2
2005
63681459

This is to certify that the
thesis entitled

A CLASSIFICATION SCHEME FOR SANDBOXES

presented by

ARUN PRABAKARAN

has been accepted towards fulfillment
of the requirements for the

Master of Science	degree in	Computer Science and Engineering

[illegible]
Major Professor's Signature

3/24/05
Date

A CLASSIFICATION SCHEME FOR SANDBOXES

By

Arun Prabakaran

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2005

ABSTRACT

A Classification Scheme for Sandboxes

By

Arun Prabakaran

In recent time there has been a marked increase in security flaws due to the increase in the use of untrusted applications. In some instances trusted application that have been compromised through common vulnerabilities such as buffer overflow attacks also result in a breach of security. On of the more prominent themes of security software today is the idea of containing applications through building isolated execution environments or sandboxes for them to operate in, thus limiting the damage they can do to the system. Several such independent tools have been developed. We have in this thesis presented a means to classify these tools, based on a basic set of primitives that we have identified to be central to the idea of sandboxing. We also present two broad categories within which implementations of sandboxes could fit. The classification scheme is visually represented assisting future designers of sandboxes in translating security objectives and policies into a sandbox design. The classification can help translate the design into an implementation. In order to demonstrate the effectiveness of the classification scheme we present a comparative analysis of the five prominent sandboxes that have been proposed or implemented.

Acknowledgement

I would like to thank my advisor Dr. Richard J. Enbody for his guidance and support throughout the duration of my thesis. The discussions in his research group formed the basis for my interest in security. His comments for the numerous presentations that I made with the research group were valuable in shaping the direction of my thesis. I would like to express my sincere gratitude to him for the time and effort that he spent in critically analyzing my thesis during the several iterations of review.

I would like to thank my guidance committee members Dr. Sandeep S. Kulkarni, and Dr. Anthony Wojcik for their time in reviewing my thesis. I would also like to thank my research group mates Krerk Pirompsompa, and Sohrab Soltani for their comments during the numerous discussions we had during the weekly research group meetings.

I would also like to thank all my other friends at MSU who helped make this thesis a possibility.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

In an increasingly connected world, mobile code plays an important role in networked systems providing as many services as possible. As more and more services go online, our dependence on third-party code increases. The introduction of large amounts of, potentially untrusted, third-party code also has the undesirable potential of compromising system security.

Some of the most common security flaws are based on buffer overflows. A resulting escalation of privileges results in compromising the system. An attack could compromise any aspect of the system such as memory utilization, processor or CPU utilization, access to secure information, or interference with the successful execution of other processes. The list of malicious processes has only grown, with varied software such as viruses boot sector viruses, executable infectors, multipartite viruses and Trojan horses, among others.

Several solutions have been put forward, including but not limited to, access control lists, intrusion detection systems, authentication protocols, proof carrying code, certificates, virtual machines, and network security tools. All these systems have individual strengths, and weaknesses. A conceptually simple approach to solve security problems caused by malicious third party code is to contain them, within a limited execution environment. The approach has been called sandboxing, a term we will define shortly, and came into popular usage with systems such as the Java Virtual Machine. As we will shortly see, several other sandboxes were also developed, mostly independent of each other.

However, there is no cohesive method to compare different sandboxes, and to see how one can build on existing implementations. The concept of Sandboxing has been defined differently in different contexts with different objectives.

In this thesis, we present core set of primitives that will define a sandbox. We provide an analysis of the different values that the primitives can take, and have imposed an ordering on the different possible combinations or values of the primitives.

In order to analyze the different implementations of the sandbox, we also present two possible design paradigms for designing sandboxes. While the paradigms could also be considered a part of the primitives, they deserve special mention as they relate to the implementation issues for a sandbox. Identifying these implementation paradigms can influence design choices.

In order to validate our classification we will compare two closely related sandboxes, JVM1 and JVM2, and we will present an analysis of how our classification system differentiates them. We will also use a criteria, based on the number of points of difference between sandboxes for each primitives, to present a visual representation of the difference between the sandboxes.

1.1 Introduction to Sandboxes

A sandbox, as a security idea originated from the Java execution environment [41] , which defined a sandbox as *"a set of rules that are created when creating an applet that ensures that the applet executes within a limited environment, when the applet is sent as a part of the web browser".*

The origins of sandbox as a containment idea can be traced to the study of fault tolerant systems, wherein sandboxes were defined as "*an execution domain, involving the code and the data segment, isolating the faults within this segment*" [53]. While these systems categorize security flaws as faults, it is a generalization that encompasses more than just security, thus deflecting much needed focus on security problems. While the generalization is useful in applying innovations in the field of fault tolerance, in a generic sense to security principles, it does not address the immediate concerns of providing safe execution domains.

Some of the important features that a sandbox will have to address in its design parameters, will include, the impact of compromise (vulnerability), threat perception or avoidance mechanism, and a system clean up facility for post detection phases. The impact of compromise will include the identification of the effects of compromising the application. For instance, compromising system applications can result in more damage than compromising user level processes. Threat perception or avoidance includes a system lookout for aberrant behavior, resulting in avoiding possible attacks. Such an objective can be achieved by restricting resources allocated to the malicious process. A system cleanup will take counter measures to clean the system after any break ins have been detected. Typically a cleanup will perform reporting operations. Based on the issues discussed above we arrive at our definition of a sandbox; "*We define the sandbox as an isolated execution environment, for one or more trusted or untrusted applications, with specific restrictions placed on resource usage and communication outside the execution environment*".

1.2 Visualization of results

In this thesis we will present a visualization scheme to readily identify properties of a sandbox, which we believe is useful in comparing sandboxes. Here we briefly introduce the visualization scheme to indicate where we are headed. In figure 1, every vertex corresponds to a primitive, which will be defined in chapter 2, and a particular sandbox occupies a certain block within the polygon defined by the primitives. In figure 1, we show an "average" representation of a sandbox, based on our classification of five major sandboxes. The solid block represents the extent of each feature that is present in the sandbox in relation to the total possible points of difference with other sandboxes. Figure 2 show the different sandboxes that we have compared. The occlusion of polygons will help illustrate the different capabilities a sandbox has compared to others. We will present more about the interpretation of such diagrams in Chapter 4.

1.3 Organization of thesis

The organization of this thesis is as follows. In Chapter 1, we will provide a brief introduction to the idea of sandboxes, including a definition that we will use. In chapter 2 we will present our primitives, along with a brief description, and the various values that the primitives take. In chapter 3 we will present the design paradigms for sandboxes, and explain the same with respect to an imaginary sandbox. In chapter 4 we will present an analysis of the classification scheme with respect to JVM1 and JVM2. We will also present the visual representation of the different sandboxes in chapter 6.

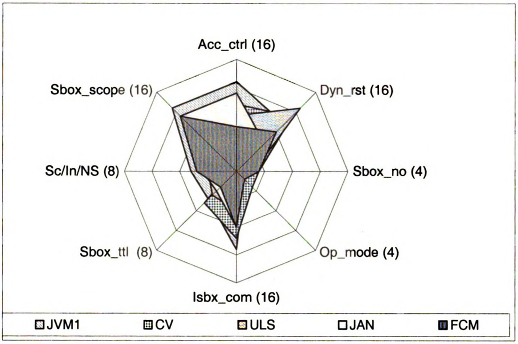

Figure 1.1: Occlusion Diagram

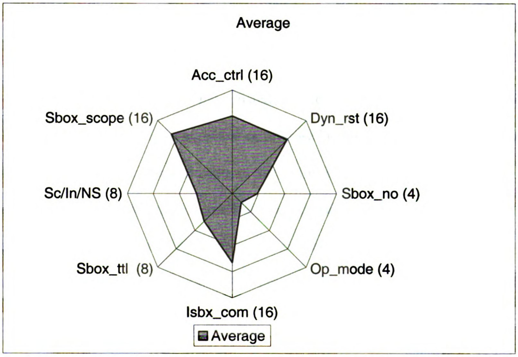

Figure 1.2: Average case sandbox

2. Primitives for classification

2.1 Introduction

The fundamental purpose of this chapter is to present the important primitives that we have identified for a sandbox. A variety of sandboxes have come up in recent times. Some of them are described in [16, 23, 34, 37, 41, 42]. Based on our examination of the various sandboxes we have formulated a set of primitives. While not all sandboxes will implement these primitives, it will be a good exercise for the designer of a sandbox to consider these alternatives before an implementation. Our expectation is that the identification of primitives will help reduce the number of alternative security measures that need to be evaluated. Identification of primitives will also help in easier translation of policy into action, as the set of primitives will help in clearly outlining the possible security scenarios the sandbox will be able to handle. Above all, the listing of a core set of primitives will help enhance our understanding of what is expected out of a sandbox.

In the sections below we have listed different primitives, and have presented alternative security solutions available or proposed in the literature. We also provide a brief justification for the inclusion of the primitives in our set. Along with each primitive, we have imposed a numerical value to indicate its relative importance with respect to other primitives. Assigning numeric values and the resulting ordering will help us during the visualization of the primitives. The method adopted to give the relative importance to the different sandboxes is a simple test of which parameter is likely to provide more security with respect to the other features. We have listed all combinations that appear in our classification scheme, in addition to prominent features. Since the values we impose are relative, they are not continuous.

2.2 Access Control *(Acc_ctrl):*

Acc_cntrl deals with the protection of the current state of the sandbox. Some of the most important features of access control include safeguarding confidential information and critical processes, provision of a flexible access control mechanism, and enforcing appropriate safety control, including password quality [17]. Most operating systems have been using the access control model. A sandbox will try and enforce some access control measures, by directly translating the security policy into an implementation. A popular implementation model is the access control matrix, which is a simple representation of user permissions for the different files. Such a technique is particularly useful in UNIX based operating systems where entities are treated as files. One example of an access control matrix is shown in figure 2.1.

Process ↓	**Files →**	***File 1***	***File 2***
Process 1		Read, write, open	Read
Process 2		Append	read

Figure 2.1 : Access control matrix

Acc_cntrl can also be represented through Boolean expression evaluation. Such ideas have been explored further in [19]. Such methods when used in a sandbox are particularly useful in complex systems that require dynamic policy implementations.

An access restriction facility can also be incorporated with the sandbox. One more technique that can be used is access control through the use of a process history, which could be particularly useful for unrecognized code. So, even if a malicious process gains access to critical system resources, based on the typical usage patterns embedded in the controlling process, we can limit the damages to the system. We will also differentiate between thread-level Sandboxing and process level Sandboxing.

Some operating systems use *capabilities* instead of access control. However *capabilities* can be considered to be a part of an access control matrix [45]. *Capabilities* are access control matrix tied to an object based system. Thus whether or not an object can access a resource depends on whether the object has appropriate *capabilities*. One could classify *capabilities* separately under our *Acc_cntrl* primitive. The reason for combining *capability* with *Acc_cntrl* is that a capability is still a means for restricting access to a process, and from a sandbox standpoint, it is necessary.

Some of the entities involved in the *Acc_cntrl* are given in Table 2.1. Clearly some of the combinations are not in use, for example UA/UTthread. The numerical values are (imposed) based on the relative strengths of the different primitive values. For example, TApp represents a trusted application and therefore is given a value of 4 over UTApp which represents an untrusted application. Similarly, we have imposed a hierarchy between applications, threads, and processes. Constrained access (CA) is given a higher score (4) compared to unconstrained access (UA), which is given 0. The relative values imposed of other features can be observed from the table.

Table 2.1: Summary of values for *Acc_cntrl*

Acc_cntrl Values / combinations	**Numerical values**	**Description**
TApp	4	Trusted Application
UTApp	2	Untrusted Application
Tthread	4	Trusted Thread
UTthread	2	Untrusted Thread
Uproc	4	Trusted Process
UTproc	2	Untrusted Process
UA	0	Unconstrained Access
CA	4	Constrained Access
UA/TApp	4	Unconstrained Access for trusted Applications
CA/TApp	8	Constrained Access for trusted Applications
UA/UTApp	2	Unconstrained Access for untrusted Applications
CA/UTApp	6	Constrained Access for untrusted Applications
UA/ Tthread	4	Unconstrained Access for trusted Thread
CA/Tthread	8	Constrained Access for trusted Thread
UA/UTthread	2	Unconstrained Access for untrusted Thread
CA/UTthread	6	Constrained Access for untrusted Thread
UA/Uproc	4	Unconstrained Access for untrusted Process
CA/UTproc	6	Constrained Access for untrusted Process
PBA/ Tthread PBA/ UTthread	10	Different policies for trusted and untrusted threads are used
PBA/ Uproc PBA/ UTproc	12	Different policies for trusted and untrusted processes are used
PBA	16	Policy Based Access

2. 3 Dynamic Policy Enforcement *(Dyn_rst):*

Policy enforcement can be dynamic or static. What we mean by dynamic enforcement is that the administrator can change the policy on the run with root access to the sandbox control components. Static enforcement, means that the configuration cannot be changed at runtime; every time a policy is changed the sandbox components need to be reconfigured. Dynamic configuration has both advantages and disadvantages. The advantage is that policy changes can be incorporated dynamically and according to sandbox needs. However, the disadvantage is that a faulty change to the policy can be dangerous, and can potentially compromise system security. Safety can be ensured by a combination of static and dynamic security specifications. For instance, it may be possible to specify a core set of policy details which cannot be compromised. Also, a major objective for management of policy with a static system is that new, and possibly malicious code should not be able to break the system. One example where malicious code uses scripts is that, under the guise of updating the system sandbox components actually re-configure to unsafe states. A breach in the sandbox may trigger a domino effect and result in the entire system in an unsafe state.

