RC 20742 (2/20/97)
Computer Science

IBM Research Report

ChakraVyuha[1] *(CV)* : A Sandbox Operating System Environment for Controlled Execution of Alien Code

Asit Dan, Ajay Mohindra, Rajiv Ramaswami and Dinkar Sitaram

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

ChakraVyuha[1] *(CV)*: A Sandbox Operating System Environment for Controlled Execution of Alien Code

Asit Dan, Ajay Mohindra, Rajiv Ramaswami and Dinkar Sitaram
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
{asit, ajay, rajiv, sitaram }@watson.ibm.com

Abstract

Sharing of unknown programs creates an atmosphere of untrust and hence exacerbates the need to secure information and physical resources from any *alien* code. A sandbox approach to execution of such foreign code is proposed in this paper. An alien code is trusted with a set of privileges for accessing *logical* (e.g., callable functions and services) and *physical* (e.g., rate and maximum consumption amount of CPU, memory, disk space, etc.) resources via a third party authentication. During installation of any code in the sandbox environment, the associated privileges (per user basis) are stored in a secure area and enforced by the Operating System during execution. The alien code may also access resources defined by other subsystems (e.g., database), and hence, in an integrated environment the subsystem specific privileges are transferred securely to the subsystem for monitoring. A prototype version based on Linux OS code has been developed.

1 Introduction

The rapid growth in connectivity and sharing of information via the World-Wide Web has dramatically increased the vulnerability of client machines to *alien* codes. Once the alien code is installed on the client machine, it can run with all the privileges of the invoking user. The invoking user may not be aware of the privileges being used. The alien code can silently obtain unauthorized access to all the system resources (i.e., *logical resources* such as files, network ports) that can be accessed by the invoking user. It may also propagate itself, attempt illegal break-ins or maliciously alter files in the client system [10]. Even if the executing alien code is restricted to accessing authorized logical resources only, it may still engage in a *denial of service* attack by monopolizing *physical resources* such as disk, physical memory, CPU, network bandwidth. For example, the alien code may execute a wasteful tight loop which results in preventing any other program from obtaining fair access to the CPU. It may also create many junk files to fill up the available disk space. If this attack is carried out with sophistication (e.g., the tight loop is executed at times when the alien code is supposed to be inactive) the user may be unable to determine the source of the attack. This problem is critical in multimedia systems, where the system attempts to guarantee Quality of Service (QoS) for all streams by reservation of shared resources. QoS guarantee can be provided to all streams only if all streams adhere to their allocated quota. Deliberate or accidental overuse of a shared resource by a stream can result in a failure to meet QoS guarantees for other streams and hence, interrupted stream delivery. The development and popularity of the Java programming language [5] has made it relatively

[0]In Hindu mythology, Mahabharata, *ChakraVyuha* (pronounced as chakra-view-ah) refers to the dynamic formation of multi layer wheel made of shields that is very difficult to penetrate in a battlefield.

easy to transport alien code over the World-Wide Web. However, even in computer systems not connected to the Web, programs can be downloaded from remote sites or installed from diskettes or CDROMs. Hence, the problem of providing a secure environment for the execution of programs exists even in traditional environments.

One solution to the above problem is to restrict via a *reference monitor* [8] the privileges that can be acquired by any alien code and to ensure that it cannot compromise the security of the system during execution. Java *applets*, for example, cannot write files on the host machine, or send messages to nodes other than the machine they were downloaded from [5]. This approach cannot be used in general, since it is difficult to devise a set of restrictions without limiting the usefulness of an alien code. Removing the restrictions (as is done, for example with Java *programs*) re-instates the security problem. A similar security problem exists with importing data for word processors and spreadsheets, where the imported data could be infected with *active content macro viruses*, and these viruses in turn, could try to access system resources without the knowledge of the user. To alleviate the problem, the macros associated with the data need to be monitored. Operating system support is also required to associate varying privileges with different programs of a user, rather than simply with individual users. For example, a multimedia application may require access to video files, and hence, guaranteed access to disks at a specified rate, while a non-multimedia application may only require access to non-video files. Therefore, an important requirement for any solution is that an alien code be granted the privileges necessary for its execution, but not be allowed any unrestricted access to the system.

In this paper, we propose a solution to the above security problem. The solution combines certification of alien code together with explicit granting of privileges for execution. In Section 2, we provide an overview of our approach embodied in the *ChakraVyuha* environment. A detailed discussion of the approach is provided in Section 3. Section 4 describes prototype user environments that can be built on top of ChakraVyuha. Section 5 contains our concluding remarks.

1.1 Motivating Examples

We now describe three example application environment scenarios that rely on varying degrees of trusts or use different enforcement processes for verification. These are the special cases of the general approach described in this paper. All three examples are viable applications in the near future.

Intranet environments: A simple security model that relies primarily on trust rather than complex runtime enforcement mechanism [10] may be employed in an intranet environment with inexpensive Network Computers. The network computer environment could be customized for a specific organization allowing execution of only the trusted applications (authorized code with proper certificates) without further monitoring of their execution.

Multimedia support: This requires runtime monitoring of execution for the purpose of controlling resource consumption by any code (alien or local). Typically, the code would be required to declare its resource needs, and the runtime environment would enforce resource usage accordingly. For example, video conferencing applications require multimedia support and access to physical resources such as the network.

Sandboxed browser environment: A more general environment than the simple trust model is required for the execution of alien code. Different helper applications and/or downloaded components may require different access capabilities to logical and physical resources, and a runtime environment would enable execution of the code in a controlled environment. Untrusted code would be run with limited access privileges. A Java or ActiveX enabled browser is an example of this environment.

1.2 Related Work

Enforcing security for networked computers has been an area of active research for quite some time. Most of the work, however, has focussed on preventing unauthorized accesses to the computer systems from the network using challenge/response passwords and authenticated token schemes such as Kerberos [11]. Other work has focussed on addressing trust issues for applications that can be downloaded from the network. Based on the level of trust placed by the runtime system, the work can be classified into three categories. Systems in the first category do not trust any program, i.e., the runtime system views all programs to be potentially malicious, and takes adequate measures to ensure that the programs do not cause any damage. The KeyKOS [7] system and the Domain and Type Enforcement (DTE) system [12] fall under this category. These systems are pure capability based systems unlike most commercial operating systems (e.g., Unix, Windows NT, etc.). In KeyKOS system, all applications are executed in protected *domains* that preserve system integrity. Similarly, the DTE system uses a runtime database incorporated into the operating system structure to control access of processes to system objects. These systems use statically defined capabilities for enforcing strict security, and thereby making them inflexible in their usage. For example, in DTE any change in application capabilities to be effective requires a complete reboot of the system.

