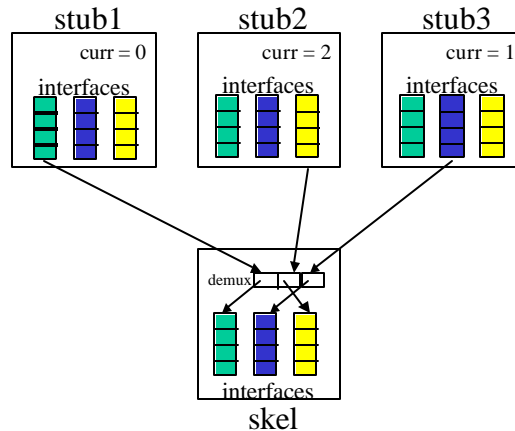
### 5.1 Implementation

We implemented our prototype in Java and the Java RMI framework because Java is portable, and we do not have to deal with the problems of intra-ORB compatibility, as we would have had we chosen CORBA for our framework. While it is widely recognized that Java performance is not yet competitive with other programming languages, our immediate goal is to compare the relative performance of different security mechanisms within the same framework.

Instead of using Sun's RMI, the prototype builds on Berkeley's Ninja RMI [5]. We use Ninja RMI because, among other things, it is open-source and provides infrastructure for authentication (public-key cryptography, DSA) and encryption (Diffie-Hellman key exchange, symmetric cryptography).

For the Elusive Interface prototype, we modified Ninja RMI's transport layer, as well as the Ninja RMI compiler. In the transport layer, a special Elusive Interface connection is setup that uses public-key cryptography and DSA to authenticate the endpoints, and a Diffie-Hellman key exchange in order to exchange an Elusive Interface key. Once this is done, the channel sends data in plain-text as directed by the stub and skel files. Additionally, the transport layer maintains some state for each connection.

We modified the Ninja RMI compiler to produce stub and skel files with multiple interface mutations, and the ability to cycle through them based on the state of the connection. Figure 4 shows the conceptual view of multiple client stubs cycling through a static set of interfaces, independently of each other.



**Figure 4** The stub and skel files have multiple interfaces. Each connection cycles through the set of interfaces independently.

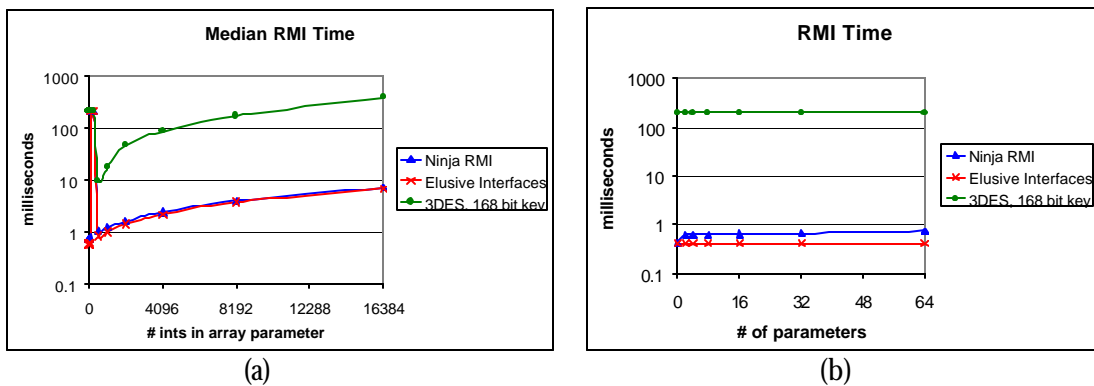
## 5.2 Performance

We ran several tests to measure the overheads of our Elusive Interface prototype, and to compare those overheads to the performance of plain-text communication, as well as session-based, symmetric encryption. Our experimental setup consisted of two 500 MHz Pentium Pro machines running Windows NT 4.0 and connected with 100 Mb Ethernet. All tests (plain-text, Triple-DES 168 bit-key encryption, and Elusive Interfaces) were performed in the context of Sun Java's 1.1.8 JDK, Ninja RMI and Symantec's JIT.