The disadvantage of a static policy enforcement mechanism is that it will be a tedious process to regularly update the sandbox on a constant basis, and consequently keeping up with current developments, perhaps, a new virus or worm, or any buffer overrun attack techniques cannot be accounted for on a continuous basis. This inflexibility may in turn render the system useless from a security standpoint for that period of time.

The advantage with a dynamic policy enforcement mechanism is that, it offsets the exact problems that the other systems with a static enforcement face, that is the ability to constantly update a given sandbox mechanism with the latest developments. We advocate such a technique for any implementation since the security landscape is constantly changing.

Other issues with respect to policy enforcement include when the necessary changes made to the system actually take effect. It could be either immediately or after a specified time period. Issues such as, what parameters are chosen to decide on the time period are also necessary. Table 2.3 summarizes the fundamental values *Dyn_rst* takes.

Table 2.2 : Summary of values for *Dyn_rst*

Dyn_rst values	**Numeric Values**	**Description**
S	2	Static Policy specification
D	4	Dynamic Policy specification
H	4	High security indicating higher constraints
M	3	Moderate security indicating potential flaws
L	0	No security measures available
TApp	4	Trusted Application
UTApp	1	Untrusted Application
Tthread	4	Trusted Thread
UTthread	2	Untrusted Thread
Uproc	4	Trusted Process
UTproc	2	Untrusted Process

Typically, our classification scheme uses a combination of one or more of the values of table 2.2. Table 2.3 describes some of the most common combinations. A Combination represents the simultaneous use of more than one value presented in

Table 2.3: Summary of *Dyn_rst* combinations

Combinations	**Numeric Values**	**Description**
SM/ TApp	9	Static policy specification for TApp
DM/TApp	11	Dynamic policy specification for TApp
SM/UTApp	6	Static policy specification for UTApp
DM/UTApp	8	Dynamic policy specification for UTApp
SL/TApp, DL/TApp	14	Other possible combinations using the nomenclature presented above

Table 2.2. While all combinations of the parameters presented in the table are used in our comparisons, other combinations are also possible.

2.4 Sandbox Number (Sbox_no):

The number of sandboxes is also an important issue when it comes to designing a sandbox. Several scenarios arise. We could have a sandboxing mechanism which does not allow the possibility of duplicating containment environments. Another issue that crops up, as a consequence of considering the number of sandboxes is sharing of privileges. For example, when a process uses a fork () to spawn another process, several issues need to be considered. These include, whether or not the forked process executes within the sandbox, or whether we create a new sandbox for the forked process. In case of the creation of a new sandbox, privileges and methods used to assign or allocate the privileges for the new sandbox need to be considered. One technique is the principle of attenuation of privileges; wherein the new sandbox will get no more privileges than the parent sandbox that spawns it. The advantage of using such a rule is that a malicious process cannot spawn a container with higher privileges. The implicit assumption here is that malicious processes will not have higher privileges to begin with.

One reason why we need to consider the number of processes executing within a specific container is that, in many scenarios, the container, apart from being a security environment, is also a method for guaranteeing the processes a certain amount of execution space, which includes, such things as processing power, memory and network bandwidth. Thus, infiltration of a sandbox by malicious processes is enough to sabotage the execution of all processes in that sandbox.

One of the practical considerations of a possible implementation of a sandbox include the inheritance of file descriptors from parent to child to retain control in a tree like structure. We can also allow for system calls for the child processes to further enhance restrictions, or to request the parent sandbox for more relaxations of security criteria. However, we can allow the parent to retain full control over the sandbox containing it's child processes. Also, as a general principle, sandboxes cease to exist as soon that the process that was contained in them stops executing. The same is true for sandboxes that contain the child processes, as against the original or root or parent sandboxes.

In our classification of different sandboxes, and in our comparative study presented in chapter 5, we take into consideration the number of sandboxes that can execute in the environment. We also consider, in our classification scheme, the total number of processes that can execute inside of a sandbox at any one time.

There are only three values assigned to *Sbox_no*. These are specifically, Static Policy Based (S/PB), Dynamic/Policy Based (D/PB), and Static/Fixed Integer based (S/Int). A D/PB based sandbox will exhibit the highest level of security, since individual processes can be sandboxed separately. S/PB follows D/PB, because once the static *Sbox_no* limit is reached, there will be no way to sandbox newer processes. S/Int is assigned the lowest value (2) since policy is not taken into account, unlike S/PB where a robust policy can set a realistic limit on *Sbox_no*. Table 2.4 summarizes the values for *Sbox_no*.

Table 2.4: Summary of Sbox_no values

Sbox_no	**Numeric Values**	**Description**
S/PB	3	Static/Policy based
D/PB	4	Dynamic/Policy based
S/Int	2	Static/Fixed integer

2.5 Operation Mode *(Op_mode):*

The two values for *Op_mode* which a sandbox can take are the kernel level (KL) and the user level (UL). Other values are a combination of those two modes. While specifying *Op_mode* we also take account of the entity that is being sandboxed. We have identified two entities: user applications (App) and the sandbox component itself (SboxC). Both SboxC and App are provided a common numerical value of 2.

A sandbox in KL *Op_mode* has the advantage of being able to sandbox important components of the operating system, such as the object manager, cache manager, the device drivers, the network drivers and the system executables such as the memory manager. Such a sandbox would be particularly useful in an environment that uses a lot of third party, untrusted code at the kernel level. Sandboxing the kernel level application will be useful in open source environments where, untrusted code could be patched with the kernel.

However, the power of such a sandbox also comes with certain disadvantages. The sandbox must be robust. If it is flawed, then we are adding faulty or potentially malicious code into the operating system ourselves, which needs to be guarded against, and care taken to ensure the safety of the sandbox that may be installed at the kernel level. In order to ensure a safe kernel, the "keep it simple" principle must be followed wherein we adopt a design philosophy of using only the bare minimum features needed to contain the process from doing excessive damage to the system. The other functions can be delegated to the user level. Another disadvantage of executing the process at the kernel level will be the need for constant context switching for several monitoring activities of the sandbox.

Another value of *Op_mode* is the user level (UL). The primary disadvantage of a UL based sandbox will be it's inability to police insecure processes at the kernel level. A flaw in a critical kernel component will easily break security measures provided by a sandbox operating in the UL *Op_mode*. A simple buffer attack providing escalation of privileges to the root level to a malicious process is enough to sabotage the execution of a sandbox operating in UL *Op_mode*. However, that does not render the UL sandbox entirely ineffectual. Some of the most important uses of a user level sandbox will be in the form of a container for independent subsystems within such as an executing environment for a programming language or a virtual machine.

A typical setup will be for a system to have a single master sandbox at the KL and then several of the same type in the UL. Such a setup will provide fine grained controlled over the various UL-App and also provide assurance for the safety of the kernel level process.

Table 2.5: Summary of Op_mode values

Op_mode	**Numeric Values**	**Description**
SboxC	2	Sandbox Components
App	2	Application
KL	2	Kernel Level
UL	0	User Level

A summary of the values that the *Op_mode* primitive can take is given in the table below. The primitives can be a combination of the different values given in the table 2.5

2.6 Inter-sandbox communication *(Isbx_com):*

Isbx_com addresses the inter sandbox communication mechanisms. Processes do not function in isolation. There are different methods in which the processes can communicate [21]. Some of the most important inter process communication (IPC) techniques include the use of shared memory, pipes, messages queues, and remote procedure calls (RPC). However, a process placed within a sandbox will have communication restrictions placed on it. Without communication, one could have a perfectly secure, but dysfunctional sandbox. The issue of how to allow processes contained within a sandbox to communicate and yet ensure security is fundamental to the design of any sandbox. *Isbx_com* addresses this concern.

One way for communication to take place between sandboxes is to encapsulate the communication stream in the form of a secure communication path. For example, to communicate between processes A and process B, we need to have two points of contact with a safe stream. The process handling the safe stream can then in turn be encapsulated within a simple containment environment, such as the sandbox itself. A simple illustration is shown in Figure 2.2.

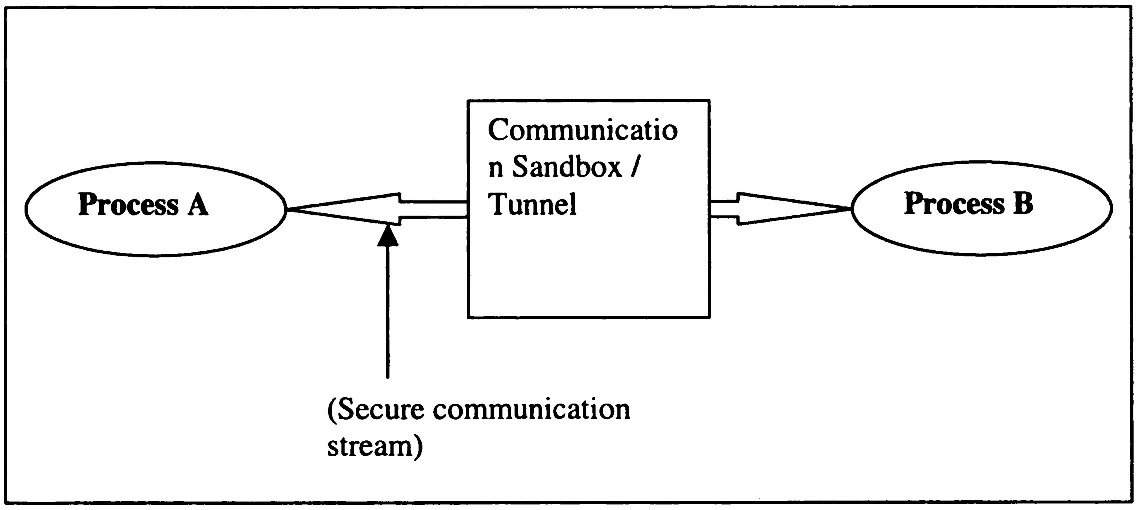

Figure 2.2: *Isbx_com* through secure stream (S/ipc, S/pipes, and S/mques)

The actual implementation of the communication stream could take several forms, including pipes (S/pipes), message queues (S/mques), or secure remote procedure calls (S/ipc). In addition to the regular implementations, they will pass through a sandbox, as shown in Figure 2.2, to ensure safety. The communication sandbox itself will be defined by communication protocols that may be built into the sandbox or specified by means of a policy. An important advantage that the secure communication stream based model offers is that we are not actually altering the communication mechanism provided by the system of the platform. We are merely using the existing platform in order to create a more secure communication environment.

A problem arises when more than two process need to communicate, such as in group communication scenarios. While we do not go into the specifics of the scenario, a broad generalization could be a communication system where all processes connect to a common mailbox (S/mbox) and then use a routing technique (not unlike network routing techniques) to connect to other processes. The S/mbox method for *Isbx_com* apart from providing scalability to a large process group will also provide a central point of communication. On the other hand, there are disadvantages to having a central point of communication, including a single point of breakdown. However, such a *Isbx_com* mechanism will be definitely useful in a sandbox with a high probability of multiple cooperating processes. A simple illustration outlining our idea is given in Figure 2.3.

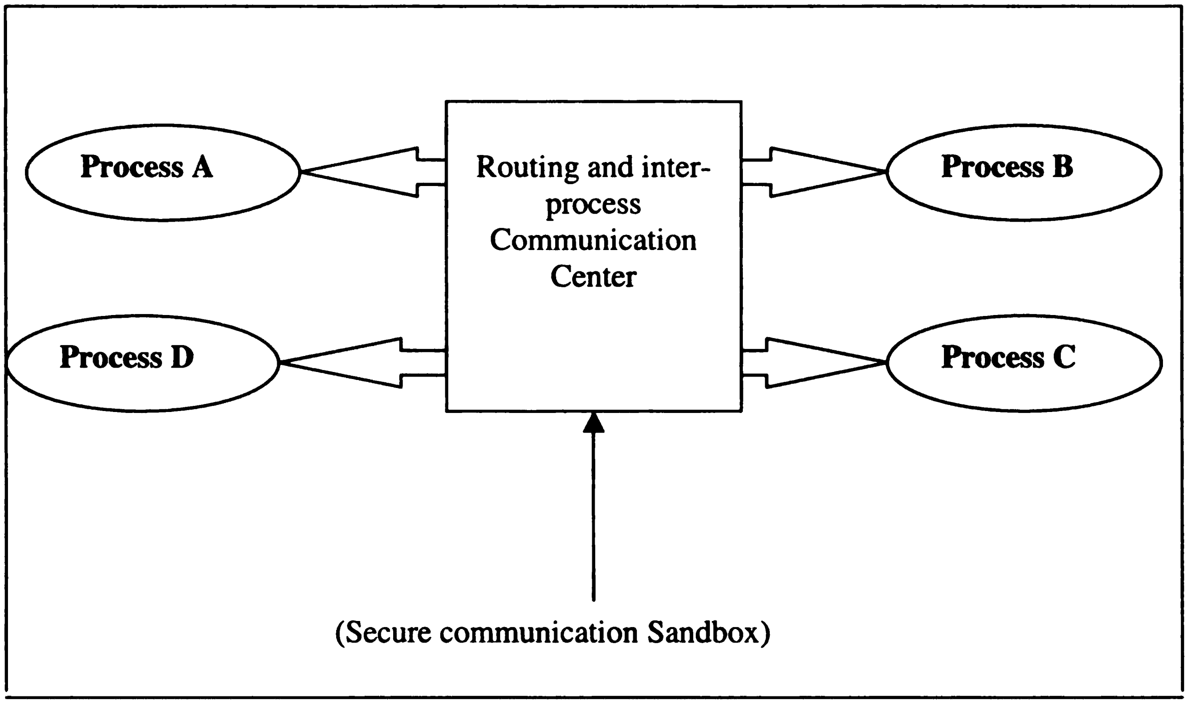
Process A
Routing and inter-process Communication Center
Process B
Process D
Process C
(Secure communication Sandbox)

Figure 2.3: *Isbx_com* through S/mbox

Another important technique for *Isbx_com* will be tunneling the communication through a KL (defined under *Op_mode)* sandbox. The advantages of such a process will include added security in the form of assurance from the compromised KL-SboxC. To illustrate the difference between a UL (defined under *Op_mode)* secured *Isbx_com* and a KL secured *Isbx_com*, we consider a buffer overflow attack leading to escalation of privileges by a malicious process. In such a scenario, secure *Isbx_com* can be sabotaged by the malicious process with root privileges. However, the KL communication sandbox can protect the system against such attacks.