Systems in the second category trust certified authors to write programs that do not exhibit malicious intent. Issues such as trust-worthiness of applications have become important due to the growing popularity of the World-Wide Web and widespread availability of downloadable freeware such as ghostview and mpeg player. The Betsi [10] system and the work on application specific-interpreters [6] fall under this category. The Betsi system provides an infrastructure wherein the software developer can attach a digital certificate to the application code, which the user can then use to verify the authenticity and integrity of the software. The system thus provides a mechanism for users to verify whether any unauthorized modifications have been made to the software by hackers. In the work on application-specific interpreters [6] the authors describe an architecture that allows the software developers to specify the resource and access control requirements of the software. This information is then used to generate an application-specific interpreter which monitors the accesses of the downloaded applications. Their system architecture also defines system services for access control of the downloaded application.

In general, a higher level interpreter cannot monitor/control fine granularity accesses to various system resources as well as logical and physical resources owned by another subsystem (e.g., database, physical resources of the server node). Without a detailed execution plan corresponding to a subsystem access that enumerates the usage of various sub-system resources (e.g., mapping of an SQL query execution to database records and fields), it is impossible to monitor all accesses. Hence, sub-systems are the best place to monitor/control accesses to resources owned or defined by them.

In the absence of mechanisms for resource access control per application per user, the systems in the third category trust the operating system to provide adequate mechanisms and policies (via a higher level reference monitor) to protect users from suspect programs. The Janus system [4] falls under this category. Janus is a user-level implementation of a "sandbox" architecture that enables a user to restrict an application's access to the operating system resources, and thereby, reducing the security exposure to a malicious application. In a configuration file, the user specifies the access control list for all untrusted applications. Such untrusted applications are then executed in the sandboxed environment that restricts the application's access to operating system resources based on the access control lists specified in the configuration file. The sandbox environment acts as a reference monitor for controlling accesses to system calls. The difference between their scheme and our work lies in the overall design and the scope of resources that are protected. The Janus system is primarily concerned with protecting application's access to the file system, and communication resources. The system does not handle *denial of service* attacks by untrusted applications. Our work provides a generic framework that allows the operating system to authenticate and verify installed applications, dynamically control access to both logical and physical resources, and allow privileges associated with an application to be transferred to any other application and/or subsystem.

2 Overview of the Approach

Our approach incorporates three major items:

specifying in a flexible manner in a *resource capability list* (RCL) the set of permissions and resource access privileges required by the code,

having the code and its RCL authenticated by a trusted third party, before a client receives them, and finally,

enabling the client system to agree and allocate all or a subset of the permissions and resources specified in the RCL and enforcing these capabilities, i.e., ensuring that the permissions are not violated or resource consumptions have not exceeded specified limits.

The overall system that implements this approach is shown in Fig. 1. The system consists of one or more code production systems, certification agencies, servers and clients. A code production system communicates with a certification agency, which is a trusted third party. The certification agency issues a certificate for the code identifying its source and a certificate for the RCL of that code. The separate certificates for the code and the RCL allows the two entities to be dealt separately, i.e., they could be downloaded separately. Once the certificate is issued, it is not possible for any party to modify the code or RCL without invalidating the certificate. The code and its RCL are stored on a server along with their certificates. Note that while the above figure describes the most general framework, all of the components need not be present in all implementations. For example, the RCL may not be defined in advance for all alien codes. In this case, the RCL may be the default RCL associated with the sandbox environment in the client system, or it may be defined on-the-fly during installation at the client system. Further details on RCL definition are provided in Section 4.

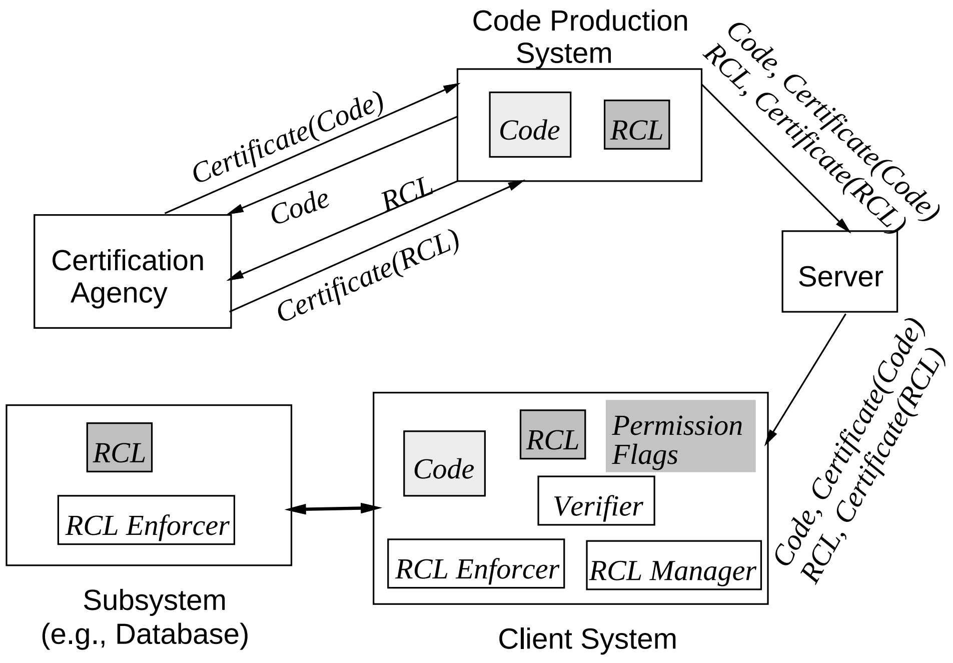


Figure 1: A. Block diagram of the proposed system.

The main focus of this paper is on the client system. Within the client system, the functions described below can be implemented either within the operating system kernel or as another layer on top of the operating system (see, Figure 1). The shaded areas represent entities such as code, RCL or environment specific permission flags. The system monitor layer (i.e., the CV layer) consists of a verifier of code certification, RCL manager that maintains RCLs and a RCL enforcer that performs a “gatekeeping” function, i.e., controls and monitors access to system resources. When a client receives the code and its RCL, it invokes the verifier to verify that the code and RCL are valid (not tampered with). Once the code is verified, the RCL manager is responsible for determining which subset of parameters within the RCL are to be allowed. It then stores the code and its modified RCL in a secure location. The RCL manager can also be invoked subsequently by a privileged user to alter these capabilities. The RCL enforcer is invoked when the code is executed. It continues to monitor accesses by the code to ensure that the permissions and resource consumption limits are not violated. The subsystems may have their own set of resources to protect. Hence, the RCL manager is also responsible for transferring securely to other subsystems relevant capabilities (e.g., data access capabilities relevant to the database subsystem).