The one-time startup overhead associated with setting up a channel between the two machines is approximately 0.12 seconds for plain-text communication, and 4.2 seconds for both the Triple-DES and Elusive Interfaces channels. Experiments show, however, that there is substantial variation in these numbers between runs.

Figure 5a shows that the overhead added by Elusive Interfaces over plain-text is minimal, proving that Elusive Interfaces can be used effectively in high-speed networking environments. In

the graph, there are three lines representing RMI latency, where the remote method takes an array of integers, and the array size is varied along the x-axis. Elusive Interfaces' performance is within 3-20% of plain-text, Ninja RMI. For the data point of 1K integers in the array parameter, Elusive Interfaces is 11 times faster than Triple-DES, increasing to 56 times faster for a 16K array. Substituting DES in place of Triple-DES would only gain us a factor of 3, making the asymptotic performance of Elusive Interfaces 18.6 times faster than DES. There is a performance anomaly for the base Ninja RMI sending between 123 and 363 integer arrays. The anomaly affects the Triple-DES communication from 0 to 1K integer arrays. We have narrowed the cause of this anomaly down to a conflict between Ninja RMI thread scheduling on NT and array serialization, but have yet to eradicate it.



**Figure 5:** (a) The RMI time, where the remote method takes an array of integers, the size of which varies along the x-axis. (b) The RMI time as a function of the number of integer parameters, as indicated by the X-axis. Note, both use a logarithmic time scale.

Figure 5b shows the overhead of RMI calls as the number of parameters in the method invocation increases. The graph shows that the parameter-order mutation described in Section 3 adds no overhead to the plain-text communication. This result shows that Elusive Interfaces' performance scales with the complexity of an interface, which is important since the degree of security improves as the complexity of the interface increases.

Finally, even though the prototype does not implement parameter buffering, Figure 5b shows that sending 0 through 64 integers makes little difference to the performance. Thus, we are optimistic that incorporating parameter buffering into our prototype will have negligible impact.

## 6 Related Work

The typical approach to securing communication channels is symmetric key encryption[6, 7]. Our approach differs by focusing on the needs of component applications running over high-speed networks. Elliptic Curve Cryptography (ECC) has been proposed as a lower cost form of public-key cryptography, but it is only 10 times faster than RSA[8]; and RSA is about 1000 times slower than DES[9]. Thus, ECC is at least 30 times slower than DES3, making it unsuitable for high-speed network.

Java[10], CORBA[11] and DCOM[12] are messaging middlewares for distributed applications. They each have their own security model addressing management and policy issues; however, they do not specifically deal with security mechanisms for exported interfaces.

A few security systems are beginning to exploit and respond to the issues raised by extensive use of IDL documented distributed object interfaces. In particular, Odyssey Research Associates has built a CORBA Immune System[13], which detects misuse of such public interfaces.

Globus[14] and Legion[15] are projects that have adaptive security in high-performance, distributed systems. The security in Globus focuses on secure process creation and security-enhanced parallel libraries. Their tools enable fine-grained management of security mechanisms. Legion, on the other hand, pushes the responsibility of security into the objects, requiring them to define three security-related methods. While they provide some defaults, it is unclear how extensively the application programmer must think about security.

[16] describes the Deception Tool Kit (DTK)[17] as a type of interface permutation. The DTK deceives attackers by making it appear that there exist servers with known security vulnerabilities. While the attacker is attempting to break into these counterfeit servers, their every move can be recorded. The interface permutation is the location of the actual server, which means that the search space is only as large as the numbers of machines on the network.

Software diversity[18] is used to prevent the same attack from working on all instances of an application. By having multiple implementations, the likely hood that an attack which exploits a

vulnerability on one will be successful on the others, is minimized. Elusive Interfaces is related to software diversity in that the interface mutations on otherwise identical objects are different.