Policy based communication (P/comm) is another possible value for *Isbx_com.* P/comm represents the communication protocol that is based on policy. In many cases P/comm will be associated with one or more of S/ipc, S/pipes, and S/mques. MISC represents middleware-induced security. Several popular sandboxes use MISC (as seen in Chapter 4, JVM uses MISC).

For relative ordering that we have imposed on the values for *Isbx_com,* S/ipc, S/pipes, and S/mques get the highest value (8). MISC relegates the *Isbx_com* to an entity outside the sandbox, and hence is assigned a lower value (6). S/mbox provides a central point of failure and therefore is assigned a value of 6. Table 2.5 shows the important values for *Isbx_com.*

Table 2.5 : Summary of values for *Isbx_com*

Isbx_com	**Numerical Values**	**Description**
MISC	6	Middleware induced security communication
S/ipc	8	Secure sandbox based Interprocess communication
S/pipes	8	Secure sandbox based pipes
S/mques	8	Secure sandbox based message queues
S/mbox	6	Secure sandbox based mailbox
P/comm.	8	Policy based communication (*Isbx_com)*

2.7 Sandbox Time to Live *(Sbox_ttl)*:

Sbox_ttl, the time that a sandbox exists, is important because unused or redundant sandboxes pose additional management overhead for the controlling components, particularly in the component-based approach which we will discuss in chapter 4. A sandbox can live for just as long as the process that it contains exists or it can exist for a specified time period; for example a fixed number of clock cycles. The length of the lifetime of the sandbox can also be a deciding factor in the complexity of implementation of the sandbox. For example, we can also have an implementation of a sandbox where extensive modifications to the file systems and the various related KL components are needed to support the additional lifetime of the sandbox. KL sandboxing will be a useful idea for important processes that need safe containers to execute, such as virtual memory managers, which are guaranteed to execute at least for a significant portion of the system lifetime. Some disadvantages of such a scheme are that the additional changes to kernel code will have to be well audited. Modifications to KL components could lead to an escalation of costs, either in the way of KL implementation, or through performance overhead due to the cost of additional context switching.

We have defined four values for *Sbox_ttl*, taking into account all the factors from the above discussion. *Sbox_ttl* with a thread lifetime (Thr_lt) is a sandbox that is specific to a thread, and is destroyed when the thread stops executing. *Sbox_ttl* with an application lifetime (App_lt) is a sandbox that is destroyed as soon as the application ceases to exist. *Sbox_ttl* with a process lifetime (Prcs_lt) is a sandbox that is specific to a process and is destroyed as soon as the process stops executing. The longest *Sbox_ttl* occurs when the sandbox exists for the duration of the execution environment (Sys_upt), which usually is the operating system.

We impose the relative ordering as follows. Thr_lt is given the lowest score 4, since providing independent threads with sandboxes will introduce additional overhead. App_lt is given a score 6 higher than Thr_lt, because we expect the number of applications to be lesser than the number of threads, thus reducing the overhead. Prcs_lt is provided a score 7, higher than App_lt because, when *Sbox_ttl* is Prcs_lt, the sandbox ceases to exist as soon as the process (that was sandboxed) terminates, presenting a lower possibility for a malicious process to misuse a ghost sandbox (one that exists even after the process that it contained has stopped executing). Sys_upt is provided a higher score of 8, contrary to a lower score that we will expect based on our experience with Thr_lt, App_lt, and Prcs_lt. The reason for the high score is that sandboxes with a Sys_upt (like Chakravyuha, described in Chapter 6) will coexist with a lower *Sbox_no*. The combination of a lower *Sbox_no* and Sys_upt provides manageable overhead, keeping the sandbox design simple. *Sbox_ttl* is summarized by the Table 2.6.

Table 2.6: Summary of values for *Sbox_ttl*

Sbox_ttl	**Numerical Values**	**Description**
Thr_lt	4	Thread Lifetime
App_lt	6	Application Lifetime
Prcs_lt	7	Process Lifetime
Sys_upt	8	System (execution environment) Uptime

2.8 Sandbox Scope *(Sbox_scope):*

Sbox_scope refers to the span of the sandbox, i.e., what entities (processes, applications, threads, or the entire operating environment) can be contained by the sandbox.

There are four different values that have been assigned to *Sbox_scope*. *Sbox_scope* takes the value global (G) when all entities in the operating environment can be contained by the sandbox. *Sbox_scope* takes the value local (L) when one or more entities (processes, applications, and threads) can be ruled out of being contained by the sandbox.

The two values G and L are assigned with a time frame during which the *Sbox_scope* can be global or local. Persistent (Prst) value for sandboxes is assigned when the sandbox has the ability to contain all entities (processes, applications, and threads) all the time. Otherwise we assign Transient (Trnst) for *Sbox_scope.*

We impose a numerical ordering on these values. When *Sbox_scope* takes the values G, more processes can be contained, and hence more security 8, while we assign a slightly lower value 6 to L. The duration for which *Sbox_com* is G or L is of lower significance, and therefore we assign a maximum value of 4 to Prst, and a value of 2 to Trnst.

Sbox_scope is different from *Sbox_ttl.* By Prst and Trnst, we refer only to the duration of value G or L for *Sbox_scope.* On the contrary, by *Sbox_ttl,* we mean the lifetime of the sandbox. Table 2.7 summarizes *Sbox_scope* values.

Table 2.7: Summary of value for *Sbox_scope*

Sbox_scope Values	**Numerical Values**	**Description**
G	8	Global/applied across all entities
L	6	Local/applied to specific entities
Prst	4	G or L value is persistent
Trnst	2	G or L value is transient

2.9 Scalability *(Scl):*

Since we also intend the classification effort to form a design manual for designers for future sandboxes, we evaluate sandboxes on their scalability (*Scl*). *Scl* refers to the ability of the sandbox to balance security with an increasing number of entities (processes, applications, and threads).

One instance where *Scl* is important is as follows. Some sandboxes need a healthy ratio of used/available heap space to function efficiently. Heap space could be a serious limitation for newer sandboxes to be created, when the number of processes increases. Shortage of heap space could in turn result in a forcible change of sandbox behavior leading to an insecure state.

From a security standpoint scalability is not a core issue, since it comes into picture only for a subset of the sandboxes and only under special cases. *Scl* can take the value Y (when the sandbox is scalable) and N (when the sandbox is not scalable). Table 2.8 summarizes the values for *Scl.*

Table 2.8 : Summary of values for *Scl*

Scl	**Numerical Values**	**Description**
Y	2	Yes
N	0	No

2.10 Interoperability (*Int_op*):

The interoperability (*Int_op)* of the sandbox includes the interoperability with other sandboxes of the same type installed in different installations applicable specifically to Chakravyuha, described in chapter 6. *Int_op* is also important in the case where we will limit the values that *Int_op* can take an yes (Y) or no (N). Table 2.9 summarizes the value for *Int_op.*

Table 2.9 : Summary of the values for *Int_op.*

Int_op	**Numerical Values**	**Description**
Y	2	Yes
N	0	No

2.11 Native security services utilization *(Nsss):*

A sandbox could be utilizing the security services that are already present in the system. Native security services can include such components as Stackguard [44], and authentication components. We will restrict the values permissible for *Nsss* to be either a a yes or a no. The values for *Nsss* are summarized in table 2.10.

Table 2.10: Summary of values for *Nsss*.

Nsss	Numerical Values	Description
Y	2	Yes
N	0	No

2.12 Duplicating Containment Environment *(Dcenv):*

Dcenv refers to an issue that we need to address in the case of a process that is contained within a sandbox that could spawn *(fork)* another process. In this case, two possibilities arise; either the new process exists within the same container or in a new container. *Dcenv* is kept relatively simple by following a static policy implementation. If the sandbox allows a high degree of *Dyn_rst*, then *Dcenv* could form a protection mechanism against compromising the core of the sandbox. While more complicated methods for duplicating containers can be formed, we will restrict our value to either yes, (Y) or no (N). However we do not apply *Dcenv* to the existing sandboxes that have been considered for classification in this thesis. The feature is provided for extensibility of the classification scheme. Table 2.11 summarizes the values for *Dcenv*.

Table 2.11: Summary of values for *Dcenv*.

Dcenv	Numerical Values	Description
Y	2	Yes
N	0	No

2.15 Conclusion

In this chapter we introduced the idea of our primitives for Sandboxing. The primitives themselves are borrowed from different aspects of security. What is unique here is that we have identified the most (as we will see in the next chapter) appropriate primitives to classify the different sandboxes. The primitives will also serve as a means to serve as a design manual for designers of newer sandboxes. The classification scheme itself is an effort in that direction. The classification scheme assumes special significance in design scenarios, where we want to relate design, policy, and the level of security that can be attained with minimal redundancy. Elimination of redundancy is particularly useful because, several security features with overlapping responsibilities are often a source of redundancy resulting in lower system performance. In chapter 5 we will see how the various primitives interact and the effect of policy. While the study of policy itself is outside the purview of our work, we report some preliminary work in that direction, aimed at providing designers a link between sandbox design and policy. Policy is particularly important since it forms the core of a sandbox in many cases, as we will see in Chapter 4. In chapter 5 we will also discuss if any of the primitives are necessary, and if any are dispensable. It will give us a better perspective, since we do not need that information for the classification process itself, but will greatly enhance our interpretation of the classification scheme.

3. Sandbox Design Categories

In this chapter we try to answer the following questions. What are the ways in which the primitives presented in Chapter 2 can be implemented? Is it possible to identify categories from among the different designs of sandboxes with respect to implementation? If so, how many different categories? What are the pros and cons of each model?

The organization of this chapter is as follows. We introduce the sandbox design categories, and we present the two categories with a brief description of each. We then discuss the importance of the categorization process, and specify its relationship to the primitives. Finally, we illustrate how categorization works with respect to a specific example.

3.1 Sandbox design categories - Introduction

Based on our analysis of different implementations of sandboxes we have identified two categories of design; component based, and layered (also called incremental). In this section we will describe both the categories.

3.1.1 Component based design

As defined by D'Souza et al. [2], component based software architecture "*is an approach to software development in which all artifacts – from executable code to interface specifications, architectures, and business models; and scaling from complete applications and systems down to small parts – can be built by assembling, adapting, and wiring together existing components into a variety of configurations*".

Using the above definition we can infer that a component (we use component in the context of a sandbox) can be defined using the following characteristics [1]: Components are either de-coupled or loosely coupled, and development and testing of each of these components can be done independently. Component both provide well specified interfaces and services. Components can be modified dynamically based on the policies since they are designed independent of each other.

One of the advantages of a component-based design is that it provides a way to map abstract policy statements to components and their interfaces. The design provides a way to protect components from malicious attacks launched on other components.

Some of the disadvantages of a component-based design include a complicated inter-component *Acc_cntrl* (Primitive defined in Chapter 2) and tough design decisions in terms of division of responsibilities between components.

Practical examples in the component-based category include the Trusted Computing Base [3], where the ideas have been proposed in the following ways. (TCPA based systems, though not numerous, could become pivotal if the related initiatives [48], are carried out to completion.) Some clarifications are needed however. A trusted computing base need not be provided to enforce process containment. The different components need not involve the use of a specific API as is the case with trusted computing base-type systems.

A component-based design with six components is shown in Figure 3.1. One drawback with this design shown here is the central point of control, at component 4.

Components 1,2 and 3 operate in the user level while components 5 and 6 are kernel components. Component 4 could be either user or kernel level.