It is possible for different users to be allocated different types and/or amounts of resources and permissions in the recipient system for the same code. This can be done at the time the code is installed by the RCL manager. (The issues on ease-of-use are addressed in Section 4.) The RCL manager looks at the privileges given to different users and combines (i.e., intersects) those capabilities with the resource access privileges and permissions allowed to the code. In this case the set of resource access privileges and permissions for each user would have to be stored separately. Additional enhancements or reductions of capabilities on a per-user basis for a system wide installed code are also possible. During code execution the resource consumption limits and permissions would be enforced on a per-user basis. Alternatively the resource consumption limits and permissions can be determined during execution of the code, where the RCL enforcer determines dynamically the permissions and limits, by looking at the privileges given to the invoking user and combining those allowed to the code. The details of the implementation are described in Section 4.

3 Detailed Design

3.1 Trust and Authentication

Since the code to be executed could come from a variety of unknown, and possibly, untrusted sources, we need to find a way to guarantee the identity of its author and secure its integrity. The same applies for the RCL[1] that comes with the code as well. Our approach here is similar to the one proposed in [10]. A code production system communicates with a Certification Agency (CA), which is a trusted third party. The CA issues a certificate for the code and a certificate for the resource list of that code. Once the certificate has been issued, it is not possible for any party to modify the code or resource list without invalidating the certificate. The code and its RCL, along with their certificates are stored on a server. A client downloading the code or access list can verify the integrity of the code/resource list.

The CA provides a public key K that is widely available and known to the client system. In addition, the CA has a private key P. To provide a certificate for the code, the CA creates

[1]Here, the RCL is the declaration of permissions and physical resources requested by this code for its execution. Alternative interpretations are possible for a RCL associated with a downloaded code, where it may represent already granted capabilities.

a certificate containing the code name and the cryptographic hash of the code and signs it using its private key. The certificate cannot now be changed without invalidating the CA's signature. Similarly the CA creates and signs another certificate for the RCL associated with that code (if desired there could be a single certificate for both the code and its RCL).

Upon obtaining the code/RCL and its certificate, the verifier in the client system first checks to see if the CA's signature on the certificate is valid (using CA's known public key). It then computes the cryptographic hash of the code/RCL and verifies that it matches that in the certificate. If the signature is not valid, (i.e., the hash does not match,) the code and RCL are rejected.

Once the code and its RCL have been verified, the client system may elect to allow certain privileged users to modify the code or the RCL (See, Section 4 for further details). However other users should not be allowed to modify them. Thus the RCL and permission flags must be stored in a secure area; reading or updating this area is itself a privilege enforced by the RCL enforcer. Finally, CV allows downloading (or installing from CDROM) of any code without any certificate as in the conventional systems. Such untrusted codes when executed via sandbox environment will be given minimum default resource capabilities.

3.2 RCL Management and Enforcement

RCL management and enforcement is a function that must be performed by the operating system. This function will require a gatekeeper/monitor layer within the operating system that is capable of allocating, keeping track, and enforcing resources and permissions dynamically. A separate reference monitor associated with each subsystem will enforce privileges specific to that subsystem. During execution, the RCL manager transfers securely the privileges associated with each subsystem to its associated reference monitor. In the following, we describe the format of the RCL together with the details of the other components.

3.2.1 RCL format

The RCL is divided into several categories. Apart from the capabilities related to the system resources, there may be various other subsystem related capabilities. The capabilities for the system resources can be divided into two parts: (a) the capabilities for the physical resources specifying the required physical resources and (b) those for the logical resources (e.g., files, network ports) that need to be accessed. Each item in the physical resource list contains four fields. The first field is the name of the physical component (e.g. disk, physical memory, CPU). The attribute field specifies the actual resource in the case that there is more than one resource associated with a physical component (e.g. space or bandwidth in the case of disk). The other two fields specify the maximum allowable consumption and consumption rate of the resource, respectively. Note that while some resources refer to the same amount in all machines (e.g., disk space), specification of other resources (e.g., required CPU MIPS) may be machine dependent. Either the amount needs to be expressed for a reference machine, and translated for the client machine during code installation, or the amount should be expressed in a machine independent manner (e.g., Bandwidth or data movement per unit time).

Logical Resources

Syntax:

```
<Statement> ::= <Logical resource name> <Parameter Definition> <Range List>
<Logical resource name> ::= <Package name> < Function name>
<Parameter definition> ::= <Parameter type>,...
<Parameter type> ::= INT | STR
<Range list> ::= [ <ParmSet range>, ... ]
<ParmSet range> ::= {<Parm range>, ...}
<Parm range> ::= <Integer range> | <String range> |
<Integer range> ::= <Integer> | <Integer> − <Integer> |
                      <Integer>, <Integer range> |
                      <Integer>−<Integer>,<Integer range>
<Integer> ::= <Simple integer> | <Environment variable>
<String range> ::= <Environment variable>, <String range> |
                   <String>, <String range>
                   <epsilon>
```

Figure 2: BNF for the language used to specify the logical resource list.

Access to logical resources, or client system facilities, is controlled by specifying the list of functions external to the program that may be invoked, together with restrictions on the allowable parameter combinations. For example, restrictions on the files that can be accessed in read-write mode can be enforced by specifying the allowable values for the filename parameter in the `open` call when the open mode is read-write. In many situations, it may be necessary to allow access to resources whose name is client-system dependent. For example, a database query program may need access to a database, whose name may vary in different client systems or between different users on the same system. This is handled by allowing the use of environment variables whose values can be determined during execution. Changing of such environment variables is itself a privileged operation.

Figure 2 shows, in BNF, the syntax of the language used to specify the logical resource list. Informally, each statement in the language specifies a valid combination of parameters for a particular function. The statement consists of the function name, followed by the types of the parameters and the range of allowable values for each parameter. The function name is a combination of package name and the actual function name to handle the case where the same actual function name (e.g. `open`) is used by multiple subsystems. The parameter types may be integer or string, denoted by `INT` or `STR`, respectively. For integer parameters, the allowable values are specified as a range of integers. For string parameters, regular expressions (in the Unix sense) are used to specify the allowable values. Environment variables are specified by prefixing the name of the variable with `$`; for example `$MAXINT` may specify the maximum integer value on the system and `$USER_DB` may specify the name of a user-specific database.

Note that the logical resource list should not include specifications of the type "access all files except <file-list1>". Specifications of this form implicitly assume that the only files to which access needs to be restricted is <file-list1>. This approach is adequate if the only files that need to be protected are a well-known set of system files that are the same on every system. In general, however, the files that need to be protected on client systems may vary. Hence, it is safer for programs to explicitly specify the resources needed. However, set difference may provide ease of expression as shown in Section 4 (See, Figure 11).