## **7 Conclusions and Future Work**

We described Elusive Interfaces, a low-cost mechanism for protecting distributed interfaces which utilizes message transformations to conceal the communication. We provide an analytical evaluation of the search space in terms of both active and passive attacks; and show that the number of mutated interfaces grows exponentially with the number of methods in the original interface. We describe a prototype implementation with asymptotic performance within 3% of the plain-text channel, and over 56 times faster than the encrypted channel. We believe this makes Elusive Interfaces suitable for high-speed networking environments.

In the immediate future, the source of the performance anomaly shown in Figure 5a needs to be discovered and eliminated. We will also explore the performance of additional interface mutations (such as parameter buffering and message buffering). There are several directions to expand upon Elusive Interfaces in the long term. First, the prototype needs to be ported to a high-speed networking environment, utilizing user-level messaging layers rather than TCP. Secondly, the prototype should be expanded into a system that can dynamically switch between Elusive Interfaces and other security mechanisms such as encryption. Finally, it would be interesting to integrate Elusive Interfaces into an intrusion detection system, which can utilize information about unauthorized invocations to detect an intruder.

## **Acknowledgements**

The research described is supported in part by DARPA orders #E313 and #E524 through the US Air Force Rome Laboratory Contracts F30602-99-1-0534, F30602-97-2-0121, and F30602-96-1-0286. It is also supported by NSF Young Investigator award CCR-94-57809 and NSF EIA-99-75020. It is also supported in part by funds from the NSF Partnerships for Advanced Computational Infrastructure -- the Alliance (NCSA) and NPACI. It is also supported by a National

Science Foundation Graduate Research Fellowship. Support from Microsoft, Hewlett-Packard, Myricom Corporation, Intel Corporation, Packet Engines, Tandem Computers, and Platform Computing is also gratefully acknowledged.

## References

- [1] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In *SuperComputing '98*, Orlando, Florida, 1998.
- [2] M. Malkin, T. Wu, and D. Boneh. Experimenting with Shared Generation of RSA keys. In *Internet Society's 1999 Symposium on Network and Distributed System Security*, 1999.
- [3] Ritter's Crypto Glossary and Dictionary of Technical Cryptography,  
<http://www.io.com/~ritter/GLOSSARY.HTM#Keyspace>
- [4] EMBL Outstation European Bioinformatics Institute, <http://www.ebi.ac.uk/index.html>
- [5] The Ninja Project, <http://ninja.cs.berkeley.edu/>
- [6] Introduction to SSL,  
<http://developer.netscape.com/docs/manuals/security/sslin/contents.htm>
- [7] SSH Secure Shell, <http://www.ssh.com/products/ssh>
- [8] Current Public-Key Cryptographic Systems, <http://www.certicom.ca./research.html>
- [9] B. Schneier, *Applied Cryptography*, 1 ed: John Wiley & Sonce, Inc., 1994.
- [10] Java Security FAQ, <http://java.sun.com/sfaq>
- [11] The CORBA Security Service Specification (Revision 1.2),  
<http://ftp.omg.org/pub/docs/ab/97-05-05.pdf>
- [12] Security in COM+,  
[http://msdn.microsoft.com/library/psdk/cosdk/pgservices\\_security\\_5jbz.htm](http://msdn.microsoft.com/library/psdk/cosdk/pgservices_security_5jbz.htm)
- [13] M. Stillerman, C. Marceau, and M. Stillman. Intrusion Detection for Distributed Applications. *Communications of the ACM*; vol. 42, pp. 62-69, 1999.
- [14] I. Foster, N. Karonis, C. Kesselman, and S. Tuecke. Managing Security in High-Performance Distributed Computations. *Cluster Computing*; vol. 1, pp. 95-107, 1998.

- [15] A New Model of Security for Metasystems, <http://legion.virginia.edu/papers.html>
- [16] C. Cowan and C. Pu. Death, Taxes, and Imperfect Software: Surviving the Inevitable. In *New Security Paradigms Workshop*, 1998.
- [17] The Deception Toolkit, <http://www.all.net/dtk/dtk.html>
- [18] S. Forrest, A. Somayaji, and S. Ackley. Building Diverse Computer Systems. In *6th Workshop on Hot Topics in Operating Systems*, 1997.