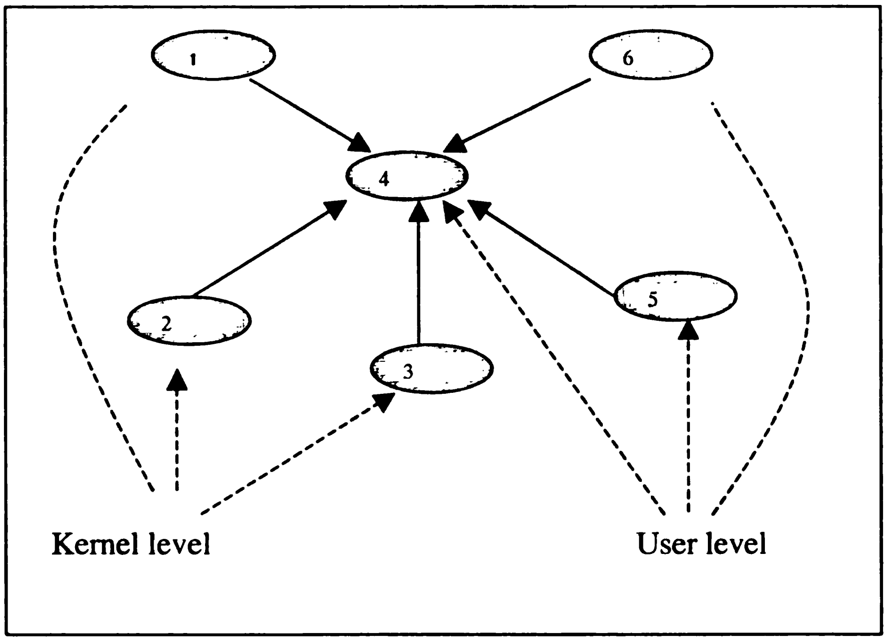

Figure 3.1: Category 1 -A component based sandbox

3.1.2 Layered (Incremental) design

Unlike a component-based design, in a layered design, we rely on an incremental approach. One layer provides services to another layer while using services from another layer. The concept is illustrated in Figure 3.2. The incremental approach can be compared to the different layers of a network protocol model such as TCP/IP or OSI. The same design considerations that hold for the need for layers in a network protocol hold for a layered sandbox design.

The advantages with layered design include the ability to add new services at every layer, without having to modify the interface between the layers, since the interfaces are standardized. The relationship between different layers is thus greatly simplified, since we only have to be concerned with the interface between two adjacent layers.

A major disadvantage of the layered design is that it is more rigid, and the absence of any one layer can cause the sandbox to be dysfunctional.

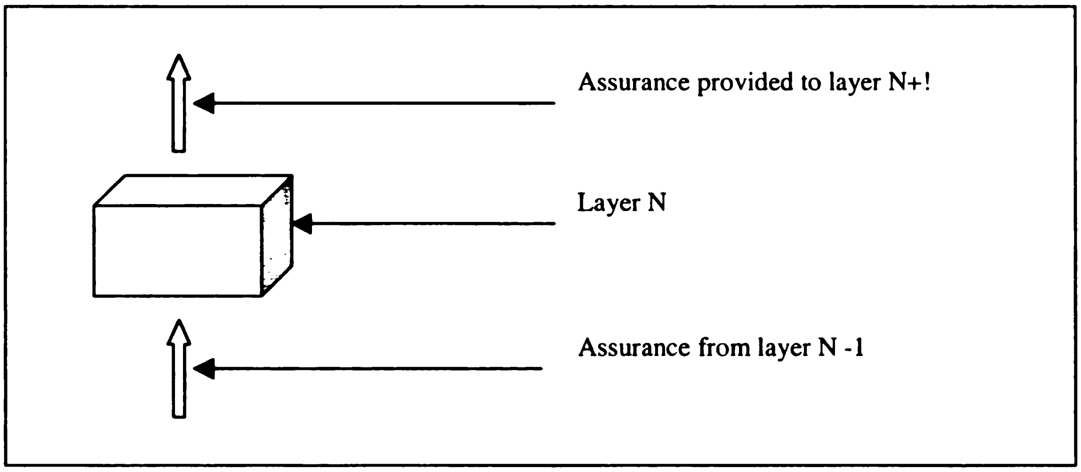

Figure 3.2: Category 2 – A Layered sandbox design

3.2 Significance of the categorization

The primitives identified in the previous Chapter 2 are not useful in isolation when designing sandboxes. While they help in identifying important functionalities of a sandbox, to be an effective security tool they do not help us understand the implementation aspects of sandboxes. The performance of a sandbox is ultimately dependant on the implementation. Hence we have given importance to the different categories of sandboxes.

Most sandboxes, with the exception of JVM, appear to have taken an action-reaction approach wherein the idea of containment was applied as an afterthought to mitigate specific security problems. We have identified categories to provide this insight into sandbox design, at the time of policy specification itself. The categories were a result of an iterative analysis of the sandboxes.

3.3 Relationship to the primitives

The primitives identified in Chapter 2 can be applied universally, and can be used to design sandboxes in either category. We do not identify any primitive specific to a category. However, it is possible to associate a certain primitive more with a category. We can illustrate this difference with respect to *Sbox_ttl*. In the component based category, we can have different components having different *Sbox_ttl* values, since the components (with the exception of critical components) are independent of each other. In the case of a layered approach we find that all layers have to share the same *Sbox_ttl*. Application of different values could result a certain layer being unavailable while other layers continue to be active.

While classifying the sandboxes in Chapter 5 we will specify which category the sandboxes come under. We will illustrate the two layers with a specific example in Section 3.4.

3.4 An illustration with Windows NT

We present a case study of the two categories with respect to Windows NT. We have chosen Windows NT because it falls right in the middle of the categories and will help our understanding of categories (The same reason is applied while choosing a sandbox for illustrating our classification scheme in Chapter 4.). We make use of the ambiguity, with respect to categorization, in Windows NT, to use one system

(Operating System) and explain how the two categories fit in, thus providing an easy way distinguish between them.

We provide a brief introduction to Windows NT in Section 3.4.1 and present the component based (sandbox implementation) category in Section 3.4.2. In Section 3.4.3 we present the layered approach. Figure 3.3 shows the a block diagram of Windows NT.

3.4.1 Introduction to Windows NT

Windows NT is divided into three parts; User level Subsystems, Kernel Level Subsystems and the Hardware layer. A block diagram of Windows NT is shown in Figure 3.3.

3.4.1.1 User Level Subsystems

The use of different subsystems implies that Windows NT has at least as many points of failure as there are subsystems. The user level subsystems in Windows NT correspond to an OS/2 Application platform, a Win32 application platform, a POSIX application platform, each acting as an independent virtual machine for its processes. The different Virtual Machines are responsible for the secure execution of the processes executing within them.

3.4.1.2 Kernel Level Subsystem

The main part of the Kernel is the executive services platform. The executive platform can be further divided into major components such as Object Manager, Process Manager, a Local Procedure call Facility, and a Virtual Memory manager. A key component that could be subject to serious security flaws is the I/O manager. The I/O manager contains subcomponents such as the cache manager, the file system

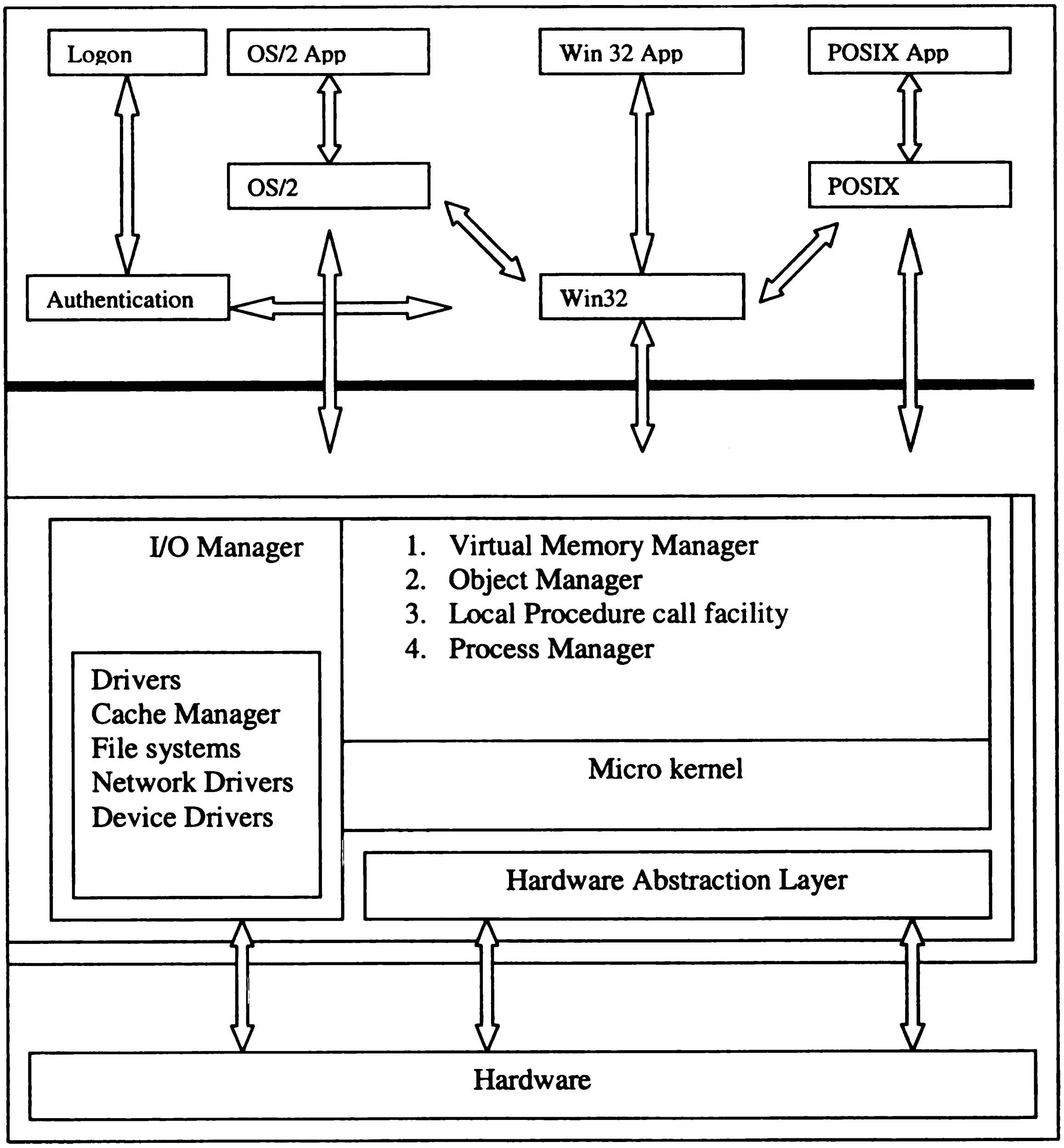

Figure 3.3: Windows NT block diagram

handler, the network drivers and device drivers. An implication of additional complexity for the I/O manager is that it becomes difficult to monitor additional code that is integrated into the Kernel level-operating mode. Such a scenario whereby a compromised component, either maliciously or by oversight, is introduced in the kernel level compromises any security mechanism that is built into Windows NT at the user level. An authentication system in isolation will be unable to provide the necessary security, due to the presence of other entry points, including but not limited to network connections. Compromised Kernel component can result in escalation of privileges for malicious user level programs. Escalation of privileges is one of the main sources of buffer overflow attacks as explained in [50].

3.4.1.3 Hardware Components

In Windows NT, the hardware abstraction layer provides the interface to the hardware with which the operating system interacts with.

3.4.2 Potential Security Flaws in Windows NT

Before discussing the two categories, we will evaluate possible security flaws in Windows NT. Buffer overflow [4] is a recurrent theme in most security attacks [9]. For example, in the OS/2 Subsystem, we can have an OS/2 application that could be compromised. The compromised OS/2 application can then hijack a core component in the Win32 subsystem. We note that the POSIX subsystem also has access privileges, as described in [49], with the Win32 subsystem. Thus it is possible for the compromised OS/2 Application to take over and launch further attacks against any user level process. Further, the compromised Win32 subsystem can be used as a point of attack to sabotage

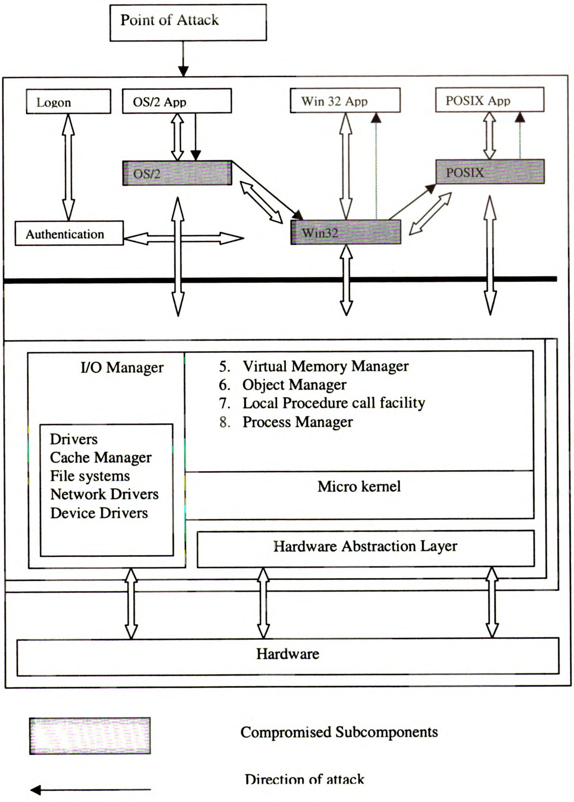

Figure 3.4: Security Flaw in Windows NT

or hijack communication with the kernel level applications. Even though the Windows NT executive services itself is not compromised during the process, communication with the executive will be compromised through the hijacked process or subcomponent (Win32 in our example). Figure 3.3 illustrates the scenario where one component is compromised.