The representation of subsystem capability will be specific for that subsystem, and is beyond the scope of the current paper.

3.2.2 RCL Enforcement

The RCL enforcer module at the client ensures that the permissions and resource consumption limits specified in the RCL are not violated. The module for enforcing system resource capabilities monitors two types of resources: physical and logical. In this section, we describe the algorithms and data structures used by the RCL enforcer for these two types of resources.

The RCL enforcer protects physical resources such as memory and CPU usage, disk storage, and disk bandwidth utilization. Access control information about physical and logical resources of the process executing an alien application code is maintained in data structures called the *Runtime Physical Resources Table (RPRT)* and the *Runtime Logical Resources Table (RLRT)*. For easy access to this information at runtime, pointers to these tables exist in the *u-area* of the process (see Figure 3). When an application is initially loaded for execution, the RPRT and RLRT data structures are initialized from the information specified in the RCL configuration file associated with this application. In the next two subsections, we describe in detail the RPRT and RLRT data structures and their use in monitoring of resource accesses.

3.3 Physical Resources

The RPRT data structure contains a row for each resource access capability (see Figure 4). The *current consumption rate* and *current usage* fields are used to keep track of the current consumption rate and the total resource consumption so far, respectively, of the code during run time. The RCL enforcer also keeps track of the code start time, $CodeStartTime$.

During process execution, the RCL enforcer references and updates the information stored in the RPRT data structure using the algorithm outlined in Figure 5 to allow or deny the request for accessing physical resources. Two parameters, the amount of requested resource, $REQAMT$, and the estimated time of consumption, $COMPT$, are passed for each

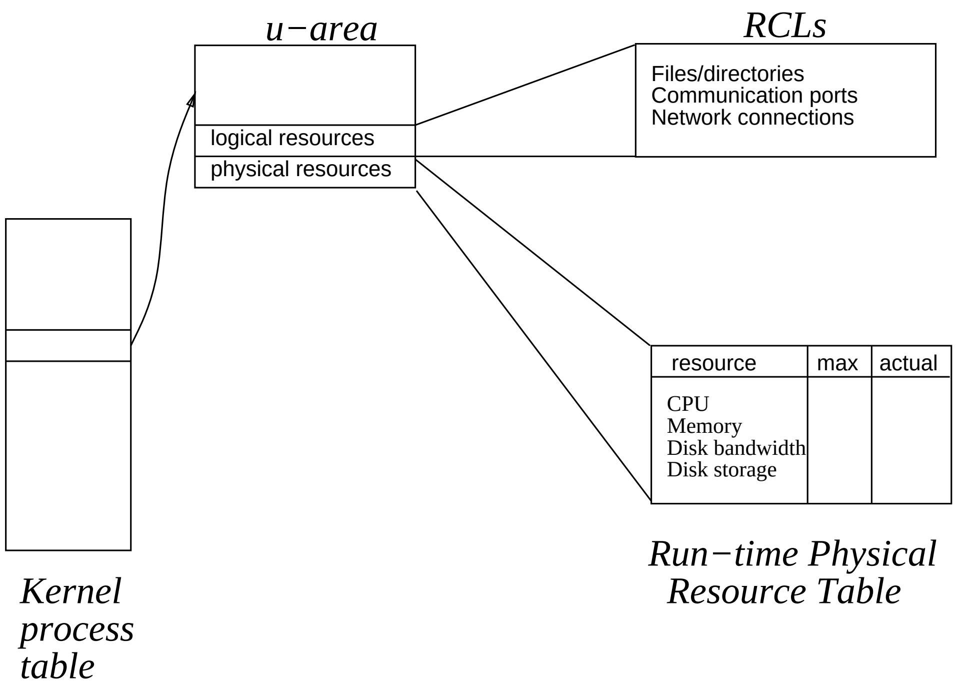


Figure 3: Data structures used by the RCL enforcer.

Physical Resource name	**Physical Resource attribute**	**Maximum consump. rate**	**Maximum usage**	**Current consump. rate**	**Current usage**
Disk	Space		100 MB		*y1*
Disk	I/O	100/s	10 MB	*x2*	*y2*
Physical Memory			8MB		*y3*
CPU	50 Sec	50 MIPS	1 sec	*x4*	*y4*

Figure 4: Runtime Physical Resource Table

access request for a physical resource. For disk I/O, the $REQAMT$ is the amount of disk I/O and the $COMPT$ is the estimated time for the I/O to complete. The RCL enforcer then locates the row in the RPRT for this resource and checks whether the maximum consumption amount, $MaximumUsage$, is specified, and if the $(CurrentUsage + REQAMT)$ exceeds the $MaximumUsage$. If so, it returns a failure indicating that the consumption is not allowed. Alternatively, if it does not exceed the $MaximumUsage$, the RCL enforcer computes the projected consumption rate, $NewConsumptionRate$, of this resource as $(CurrentUsage + REQAMT)/(CurrentTime + COMPT - CodeStartTime)$. The RCL enforcer then checks to see if the $MaximumConsumptionRate$ is specified and if the $NewConsumptionRate$ exceeds $MaximumConsumptionRate$. If the $MaximumConsumptionRate$ is not specified or the projected consumption rate is not greater, the RCL enforcer returns with an indication that the consumption is allowed. If the projected consumption rate exceeds its limit, the RCL enforcer computes the required delay before this operation is allowed as $(CurrentUsage + REQAMT)/MaximumConsumptionRate - (CurrentTime + COMPT - CodeStartTime)$ and returns the required delay with an indication that the consumption has to be delayed for the computed delay.

```
CheckConsumptionRequest(){
    If (( REQAMT + CurrentUsage) > MaximumUsage))
        return( FAILURE );
    Compute NewConsumptionRate for this request;
    If (NewConsumptionRate > MaximumConsumptionRate){
        Compute MinimumDelay before consumption;
        return( MinimumDelay);
        }
    return( SUCCESS );
}

UpdateCurrentUsage(){
    CurrentUsage = CurrentUsage + REQAMT ;
}
```

Figure 5: Algorithm used by the RCL enforcer to manage physical resources.

Once the consumption request is allowed, it is put on the queue for that resource. If the resource is currently available, the request is serviced. Otherwise, the resource scheduling policy (as in traditional systems) prioritizes these allowable requests. For example, the multimedia requests may be given a higher priority to guarantee QoS [9].

After the consumption of the resource, the RCL enforcer is invoked again with a parameter specifying the amount of resource consumed, $CONSAMT$. The RCL enforcer then updates the actual resource consumption, *CurrentUsage*.

3.3.1 Enforcement of Limits for Child Processes

It is critical that any secure system enforces fairly the limits for all processes including child processes. There are various alternatives that can be used. First, each child process may be assigned the RCLs and resource consumption limits of the parent process. While the policy is simple to implement, and enforces logical accesses correctly, it creates a caveat for bypassing resource limits. A process can fork many child processes for circumventing these resource limits. The second alternative is to use the limits of the parent process as the maximum capability of the process and all its children, i.e., the sum total of capabilities of the parent process and all its children cannot exceed the original limit of the parent process. For ease of tracking, and for efficiency, each child process may be assigned a fraction of the resources available to the parent process. The allocation policy decides how to divide the available resources amongst the forked processes. In our implementation, we chose the second alternative. The RCL enforcer equally divides the limits of the parent process among all its children, and checks against this limit at runtime.

3.4 Logical Resources

Similar to the RPRT, access control information for the logical resources is maintained in a data structure called the Runtime Logical Resources Table (RLRT). The logical resources constitute file, directories, communication ports and various other system service calls. The RLRT data structure is also initialized when an application is loaded into memory. The RLRT contains a row for each system service access capabilities. Each row has a flag that indicates whether this service call is allowed *always, never* or needs to be *verified* ($ALWAYS$, $NEVER$ and $VERIFY$, respectively). If the system call needs to be verified, then a list is provided that contains both the allowable and non-allowable ranges of parameter combinations (See Figure 6).

3.4.1 Enforcement of Access Control to Logical Resources

The RCL enforcer monitors all systems calls, i.e., access requests to logical resources in the system. This monitoring requires that the RCL enforcer be invoked for each system call made by an application. The RCL enforcer then validates the parameters of the system call against those specified in the RLRT data structures and either allows or disallows the call from continuing.

The RCL enforcer references the information stored in the RLRT data structure using the algorithm outlined in Figure 7 to allow or deny the request for accessing logical resources. During each system call made by the application code, the RCL enforcer locates the number of parameters, their values, and the name of the system call being invoked. The exact method for doing this is implementation dependent (discussed later); for example, in Java, these are located on the operand stack. The RCL enforcer then locates the row for

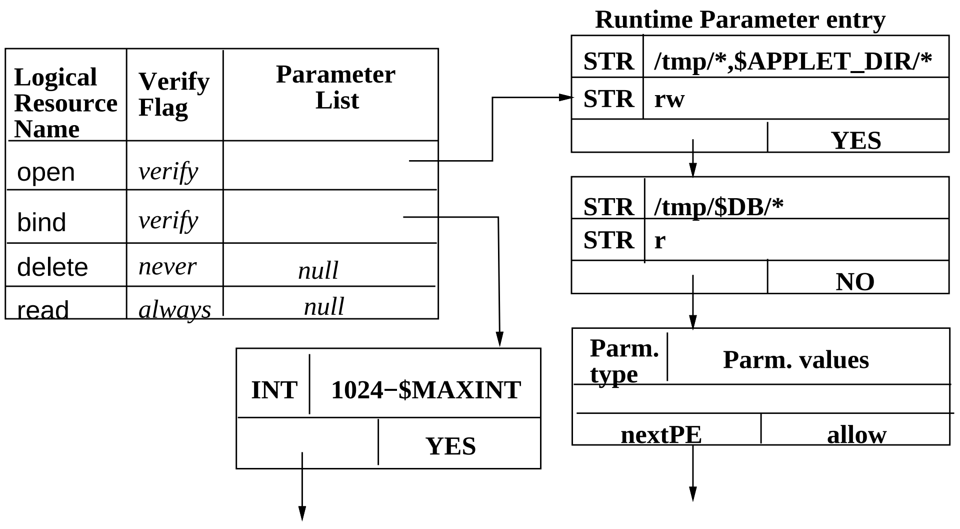


Figure 6: Runtime Logical Resource Table

this function in the RLRT and executes the algorithm in Figure 7. If the given parameter combination for this call does not lie in any of the allowable ranges then the call is not allowed. Otherwise, the $AllowFlag$ is first set to YES and subsequently, further checked to see if the parameter combination lies in a non-allowable subrange (within the allowable range). The call is allowed only if no such subrange is found.

```
CheckAccessRequest(){
    if (VerifyFlag == ALWAYS ) return ( YES );
    if (VerifyFlag == NEVER ) return ( NO );

    Set AllowFlag to NO ;
    If found( allowable parameter range entry that includes
              the current parameter combination ) then
                Set AllowFlag to YES ;
    If found( non−allowable parameter range entry that includes
              the current parameter combination) then
                Set AllowFlag to NO ;
    Return(AllowFlag);
}
```

Figure 7: Algorithm used by the RCL enforcer to manage logical resources.

3.4.2 Implementation Choices for Monitoring Accesses to System Resources

We now describe three different approaches to monitor systems calls:

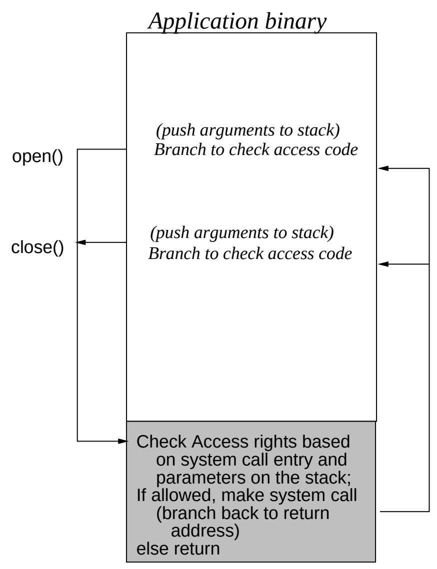
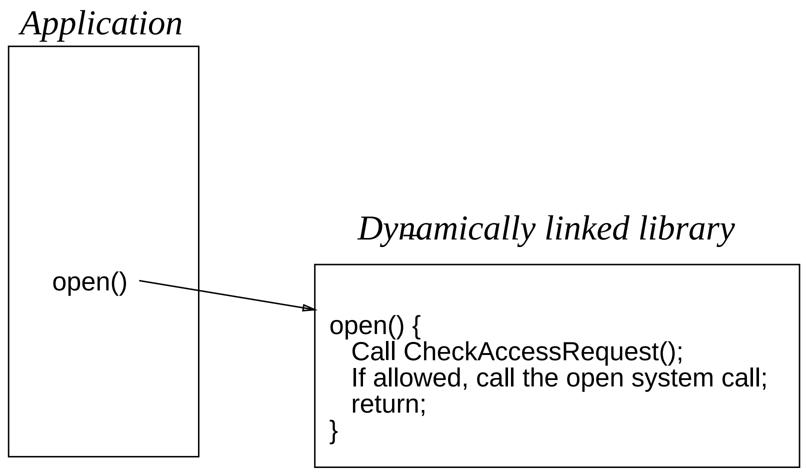

Application binary
(push arguments to stack)
Branch to check access code
open()
(push arguments to stack)
Branch to check access code
close()
Check Access rights based on system call entry and parameters on the stack;
If allowed, make system call (branch back to return address)
else return

Figure 8: Preprocessor for modifying an application's binary code.