Let us consider another scenario in which the user loads a new device and the device driver from a trusted third party vendor. The device driver code may contain some genuine leaks/holes that could be exploited. It is then possible that the executive is compromised leaving the operating system in a compromised state. While a driver can be contained, it is necessary to be prepared for the scenario in which such containment is necessary.

3.4.3 Category 1 – Component based design

We have identified six major components for the (hypothetical) sandbox for Windows NT. The six components are listed in Table 3.1. The relative interaction between the components was shown in Figure 3.1. We now describe the individual components.

Table 3.1: Sandbox components for Windows NT

Component#	Description
1	A Sandbox Controller
2	OS/2 Sandboxing component
3	Win32 Sandboxing Component
4	POSIX Sandboxing Component
5	Windows Executive Services Sandboxing Component
6	Hardware Protection (TCPA/Secured Memory Access)

3.4.3.1 A sandbox Controller

The sandbox controller coordinates the functionalities of the other sandbox components. The sandbox controller is a central point of control. One key disadvantage for any centrally controlled security mechanism is that the failure of the key component can lead to the failure of the dependant components, or the failure of the sandbox as a whole. However, we can insure the system against such failures by carefully selecting the policy. The same can be done for any component; However, the importance of doing so is magnified in a controller because; compromising individual components can result only in compromising the subsystems to which they correspond. Doing so to the controller could jeopardize the entire system.

3.4.3.2 Subsystem based sandboxing components

There are several subcomponents defined in the sandbox. These include the OS/2, Win32 and the POSIX sandbox components. The advantage of such a sandbox is that compromising an application in the OS/2 system will not affect the Win32 and the POSIX systems, and is a way to insulate independent components (POSIX, OS/2) from each other. Subsystem based sandboxing components also help in preventing the hijacking of the components, through malicious system call interceptors.

3.4.3.3 Kernel level security components

The kernel level security components include a sandboxing mechanism for the executive services such as device drivers, cache memory interface software, object manager security interfaces, and local procedure safety issues. A major functionality for the kernel level components is for checking against return address protection for kernel procedure calls, which in turn help in protecting against buffer overflow at the kernel level, thereby preventing an escalation of privileges for the malicious program. In a component-design a kernel component is usually different due to the complexity of managing the various kernel features. Further, extra caution has to be taken before patching in such a component because, compromising the kernel component is another way of compromising the whole operating system.

3.4.3.4 Hardware Protection

We do not rule out the possibility of a hardware based protection mechanism. Future security mechanisms could be based directly on hardware protection mechanisms. Trusted Computing Platform alliances are an effort towards hardware protection. Given the amount of industry effort [10] for hardware protection we cannot rule out significant levels of sandboxing at the hardware level, justifying the inclusion of hardware protection as a component.

Figure 3.5 shows the block diagram of Windows NT after including the sandbox components.

3.4.4 Category 2 – Layered design

Similar to our discussion of the component based model we present the layered design. We have identified 5 layers, and have very briefly discussed them below.

3.4.4.1 Logon/Authentication layer (LAL)

The Logon/Authentication layer will form the topmost layer of the Sandboxing security stack. User authentication modules in operating systems will fall in the LAL.

3.4.4.2 Virtual Machine Handler (VMH)

The Virtual Machine Handler layer, which will also operate in the user level, is more useful for the containment of processes within the individual subsystems such as the OS/2, Win32 application, and POSIX applications.

3.4.4.3 System Call Interception Layer (SCIL)

The System Call Interception Layer is important for the interception of system calls, between all the individual subsystems, and the different executive or kernel level services. The information gathered in SCIL could be passed on to the higher layers.

3.4.4.4 Kernel Components Monitor (KCM)

The different kernel services such as object manager, the process manager, the local procedure call facility, and the virtual memory manager are monitored by the KCM layer.

3.4.4.5 Hardware Components Layer (HCL)

The Hardware Components Layer, at the lowest end of the stack, will form the basis for any hardware action that may be taken to promote sandboxing. At this point, such a layer will serve only the purpose of scalability. The sandbox itself is shown in the Figure 3.6.

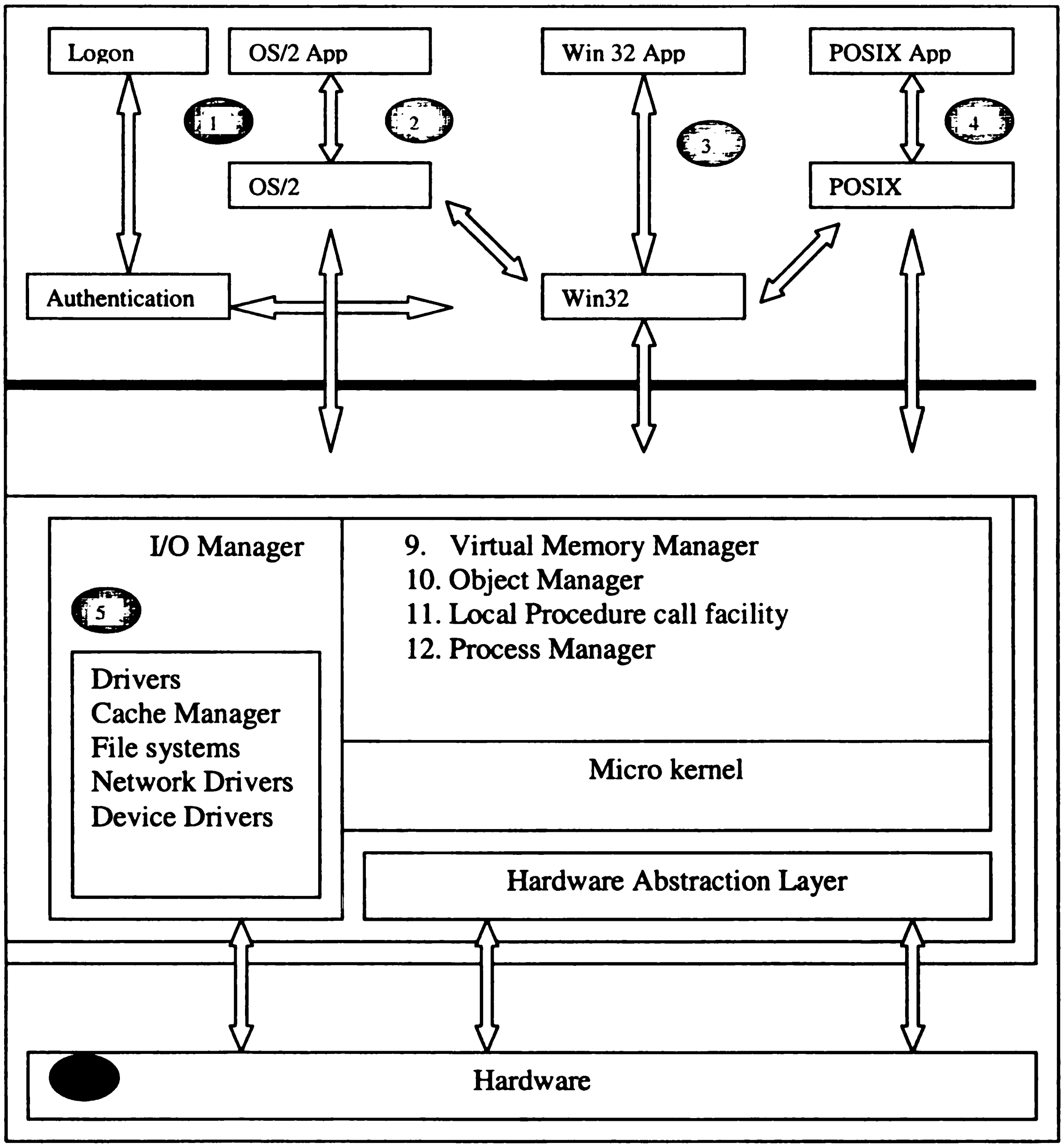

Figure 3.5 : Windows NT after including sandbox components

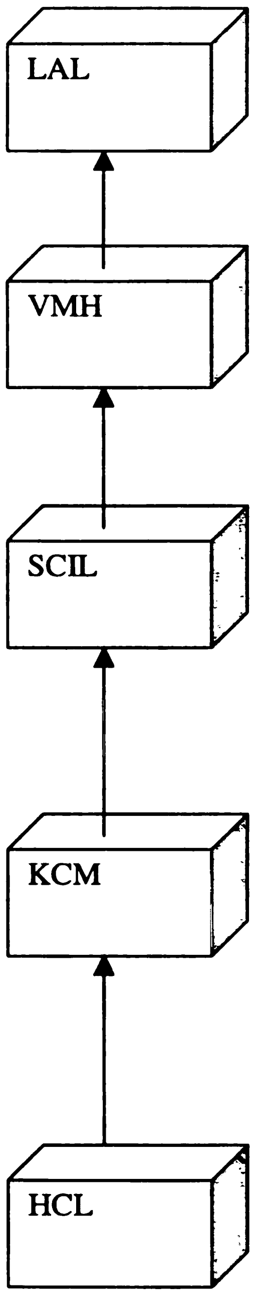

Figure 3.6: Windows NT with the layered design.

3.6 Conclusion

In this chapter, we presented two sandboxing architectures and analyzed the differences with the help of an example. While the architecture itself could be categorized as a primitive, we chose to give it a different dimension because architecture is not defined by a policy in case of system, which requires a dynamic translation of policy requirements, which is increasingly the case. In Chapter 4 we have mentioned what category the sandbox falls with respect to the architecture. However, we have not used the architecture in the visual representations in Chapter 5. The reason is that the visual representation is a tool to enable designers to define behavioral aspects of a sandbox as against implementation aspects. Implementation aspects will be difficult to predict without taking the underlying system into account.

4. Classification of sandboxes – An illustration with JVM

In this chapter we present the analysis of JVM with respect to our primitives. The illustration will be typical and apply to other sandboxes as well. We first present a brief description of the JVM sandbox, including its features, and the classification. Subsequent to the actual classification, we will also present how we are able to provide fine-grained distinction between two closely related sandboxes within our classification system. For demonstrating fine-grained classification we will choose two sandboxes that are closely related within the JVM family. We will give reasons for their choice, outline similarities, and then describe how our classification system is able to present a clear difference between the two.

4.1 Java Virtual Machine (JVM)

The JVM is the one of the most popular sandboxes in use and a pioneer of the concept of sandboxes for security. The virtual machine provides a sandbox for the applications to execute. The actual implementation of the JVM is discussed in [24]. The JVM security architecture protects the system from third party code in the form of a sandbox. The JVM provides its own instruction set and a built in compiler. The JVM security [41] is based on the following important components:

- The class loader architecture
- The class file verifier
- Language based security
- The security manager and the Java API

The class loader architecture prevents malicious code from interfering with trusted applications by forming a protective layer around the trusted applications and places code into separate sandboxes. The idea of a class loader is directly tied to the programming language concepts of namespaces.

The class file verifier provides consistency checking between the different modules of the class loader. The verifier is a multi-pass entity, with different passes performing structural checks on the class file, semantic checks on the file data, bytecode verification and verification of symbolic references.

Language based security features include type safe reference casting, structure memory access, automatic garbage collection, array bounds checking, and checking references for NULL. The security manager and the Java API help maintain the integrity of the external entities (processes, applications).

In the following sections we discuss the JVM with respect to our sandbox primitives. However, we have kept the discussion at a more abstract level. We will, however, demonstrate a more fine-grained differentiation between the different versions of JVM in chapter 5. Specifically we will choose JVM1 and JVM2 to demonstrate the effectiveness of our idea.

4.1.1 Access Control *(Acc_cntrl):*

We have assigned *Acc_cntrl* for JVM, the value of Policy Based Access (PBA). Restrictions in JVM have to be provided by the system administrator during implementation. Policy can be changed statically. The policy applies uniformly to all applets that execute within the JVM. Therefore we do not differentiate between TApp and UTApp. PBA also has the highest score, 16, despite the static nature of PBA in JVM. One argument against the high score will be the static nature of JVM. We address the issue in *Dyn_rst*.

The JVM is accessible to programs or applets compiled into the bytecode for JVM, and provides a very simple form of access control for the sandbox. However, there is no protection offered to individual applets from other applets within JVM. Due to the increase in the presence of mobile code, interesting access control mechanism for JVM based systems have been proposed [29]. However, these are not exactly a part of the system. Thus the operating system and execution environment is protected from malicious JAVA applets; genuine applets within the sandbox are not protected from malicious applets.

4.1.2 Dynamic Policy enforcement *(Dyn_rst):*

Dyn_rst is on the higher side for JVM. As discussed in Section 4.1.1, the system administrator can change the policy, upon which the changes come into immediate effect. Compared to the best case scenario (a score of 16) for *Dyn_rst*, which is a fully dynamic translation of policy, we have assigned a score of 12 for JVM. The code for *Dyn_rst* for JVM is DH/TApp – DH/UTApp. This means that *Dyn_rst* for JVM takes the same value for both trusted and untrusted applications. Specifically, in the case of JVM the applications are Java applets.

Dyn_rst is also dependant on the *Acc_cntrl* methods for JVM. However, *Acc_cntrl* works only in generic terms, that is, to protect the execution environment (operating system) from malicious processes, and not necessarily a genuine applet from a non-genuine one. We discuss the policy enforcement only with respect to the JVM versus non-JVM processes. Intra JVM process protection is outside of the purview of policy that drives *Dyn_rst*. Changes to policy cannot be made dynamically.