- **Binary code modification:** – In this approach, the downloaded application code is preprocessed and modified to insert hooks in the binary code for calling RCL enforcer (i.e., `CheckAccessRequest()` function. See, Figure 7 and 8). Note that all the required parameters for system calls as well as the return address are already placed on the stack. The `CheckAccessRequest()` code verifies these parameters and then branches to the system call routine. At the end of the system call it returns to the original location in the main code. This approach does not require any modifications to the operating system in order to implement any security checks. On the downside, the user has to preprocess all untrusted applications. This may incur a lot of overhead. Also, this is not a full proof solution, since a program may use encryption to hide system calls in its code.

- **Modification of service libraries:** – The idea here is to provide on each machine a duplicate set of libraries for various services that may make system calls (e.g., file system, language support, etc.). The new set of libraries calls the `CheckAccessRequest()` routine to verify the parameters before making any system call. Figure 9 shows the structure of the modified system call library approach. The advantage of such a scheme is that no changes to the operating system are required. On the downside, the system cannot enforce security for applications that have system call libraries statically linked into the application.

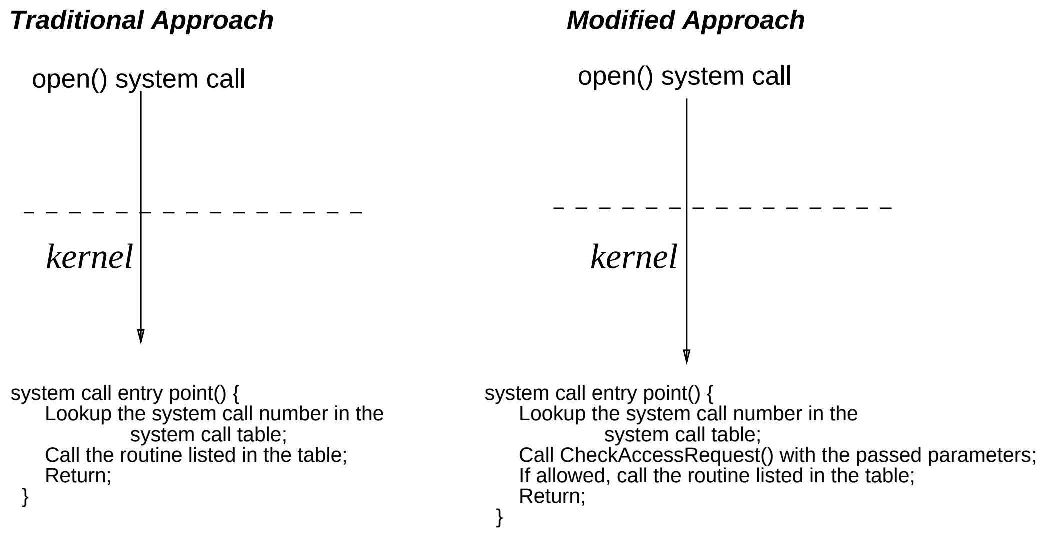
- **System call modification:** – In this approach, the operating system's call interface is modified to call the `CheckAccessRequest()` upon entry to verify the parameters of the system call. Thus, regardless of how the applications are structured, the system call entry point in the operating systems will ensure consistent security checks for all applications that are executed on the machine. The disadvantage of this scheme



```
open() {
   Call CheckAccessRequest();
   If allowed, call the open system call;
   return;
}
```

Figure 9: The modified system call library approach.

is that the operating system needs to be modified. We selected this approach to manage logical resources in the system primarily because of the consistency it offers for enforcing security. Figure 10 shows the structure of this approach.



```
system call entry point() {
     Lookup the system call number in the
                  system call table;
     Call the routine listed in the table;
     Return;
 }
```

```
system call entry point() {
     Lookup the system call number in the
                  system call table;
     Call CheckAccessRequest() with the passed parameters;
     If allowed, call the routine listed in the table;
     Return;
 }
```

Figure 10: The modified operating system call interface approach.

3.4.3 RCL Enforcement by Subsystems

The common entry point to the Kernel for all system calls provides an effective mechanism for enforcing access control for logical resources. However, enough information may not be available to adequately enforce security at the time a system call is being made. For example, in case of filesystems, it is very difficult to resolve at the system call layer whether a request for a file specified using relative paths (e.g., *../../foo*) translates to an access that should be allowed or not. This can only be determined in the file system module after the relative path has been translated into an inode for the file. Hence, in order to be effective, the RCL enforcer has to maintain a partial copy of the state maintained by various subsystems. This however, could add to the overhead and space requirements for RCL enforcer. Therefore, to simplify the design and minimize overhead, the RCL enforcer could be configured to hand off enforcement for a subset of system calls to the appropriate subsystems (e.g., File system). During process initialization, the RLRT entries for the system calls which are handed off to sub systems are set in ALWAYS mode. Finally, for non-system calls, i.e., accesses to the resources defined and managed by subsystems independent of the OS (e.g., database relations), the enforcement is naturally carried out by those subsystems.

4 Installation and Use of Applications under ChakraVyuha

We first describe the structure of a configuration file, and various types of default (e.g., system wide, per user) and application specific configuration files used by the CV. We will then detail the installation process of untrusted applications.

4.1 Structure of a configuration file

Each configuration file specifies the RCL for an application (or user). Figure 11 shows the structure of a typical configuration file. The configuration file begins with a series of import declarations that include privileges defined in other configuration files. The import declaration also allows importing of fine granularity privileges from another configuration file, i.e., privileges for a specific module. A sequence of MODULE declarations, which define privileges for various modules, follows the import declarations. For example, as shown in 11, the first block defines the module SYS_PHYSICAL and the physical resource requirements for the application. This includes CPU , physical memory, I/O bandwidth and network requirements. The next block, called the MODULE SYS_LOGICAL, specifies the logical resource requirements for the application. The list includes access controls for various system calls allowed to this application. The keywords reserved for defining this module are as follows:

- **ALWAYS –** Always allow the application to make the specific system call without any need for verification.
- **NEVER –** Never allow the application to make the specific system call.
- **VERIFY –** Verify the arguments before making the specific system call. The keywords following the verify reserved word describes how the privileges for the system call are specified. For example, the figure shows that arguments for the `open` system

```
IMPORT <cfg_file_name>;  /* import one or more cfg files */
  IMPORT <cfg_file_name>::MODULE <module_name>;
                          /* import only the privileges of
                             a particular module */

   MODULE  SYS_PHYSICAL   /* Physical resource privileges */
     CPU <max_amount> <max_rate>;
     IO <max_amount> <max_rate> ;
     MEMORY <max_amount>;

   MODULE  SYS_LOGICAL    /* Logical resource privileges */
     exit ALWAYS;    /* Always allowed */
     setuid NEVER;   /* Never allowed */
     open VERIFY STR, INT, INT   /* verify calls */
           [{"/tmp/*", *, *} - {"/tmp/admin/*", *, *}],
           [{"/foo*", {0, 1, 2}, 2..3} -
{"/fooblatz", {1, 2}, 2..3} -
            {"/fookazam", *, *}];
     unlink NEVER;

   MODULE  <subsys_name>  /* Sybsystem  privi-
leges (e.g., DB2) */
     < subsystem related privilege description>
   MODULE  <subsys_name>
     < subsystem related privilege description>
```

Figure 11: A sample configuration file.

call need to be verified before allowing the system call to complete. The privileges associated with the `open` system call are described in the form of a <*string, int, int*> corresponding to the types of the parameters that the open system call takes, i.e., *pathname, flags, mode*. Following this syntax specification, a list of comma-separated entries are described which specify the application privileges for an open call to files in various directories. The example shows that the open call should be allowed in any mode for all files in the `/tmp` directory and not be allowed for any files in the `/tmp/admin` subdirectory. The privileges associated with a system call is a union of all the privileges in the comma-separated list.

The application writer can also specify privileges for various subsystems that the application would use. For example, if an application communicates with a database subsystem such as DB2 then the application writer can specify the requirements for that subsystem using the MODULE DB2 declaration.

4.2 Types of configuration files

The different types of configuration files used in ChakraVyuha are described below. The first two configuration files (*cvmin.cfg* and *cvmax.cfg*) relate to a specific user environment, and describe the bounds on access privileges allowed to any code execution instantiated by this user. The configuration files (*sysdefault.cfg* and *userapp.cfg*) refer to the specific privileges granted to a specific application. Finally, the user *cvrc* file allows an user to specify to the CV use of a different configuration file for that instantiation.

- User Min Configuration File: The User Min Configuration File (hereafter abbreviated as *cvmin.cfg*) specifies the maximum resource consumption of a process running in a ChakraVyuha sandbox for which a configuration file could not be located. Ideally, this file will never be used since proper configuration files would always be in place. The user may customize this file. The location of cvmin.cfg file is illustrated in Figure 12 (see, for instance, `/cv/config/u/susan/cvmin.cfg`).

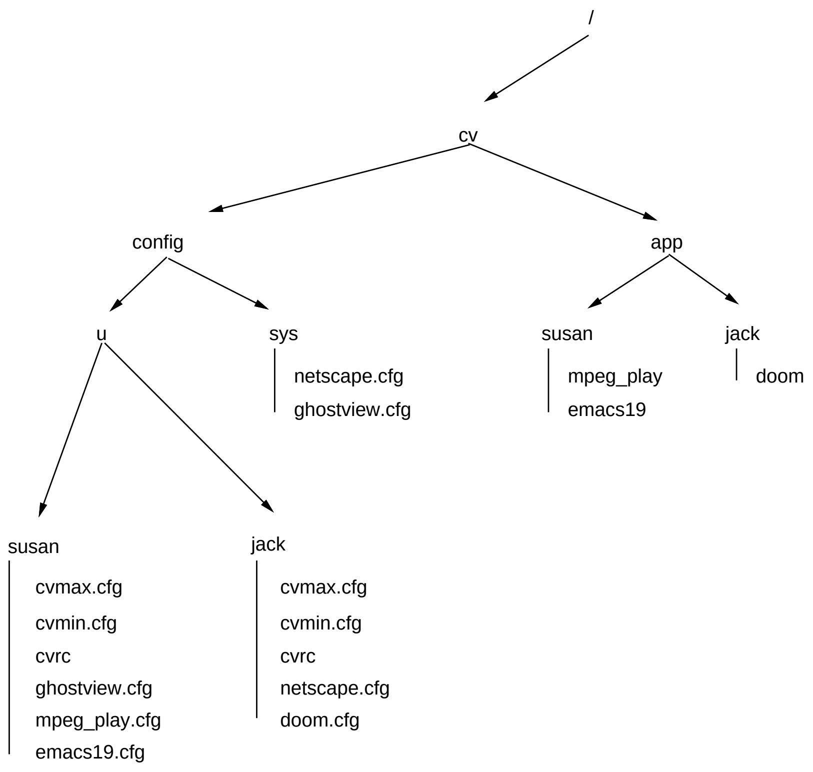


Figure 12: Configuration file hierarchy.

- User Max Configuration File: The User Max Configuration File (hereafter abbreviated as *cvmax.cfg*) specifies the maximum resource consumption (e.g., CPU rate) of any process instantiated by the user. The parameters in cvmax.cfg are set by the system administrator and cannot be modified by the user. In Figure 12, `/cv/config/u/susan/cvmax.cfg` is a cvmax.cfg file.

- System Default Application Configuration File: The system default application configuration file (hereafter, abbreviated as *sysdefault.cfg*) specifies system defaults for an application. It is generated during installation from the application RCL. The sysdefault.cfg file is optional and is provided to facilitate the use of the CV. For example, in Figure 12, `/cv/config/sys/netscape.cfg` is a sysdefault.cfg file for the Netscape browser.

- User Application Configuration File: The user application configuration file (hereafter abbreviated as *userapp.cfg*) allows the user to customize the sandbox. A userapp.cfg will *replace* a sysdefault.cfg. That is, if a userapp.cfg is specified when the sandbox is invoked, the sysdefault.cfg will be disregarded. However, if the user wishes to build upon the restrictions specified in the sysdefault.cfg, the user can `import` the sysdefault.cfg and specify enhancements to and overrides of its specifications on the granularity of each resource.