Some of the important policy measures that are the highlight of the JVM security policy include the following. JVM prevents applets from untrusted entities from reading or writing to the disk, a protection mechanism that is applicable to operating system services as well. JVM also prevents any malicious process from establishing network connections with any host other than the host from which the applet was downloaded. JVM prevents the applets from creating new applets (unless explicitly allowed by policy). JVM also prevents a new applet from downloading a dynamic library that calls a native method.

The latter two features have interesting security applications. First, malicious programs that rely on rapid self-replication to consume system resources are ruled out in JVM. Second, the provision of a strong type checking mechanism helps protect against buffer overruns. While Java's memory management system is rendered unpredictable because of the provision of the systems own garbage collection routines, the possibility of one the most serious security problems, namely the buffer overrun is ruled out. The only possibility for a buffer overrun now remains through the invocation of a native method, and providing dynamic argument passing capability.

4.1.3 Sandbox Number (*Sbox_no*):

The *Sbox_no* can be justifiably counted at one that is the JVM itself. The JVM is one big sandbox, as described earlier, between JAVA applets versus the rest of the system, as described in Figure 4.1. However, we also observe that every process of a specific type (i.e. Java applets), is handled within the sandbox. Therefore policy is enforced for all process. As a result we assign the value D, since for JAVA applets the policy is dynamically applied. The value for the JVM results in D/PB. This also implies that JVM gets a score of 4 for *Sbox_no* as discussed in Chapter 2.

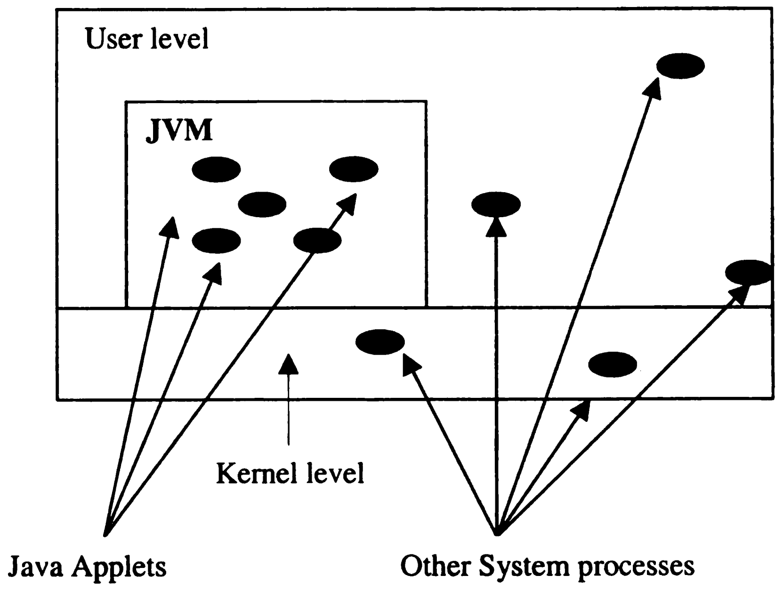

Figure 4.1: *Sbox_no* for JVM

4.1.4 Operation Mode *(Op_mode):*

The value of *Op_mode* is fixed at the user level (UL) for both the sandbox components (SboxC), and applications (App) contained within the sandbox. Thus the value of *Op_mode* is provided the code SboxC/UL – App/UL for JVM. The relative score for JVM is 2, and is on the lower side because of the lower variation in *Op_mode*, thereby limiting the scope of the applications that could be protected.

Here we provide a brief discussion of JVM related features that were taken into consideration while imposing our ordering. The operation mode of the JVM as a sandbox is the user level since the scope of the JVM is defined as processes that execute in the user level. The processes include untrusted third party applets, and genuine kernel code. Tasks that require the execution of applications in the kernel level are performed through the invocation of native methods. Thus kernel level execution is decoupled from processes operating in the sandbox. One implication of such a decoupling of actions is that a malicious process cannot deny other processes in the system any less resource than what they already have. The fundamental security features in SboxC/UL – App/UL *Op_mode* include the programming language based security, which is provided through strong type checking [30], and the presence of a garbage collector.

4.1.5 Inter-sandbox communication *(Isbx_com):*

The value of *Isbx_com* for JVM is MISC/TApp – MISC/UTApp. This is largely because the middleware present in the JVM enables communication between the applets and other processes that execute outside the JVM, as well as between applets on different computers. The score for MISC/TApp – MISC/UTApp is on the higher side (14), because in conjunction with a score of 12 for *Dyn_rst*, *Isbx_com* will not compromise security directly (excluding the possibility of a faulty middleware over which the sandbox has no control).

Here we provide a brief discussion of *Isbx_com* with respect to JVM. *Isbx_com* is a redundant feature for the JVM. There is only one sandbox, and all processes, usually JAVA applets can communicate between them. Communication between the processes in the sandbox and those outside is through well-established policy measures as defined in the JVM specification [29]. Thus the safety other applications executing in the JVM execution environment is the responsibility of the overall system policy (operating system security policies for example) and the leeway that the policy allows for the JVM itself, and not for the individual processes. The individual processes could have the same scope as that of the JVM. Provision of any restrictions on the processes within the JVM is entirely arbitrary. Related JVM security features include, the class loader architecture [31], the class file verifier [32], the security manager and JAVA API [33].

4.1.6 Sandbox Time to Live *(Sbox_ttl)*:

For the JVM the *Sbox_ttl* value is Prcs_lt. This implies that the sandbox applies to the applications throughout their lifetime. As a result we also provide a higher score of 7 (on a scale of 8), for *Sbox_ttl* for JVM. The reason why we haven't assigned 8 is because we make a distinction between Prcs_lt and Sys_upt, wherein the latter is considered a better measure of security.

4.1.7 Sandbox Scope *(Sbox_scope):*

The *Sbox_scope* value for JVM is G/TApp - G/UTApp. The scope of JVM is global (G), and is true for both trusted and untrusted applications, since applets have no choice but to execute inside the sandbox. The value assigned to JVM *Sbox_scope* is 16, and is the highest. The values for JVM are summarized in Table 4.1.

Table 4.1: Summary of values for JVM

Features Available	**Available**	**Values**
Acc_ctrl	PBA	16
Dyn_rst	DH/ TApp- DH/UTApp	12
Sbox_no	D/PB	4
Op_Mode	SboxC/UL – App/UL	2
Isbx_com	MISC/TApp – MISC/UTApp	14
Sbox_ttl	Prcs_lt	7
Sbox_scope	G/TApp - G/UTApp	16
Scl	Y	8
Int_op	Y	
Nsss	Y	

4.1.8 Sandbox design category

The JVM is a single entity, with the policy enforced in the form of different sandbox components. Thus there is no possibility of a layered approach possible with the limited functional separation of components. Thus categorizing the JVM as a component-based architecture will be appropriate. The idea is further reinforced from our discussion in Section 4.1 about the basic building blocks of JVM.

Component based architecture in turn provides JVM with the advantages and disadvantages that come with component based sandboxes. Most notable advantages of the sandbox, is the provision of enough safety components as may be necessary to protect the rest of the execution environment (operating system) from malicious third party code. A prominent disadvantage of such a sandbox is the inability to protect genuine Java applets from malicious code that operate within a single execution environment (JVM in our case). The figure 4.2 illustrates the JVM designed in the form of a component-based architecture.

4.2 Comparative Analysis of JVM Versions

We will use the JVM-1 and JVM-2 to demonstrate how abstract, and seemingly unrelated, security ideas can be linked together to make a comprehensive comparison between two different implementations of a sandbox. As we will see in this section, there are significant differences between the implementation and specification of the two versions, even though backward compatibility has been maintained for all programs. In this section we will very briefly describe the JVM, describe specific performance characteristics in JVM, and investigate features of the JVM with respect to our primitives based on the implementation.

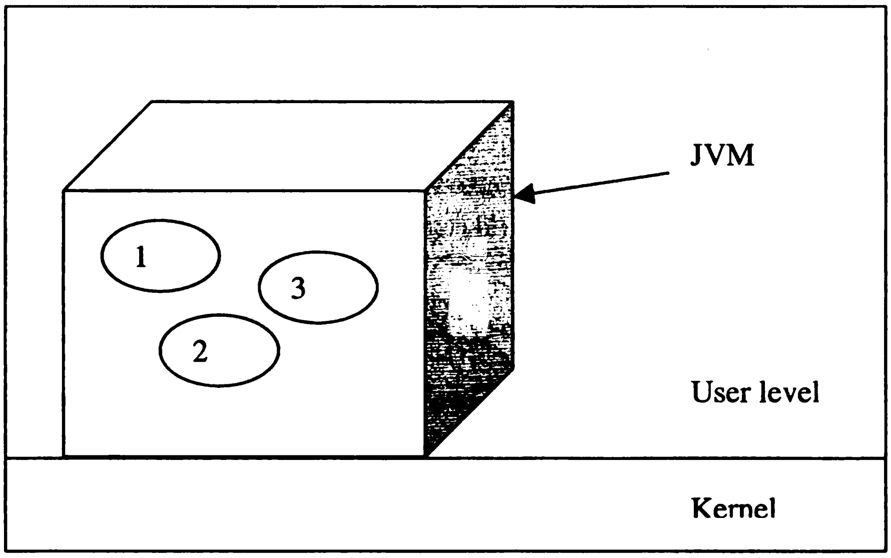

Figure 4.2: Category description for JVM

As discussed in Section 4.1, the fundamental components of the JVM have remained the same. These included the class loader architecture, the class file verifier, language based safety features, and the Java API, are structurally identical. However, functionally they have undergone a great deal of changes, as may be illustrated by the application of our classification scheme. Some of the important characteristics of a JVM used to evaluate its performance, and the units required to measure it are given in table 4.2. The metrics are adopted from [39].

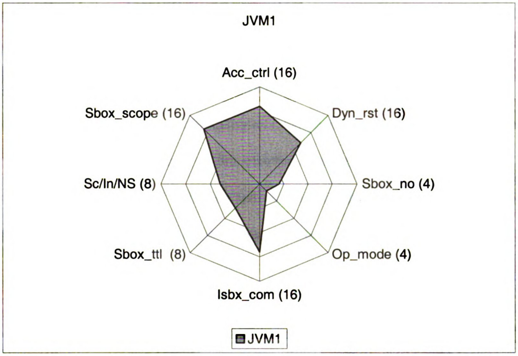

Figure 4.3: JVM1 visualization

Table 4.2: Metrics for evaluating JVM performance

Description	**Units for measurement**
Number of active thread groups	Integer
Minimum number of active thread groups	Integer
Maximum number of active thread groups	Integer
Number of active threads	Integer (Number of Threads)
Minimum number of active threads	Integer (Number of Threads)
Maximum number of active threads	Integer (Number of Threads)
Uptime for JVM	Milliseconds
Free heap space for in JVM	Kilobytes
Minimum amount of heap space	Kilobytes
Maximum amount of Heap space	Kilobytes
Total amount of Heap space	Kilobytes
Minimum Total amount of Heap Space	Kilobytes
Maximum Total amount of Heap Space	Kilobytes

4.2.1 Why JVM?

The reason for choosing JVM for the comparative analysis is to illustrate how fine-grained differences can be observed using our classification scheme. JVM1 and JVM2 illustrate perhaps the closest implementations of a sandbox idea, yet different enough for each one of them to be a separate sandbox by it's own right. The latter observation however, is only a technicality as JVM2 performs better than JVM1 on many aspects. Also, as we will see later in this chapter, both versions of JVM retain the same fundamental architecture, and will also fall within the same category, while we clearly are able to establish the differences between them on the basis of the primitives that we have defined.

Previous related studies with respect to JVM performance is given here. One detailed study mentioned in [40], performed comparative studies on the performance of a typical application (JXTA was chosen for this purpose), on two different versions of the JVM (1.3 and 1.4). JVM 1.4, with a few additions [42], became JVM 2. The important parameters studied in this chapter include differences in component design and constraints, such as ease of use, data portability, independence of platform source code, JVM friendly class design, Parameter based benchmark configuration such as startup time for JXTA applications (which will be applicable to other network based programs), and round trip time for JXTA applications. We will base our study on the performance results mentioned in [40]. Below, we will enlist the different parameters and find out the differences between the two versions. A summary of the differences is also provided in table 5.

4.2.2 Access Control *(Acc_ctrl):*

For JVM1 the value of *Acc_cntrl* is UA/TApp–CA/UTApp. While trusted applications are provided unconstrained access (UA), untrusted applications are provided with constrained access (CA). Due to the selective basis for access control, we have provided JVM1 with a value of 16, which is also the highest value rated to *Acc_cntrl.* JVM2 provides PBA, and is also provided a score of 16. The reason both values are provided the same value for *Acc_cntrl* is that they offer no difference in the final security provided to processes outside the JVM, even though the manner in which they provide this security varies slightly. We only discuss quantitative implications of the changes with respect to our classification scheme.

Acc_cntrl has evolved through the different versions of the JVM. We use results reported in [43] as the basis for our study of differences in access control and related primitives in the different versions of the JVM. Figure 4.4 clearly illustrates the various stages of evolution. JVM treats, access control for resources differently for applets and Applications that are collections of applets. The analogy holds true for threads, and process, which are collections of threads

In JVM1, the treatment for the applets for *Acc_cntrl* is different from that for Applications. In JVM1 constrained access is given to trusted applets (CA-TApp), and unconstrained access is provided to trusted applications (UA-UTAppl). As a further advancement, within JVM1, unconstrained access was provided to trusted applications, while constrained access was given to untrusted applications.