  An example snippet of a userapp.cfg file follows. In the directory structure illustrated in Figure 12, this file would be `/cv/config/u/susan/ghostview.cfg`.

```
# Snippets of Susan's ghostview.cfg file
#
# The following line imports the system default
# ghostview.cfg file.
  import /cv/config/sys/ghostview.cfg

MODULE SYS_LOGICAL
# The following line enhances the restrictions on
#the open system call.
  open ENHANCE VERIFY STR, INT, INT
        [ - {''~/Private/*'', *, *} ]

# The following line overrides the restrictions on
# the unlink system call.
  unlink OVERRIDE VERIFY STR [ {''/tmp/*''} ]
```

- User cvrc File: The User cvrc File allows an user to specify CV parameters without retyping them at every CV invocation. It is similar in nature to a `.cshrc` file or a `.Xdefaults` file. An example snippet of a cvrc file follows. In the directory structure illustrated in Figure 12, this file would be `/cv/config/u/susan/cvrc`.

```
# Snippets of Susan’s cvrc file
#
# The following option requires that a user app
# config file be processed before the CV sandbox
# may be constructed.

# The following line specifies the pathname for the
# user app config file to be used for emacs19.
  emacsv19*UserAppCfg: /cv/config/u/susan/emacs19.cfg

# The following line specifies that Susan wants to use
# Jack’s config file whenever she plays Doom.
  doom*UserAppCfg: /cv/config/u/jack/doom.cfg
```

4.3 Installation of Untrusted Applications

For effective enforcement of security, all untrusted applications are installed in the system via an uniform installation process. An utility called the `InstallCop` is used for this purpose. The steps involved in the installation process are as follows:

1. The application and RCL are downloaded to the host (via network, diskette, CD-ROM, satellite, etc.).
2. The `InstallCop` is invoked upon receiving the application and RCL.
3. The `InstallCop` authenticates the executable and RCL.
4. The `InstallCop` then generates a system default configuration file from the authenticated RCL[2] if the installation is made system wide. Otherwise, a configuration file specific to the installing user will be generated for this application.
5. The `InstallCop` names the configuration file <application name>.cfg.
6. In case of system wide installation (by the superuser), the `InstallCop` places the configuration file in the `/cv/config/sys` directory. Otherwise the file is placed in the `/cv/config/sys/<username>` directory. If there is a name conflict, the installer is prompted by `InstallCop` for a new application name. For instance, if a new version of `emacs` is installed, it can be called `emacsv2` or `emacsnew`. Alternatively, the obsolete version of emacs can be reinstalled as, for example, emacs.old. Note that we do not use the complete path name for identifying an application. By taking this approach, we allow the flexibility for an application

[2]Optionally, the `InstallCop` can allow the system administrator to edit the system default configuration file before the application is made available to users.

to move from one directory to another without having to completely reinstall the application.

7. For system wide installation, the `InstallCop` prompts the installer for an installation directory. The installation directory must be writable only by the superuser (e.g. `/usr/bin` or `/usr/local/bin`, etc). For installation by an user, either the file may already exist in the user specified directory, or it may be copied to the user specified directory at this time.

8. The application is placed in the installation directory. If there is a name conflict, the installer is prompted by the `InstallCop` for a new application name. The config file just created must also be renamed. The configuration file specifies the physical and logical resource requirements as well as the RCLs related to other subsystems. The configuration file is stored in a hidden directory on the system, whose location is known only to the InstallCop and the "sandbox" program. As described earlier, this file is used by the sandbox program to read in the physical and logical resource requirements and limits for this application.

5 Summary and Conclusion

Sharing of unknown programs creates an atmosphere of untrust and hence, exacerbates the need to secure information and physical resources from *alien* codes. Once an alien code is installed on a client machine, it can run with all the privileges of the invoking user, without any notification to the user of the privileges being used. Hence, the alien code can silently obtain unauthorized accesses to all the system resources (e.g. files, network ports) that can be accessed by the invoking user. The alien code may propagate itself, attempt illegal break-ins or maliciously alter files. It may also engage in a *denial of service* attack by monopolizing various *physical resources* (e.g. disk, physical memory, CPU).

A sandbox approach to execution of such foreign codes is proposed in this paper. Any alien code is trusted with a set of privileges for accessing various resources via a third party authentication. The list of resources may include *logical* system resources (e.g., callable functions and services), *physical* resources (e.g., rate and maximum consumption amount of CPU, memory, disk space, etc.) as well as various resources defined or owned by other subsystems (e.g., database relations). During installation of an alien code in the sandbox environment, the associated privileges (per user per code basis) are stored in a secure area and enforced by the OS and various subsystem monitors during execution.

A prototype version based on Linux OS code has been developed.

Acknowledgments: The authors would like to thank David Dion of the University of Washington, Seattle for participating in the discussion of user environment issues, and in the development of prototype based on Linux. We would also like to thank David Safford for his comments on the earlier revisions of the paper.

References

[1] Anderson, D. P., “Metascheduling for Continuous Media,” *ACM Transactions on Computer Systems*, Vol. 11, No. 3, August 1993, pp. 226–252.

[2] Benantar, M., R. Guski, K. M. Troidle, “Access Control Systems: From Host-centric to Network-centric Computing”, *IBM Systems Journal*, Vol. 35, No. 1, 1996, pp. 94–112.

[3] Dion, D., A. Dan, A. Mohindra, D. Sitaram, “*ChakraVyuha:* Design and Implementation ”, *IBM Research Report*, (in preparation).

[4] Goldberg, I., Wagner, D., Thomas, R., and Brewer, A. B., “A Secure Environment for untrusted helper applications,” *To appear in Proceedings of the 6th USENIX UNIX Security Symposium*.

[5] van Hoff A., S. Shaio and O. Starbuck (eds),“Hooked on Java”, Addison-Wesley, 1996.

[6] Jaeger, T., Rubin, A. D., and Prakash, A., “Building Systems That Flexibly Control Downloadable Executable Content,” *To appear in Proceedings of the 6th USENIX UNIX Security Symposium*.

[7] Key Logic., “KeyKOS Architecture,” *http://www.agorics.com/agorics/KeyKos/architecture/Welcome.html*.

[8] Lampson, B. W., “Protection”, *Proc. of the fifth Princeton Symposium on Information Sciences and Systems*, 1971.

[9] Ramakrishnan, K. K., L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaser, W. Duso “Operating System Support for a Video-On-Demand File Service.” *Multimedia Systems*, Vol. 3 No. 2, May 1995, pp.53–65.

[10] Rubin, Aviel. D., “Trusted Distributed of Software Over the Internet,” *Symposium on Network and Distributed System Security*, February 1995, pp. 4 7–53.

[11] Steiner, J. G., Neuman, C., and Schiller, J. I., “Kerberos: An authentication service for open network systems,” *USENIX Winter Conference Proceedings*, February 1988.

[12] Walker, K. M., Sterne, D. F., Badger, M. L., Petkac, M. J., Sherman, D. L., Oostendorp, K. A., “Confining Root Programs with Domain and Type Enforcement,” *6th USENIX Security Symposium*, July 1996.