In contrast in JVM2, both applets and applications are provided with policy-based support implying that that all options are provided on a run time basis. Further the

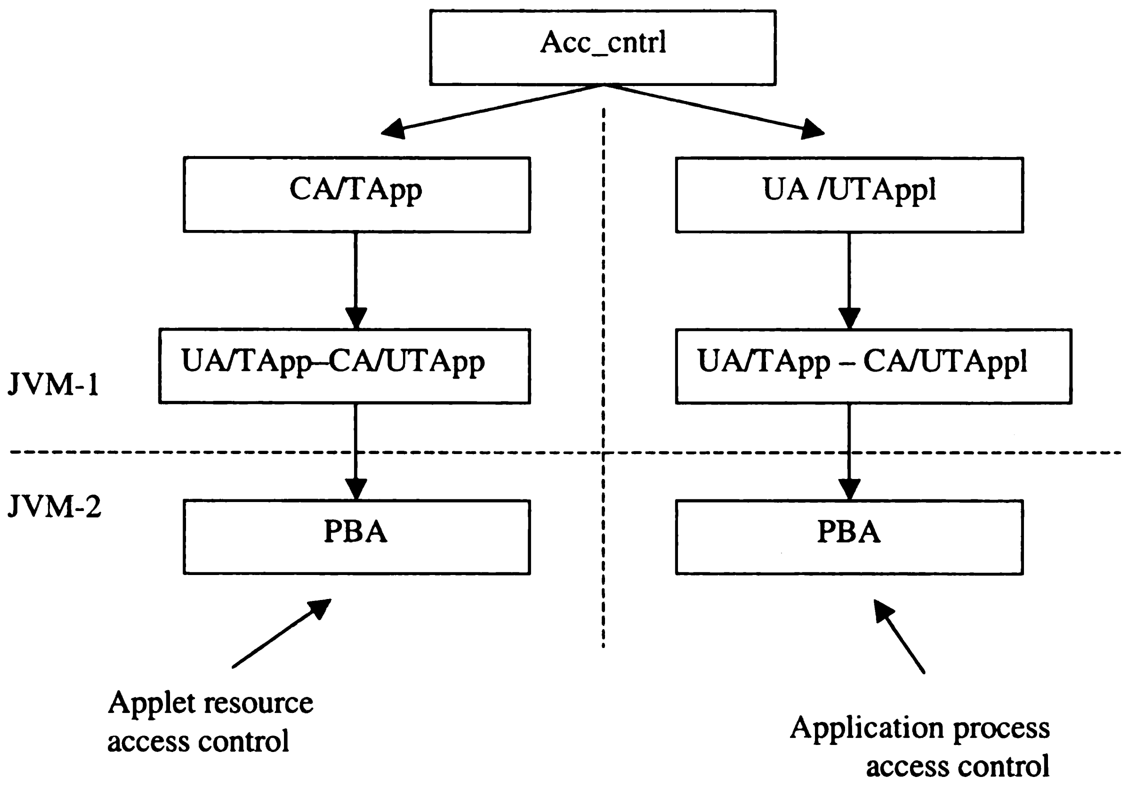

Figure 4.4: Summary of differences for *Acc_cntrl* for JVM

success of the security mechanism here is intrinsically tied the presence of a dynamic policy translation existence. Thus JVM2 provides runtime access controls in significant measure to the system for both applets and processes.

4.2.3 Dynamic Policy Enforcement (Dyn_rst)

JVM1 provides comparatively lower measure of *Dyn_rst* with both applets and applications. Most of the policy is statically bound with the application. Policy is described as based on whether the application is trusted or not, or whether the source of the application is trusted or not.

JVM2 provides a higher degree of *Dyn_rst* and we have given a higher score of 14 because JVM2 offers a greater degree of security compared to JVM1. Based on data provided in [43], we are able to illustrate *Dyn_rst* primitive with respect to both the JVM versions is given in Figure 4.4.

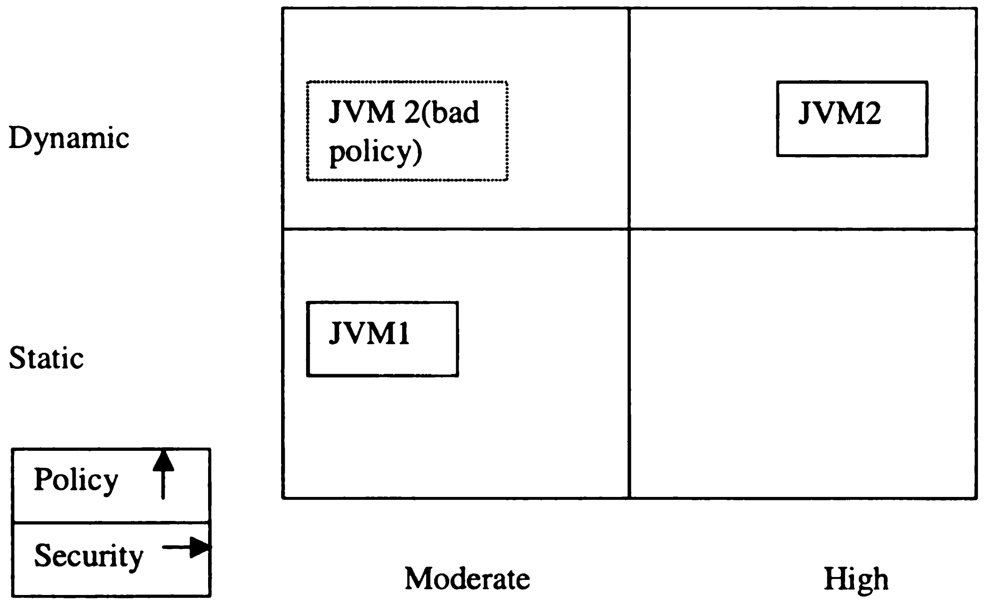

Figure 4.5: *Dyn_rst* Summary for JVM

4.2.4 Sandbox number (Sbox_no)

Both JVM1 and JVM2 are similar with respect to *Sbox_no*. We have assigned the value S/PB to JVM1 and D.PB to JVM2. The similarity is that both allow a policy based numbering for *Sbox_no*. Even when there is only one installation of JVM (both versions), we can identify different sandboxes if policy is applied to differently to different applets, and it is possible to do so. However, the two versions vary in how the translation of policy. While JVM1 allows changes, the system administrator can do them only statically. JVM2 allows dynamic changes to the policy that is applied to different parts of JVM. We have not differentiated between the two versions of JVM while imposing a numerical ordering. Both JVM1 and JVM2 have been assigned the highest score of 4.

We describe some of the important parameters that affect the value for *Sbox_no* for JVM. *Sbox_no* for the JVM is dependant on parameters such as minimum and maximum number of thread groups, minimum and maximum number of threads, minimum and maximum amount of heap space, as well as the total heap space allowed. Table 4.2 summarizes all the parameters and gives an idea of metrics used to measure them.

4.2.5 Operation Mode *(Op_mode)*

For *Op_mode* JVM1 is assigned the value SboxC/KL – App/UL, while JVM2 is assigned value SboxC/UL – App/UL. Applets execute in the user level in both versions of JVM. However sandbox components (described in Section 4.1) differ in their *Op_mode* value. We recollect here that *Op_mode* is a collection for operating modes for two aspects of a sandbox. The sandbox itself, and the processes or applications that it sandboxes. We assign a higher score (4) for JVM2 since none of the sandbox components operate in the kernel mode. We assign a score of 2 for JVM1 due to the fact that compromising the sandbox will result in compromising the kernel.

We will briefly discuss the differences between JVM1 and JVM2. In JVM1 most of the class loaders operated in the user level, with very little part of JVM1 operating in the kernel mode, this changed in JVM2. Bytecode verification became a more complicated process, and more functionality such as verification of symbolic references and linking to dynamic libraries were shifted to the kernel. Bytecode verification had a definite effect on securing the JVM components [42]. However, an indirect effect of changing Bytecode verification was that user level process, applets or Java applications, could be provided more freedom, with enhanced security. Thus while in JVM1 operation mode for sandbox components is kernel level (SboxC-KL), the applications themselves are sandboxed at the user level thus making the scope of the sandbox user level (App-UL).

4.2.6 Inter Sandbox Communication (*Isbx_com*)

Both JVM1 and JVM2 have identical values for *Isbx_com*. We have assigned MISC/TApp – MISC/UTApp for JVM1 and JVM2. Thus both trusted applications (TApp) as well as untrusted applications (UTApp) have to use a middleware induced communication (MISC) mechanism to communicate with other processes, including those that are inside as well as outside the sandbox (i.e. JVM). We have given both JVM1 and JVM2 high scores, but sought to make a slight distinction between them due to reasons mentioned below. JVM2 is provided the score of 16 while JVM1 is provided a score of 14.

One difference between JVM1 and JVM2 is in terms of newer additions to JVM2 in other primitives that impact *Isbx_com*. For example in *Sbox_no* we saw that JVM2 has a value of D/PB, increasing the possibility of simultaneous presence of multiple sandboxes. Intersandbox communication *(Isbx_com)* is taken into account when we have multiple sandboxes. Interprocess communication through middleware such as CORBA, and through RMI enables the introduction of security abstraction for communication. Using Middleware is yet another advantage of JVM, as well as a similarity in both JVM1 and JVM2. Thus JVM1 and JVM2 both have Middleware induced secure communication (MISC). The reason why inter process communication is likened to inter sandbox communication (*Isbx_com)* is because the case of JVM, secure Interprocess communication is a guarantee due to its existing middleware.

4.2.7 Sandbox Time to Live *(Sbox_ttl)*

We have assigned a *Sbox_ttl* value of Prcs_lt, representing the process lifetime. The sandbox is in place as long as the applet executes. There is no way for a process to execute outside the JVM and is the case with both versions of the JVM. Both versions of JVM receive a high score for *Sbox_ttl.* We have assigned a score of 8 for JVM2, while we assigned a score of 7 to JVM1. The reason for the slightly lower score for JVM1 is because we seek to relate *Dyn_rst* and *Acc_cntrl* with *Sbox_ttl* and the relative weaknesses of JVM1 compared to JVM2 with respect to these primitives.

4.2.8 Sandbox Scope *(Sbox_scope)*

For *Sbox_scope* we have assigned a value of G/TApp – G/UTApp for JVM1 and L/TApp - G/UTApp. The global nature of *Sbox_scope* in JVM1 is a reason for the original release of JVM1 being highly restrictive, even on genuine applets and applications. The global nature changed with release 1.3.1, however, we will restrict ourselves with the broad, and widely applicable cases for release 1. One difference, therefore, between JVM1 and JVM2 is that in the case of trusted applications (TApp) the scope was local in JVM2. We have assigned a value of 16 (the highest possible for *Sbox_scope)* for both JVM1 and JVM2, since the G/TApp for JVM1 is only more restrictive and hence more secure.

4.2.9 Scalability *(Scl)*

JVM scales to larger systems due to the components based architecture. *Scl* is particularly enabled because the most important part of the complexity is the spawning of several independent sandboxes. The large number of components will in turn increase the complexity of the Intersandbox communication *(Isbx_com)*. However, as seen earlier in the chapter that has been left to a completely different middleware component.

4.2.10 Interoperability (*Int_op*)

A system installed with JVM provides for interoperability with other systems installed with JVM. Also backward compatibility between JVM1 and JVM2 exist. Thus there is no fundamental difference between the two sandboxes with respect to *Int_op*.

4.2.11 Native security services utilization *(Nsss)*

JVM1 and JVM2 both make use of native security services. There is no differentiation between the two with respect to *Nsss*. Again; the reason for no differentiation between the two services is that the class loader and verifier mechanisms have remained almost the same between the two systems. *Nsss* enables the JVM to load local, and platform dependant security features, even as the applications and applets themselves are platform independent. Table 4.3 summarizes the comparative analysis of the JVM1 and JVM2.

Using the scoring scheme developed in chapter 2, a summary of the scores between to two sandboxes is shown below. As mentioned previously the scoring mechanism (summarized in Table 4.4) is to be used as a guide for implementing the visualization process, and represents a relative ordering that we have imposed on the various values that we identified for the primitives. Figure 4.6 represents the visualization of JVM1 and JVM2.

Table 4.3: Summary of values for JVM1 and JVM2

Primitive	JVM1	JVM2
Acc_cntrl	UA/TApp–CA/UTApp	PBA
Dyn_rst	SM/ TApp-SM/UTApp	DH/ TApp-DH/UTApp
Sbox_no	S/PB	D/PB
Op_Mode	SboxC/KL – App/UL	SboxC/UL – App/UL
Isbx_com	MISC/TApp – MISC/UTApp	MISC/TApp – MISC/UTApp
Sbox_scope	G/TApp – G/UTApp	L/TApp - G/UTApp
Sbox_ttl	Prcs_lt	Prcs_lt
Scl	Y	Y
Int_op	Y	Y
Nsss	Y	Y

Table 4.4: Summary of scores for JVM1 and JVM2

	Acc_ctrl	Dyn_rst	Sbox_no	Op_mode	Isbx_com	Sbox_ttl	Sc/In/NS	Sbox_scope
JVM1	16	12	4	2	14	7	8	16
JVM2	16	14	4	4	16	8	8	16

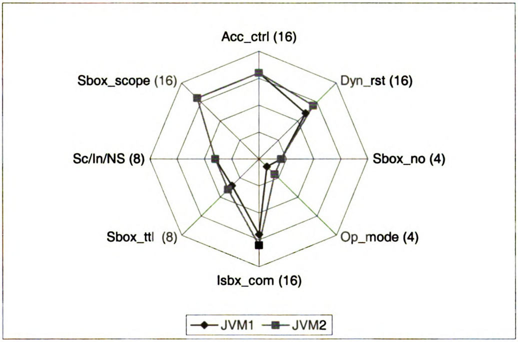

Figure 4.6: Visualization: JVM1 vs. JVM2

4.3 Observations

From Table 4.5, and the discussion leading to it, we clearly are able to observe differences between the JVM1 and JVM2. We observed that the two JVM versions have very similar implementation techniques, with minor differences. However, from a sandbox perspective, the two JVM versions are different, and using our parameters and classification scheme, we are able to establish the differences between the two. The differences even, in cases where they are subtle are clearly seen. For example in all the three cases, i.e., *Dyn_rst*, *Op_mode*, *Sbox_ttl*, and *Isbx_com* we see that the sandboxes though identical to each other are differentiated. The implications are different for different primitives. *Dyn_rst* enables JVM2 to be placed in a higher security category, when backed with appropriate policy, compared to JVM1. We are able to observe that JVM2 is likely to stay longer as a constraining entity compared to JVM1.

We are able also able to see the similarities between the two sandboxes as well. The similar shape of the two sandboxes supports the hypothesis. For instance, for the primitive *Acc_cntrl* we see that both the sandboxes have the same value. Similarly for *Sbox_scope* we see that the maximum value is used for both the sandboxes. We also observe that the two sandboxes are identical with respect to *Sbox_no*.

4.4 Conclusion

In this chapter we demonstrated the application of our scheme by illustrating how we are able to classify with respect to the JVM. We also presented a visualization of the JVM using the technique that we have adopted. In order to illustrate fine-grained differences between two closely related sandboxes we chose the two different versions of JVM and applied our technique to them. Again, we illustrated the clear-cut differences between the two versions through the visualization process. We are able to observe that JVM2 completely eclipses the JVM1 in all respects. We are able to infer the same since the latter is contained within the former.

In chapter 5 we will present the remaining sandboxes and the differences between them. We will include JVM1 for the sake of completion while discussing the other sandboxes. In chapter 6 we will provide a comprehensive comparison based on the values between the individual features.

5. Analysis of the Sandbox Classification scheme

In this chapter we will analyze the sandbox classification scheme with respect to individual primitives. In Section 5.1 we present an interpretation of the visualization followed by a brief discussion of policy to design translation in Section 5.2. We then present a discussion on the representation of the classification results, including the basis for imposing the scoring of different value for a particular sandbox, and on the relative importance of primitives in Section 5.3. In Section 5.4 we present an analysis of how the five different sandboxes that we have analyzes compare with respect to the different primitives. In Section 5.5 we take two random subsets of sandboxes and see how well we are able to observe a pattern for observing common functionalities in sandboxes, even while the primitives establish a difference between them. A detailed description of how the values for the sandboxes were provided is provided in Chapter 6.

5.1 Interpretation of visualization

Figures 5.1 and 5.2 depict the summary of the classification scheme. In this section we describe the interpretation of the diagrams representing the sandboxes. An ideal sandbox can be visualized as one that occupies the polygon completely. However, such a sandbox will pose excessive constraints on the functionality, and will result in a significant overhead in the form of different security features that may not be needed rendering the sandbox secure but nonfunctional. Thus in practice we observe sandboxes that will only occupy a portion of the sandbox, and the shape of the sandbox will in turn be influenced by the policy and the security objectives that the sandbox has been built to achieve. While policy is important in deciding the shape of the sandbox, it is not the only factor. Other issues, including implementation issues, such as the architecture (or design category as described in Chapter 3) chosen will also influence decision making at the sandbox design level.

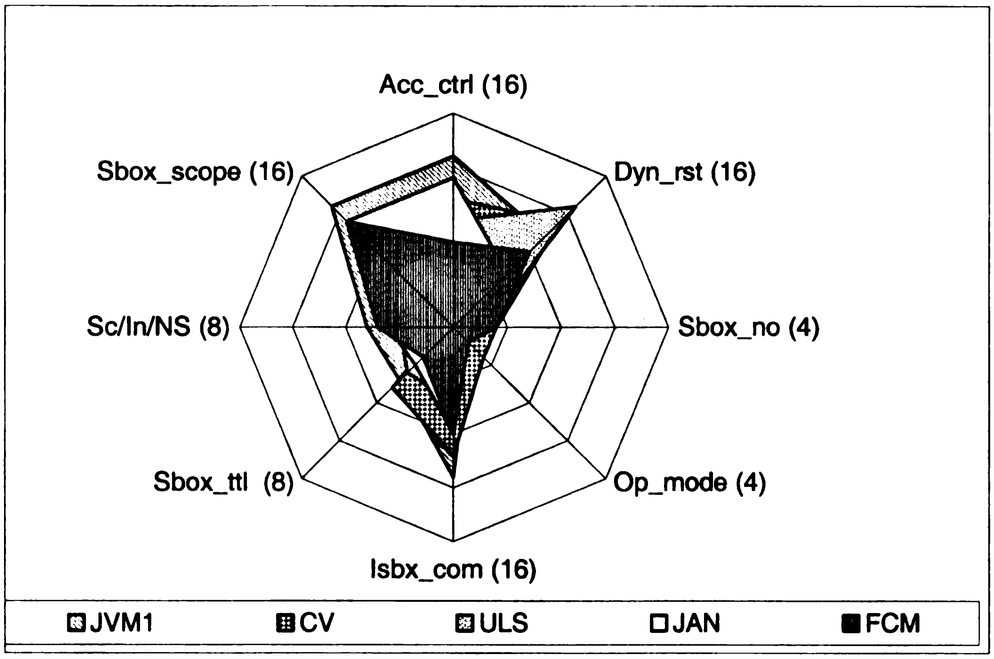

Figure 5.1: Occlusion Diagram

In figure 5.1 we have presented a representation with occlusion. The primary use of figure 5.1 is to identify how a sandbox can overshadow another sandbox with respect to a particular feature. Occlusion represents superior characteristics for the occluding sandbox with respect to a particular (occluded) sandbox. The idea of occlusion is consistent with our interpretation of the diagrams as well, wherein the sandbox occupies area directly proportional to the level of security offered. However, a sandbox that occupies smaller area need not necessarily be redundant, as it could also be a result of a reduced security requirement (or objectives) outlined by the policy.

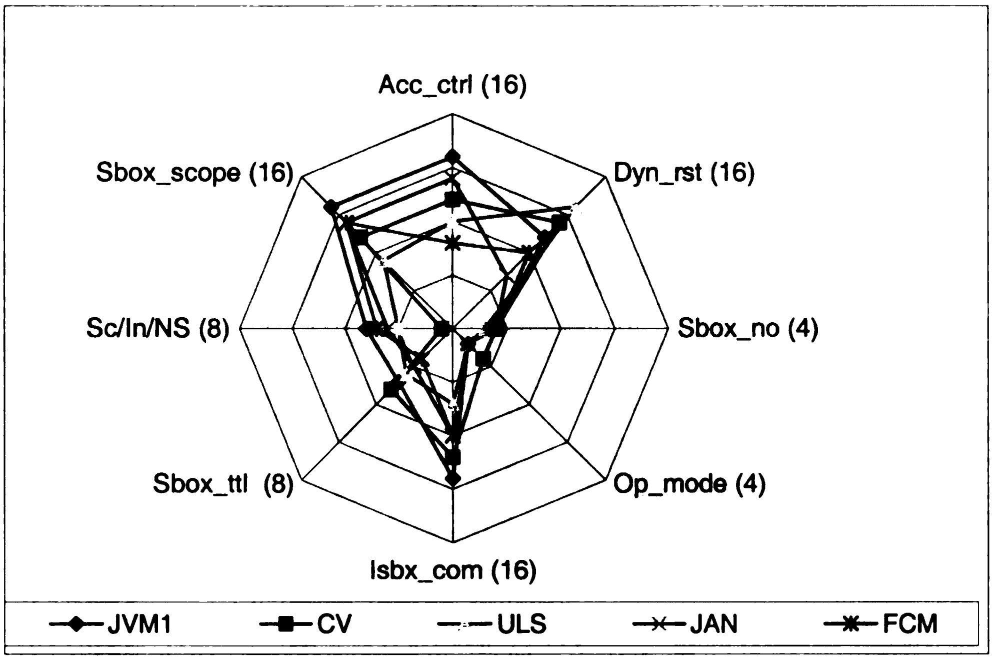

Figure 5.2: All sandboxes without occlusion

5.2 Policy to Sandbox design – The translation

In order to illustrate how our results can be used in the sandbox design, we need to look at the methods that could be adopted to translate policy to implementation. While policy itself is outside the purview of our work. Security policy must take into account the role of trust. The types of *Acc_cntrl* that we have defined as a primitive is an effort in this direction, wherein the designer can translate the policy perspectives on trust directly into implementation or design specification.

Confidentiality policies such as the Bell-Lapadula model can be useful in specifying system security policies. The primitives are defined in a way that a direct translation will be possible between the different systems easily.

Integrity policies can be specified in the form of several models that have been put forward in literature. Models such as the Biba Integrity model, Lipners Integrity Matrix model, and the Clark-Wilson Integrity model will be useful in any attempt at defining integrity policies. A good discussion of policies is presented in [52].

In this section we have attempted to summarize the primitives with respect to policy objectives. Table 5.1 provides a correlation between policies and the primitives.

Table 5.1: Policy to Sandbox design – the translation

Policy	**Primitives**
Confidentiality Policies	Acc_cntrl, Dyn_rst, Sbox_no, Sbox_scope
Integrity Policies	Op_mode, Isbx_com, Sbox_ttl

The above categorization is only a broad attempt at giving sandbox designers a direction on potential policy models. Security policy by itself is an important and a vast area, and a complete discussion of the different models is beyond the scope of this thesis.

5.3 Representation of classification results

While in Chapter 4 we have seen how the sandbox could differentiate fine-grained differences, we will, in this section, present an analysis on the basis for the scores that we impose on the different values that a primitive can take. We will assign specific numbers to the primitives and we will also be identifying the important primitives, from a classification standpoint, based on the number of points that two extreme values can differ. For example, a binary feature like *Scl* can only have one point of difference (or 2 values), while *Acc_cntrl* can have as many as 32 values (or 16 points). We will normalize the difference at each step to have a balanced comparison. We define normalization as the provision of equal weight to the different primitives during the visualization process. The table 5.2 illustrates, the primitives and the number of points of Difference (NPOD).

Table 5.2: The number of points of differences for primitives

Primitives	**NPOD**
Acc_ctrl	16
Dyn_rst	16
Sbox_no	4
Op_Mode	4
Isbx_com	16
Sbox_ttl	8
Sbox_scope	16
Scl	1
Int_op	1
Nsss	1

Based on the values in table 5.2 we can now represent visually the differences between the sandboxes. The differences between the sandboxes can be per-primitive; security based, flexibility based, or a combination of flexibility and security. Per primitive sandbox classification compares how the different sandboxes fare compared to the individual primitives. We will have this comparison for the most important sandbox primitives, those with NPOD greater than or at least four. For others, with NPOD less than 4 we have attempted to combine similar primitives, without any loss of information.

The table 5.3, gives a description of the numbers allocated to the different sandboxes on a single one point numbering system. Features are ranked in the order (which we impose) of importance from a security standpoint, and are assigned integer values. Then all possible combinations of the primitives, such as that listed for *Acc_cntrl,* in chapter 2 are numbered. The values are then assigned to the primitives for each sandbox, corresponding to its value specified in chapter 5. Table 5.3 provides a key for the abbreviations. While we have provided appropriate references to these sandboxes when they are described during classification, they are outlined for convenience in Table 1.

Table 5.3: Abbreviations for sandboxes used

Abbreviation	**Expansion**
FCM	Flexible Containment Mechanism
JAN	Janus
JVM	Java Virtual Machine
ULS	User Level Sandbox
CV	Chakravyuha

Table 5.4: Summary of scores for the sandboxes

	Acc_ctrl	Dyn_rst	Sbox_no	Op_mode	Isbx_com	Sbox_ttl	Sc/In/NS	Sbox_scope
JVM1	16	12	4	2	14	7	8	16
CV	12	14	4	4	12	8	1	12
ULS	10	16	3	2	7	6	5	9
JAN	14	7	4	2	10	5	6	14
FCM	8	10	4	2	10	4	7	14

The table 5.4 below has a cumulative score for the different sandboxes. The two cumulative scores represent different groupings of primitives. The grouping is done in order to demonstrate the importance of specific groups of primitives, as we will observe later in this chapter. We have selected the groups randomly.

Table 5.4: Cumulative scores for different sets of primitives

	Cumulative Score - 1	Cumulative Score - 2
JVM1	58	23
CV	50	17
ULS	42	17
JAN	45	19
FCM	42	19

Table 5.5: Primitive-wise score for cumulative score - 1

	Acc_ctrl	Dyn_rst	Isbx_com	Sbox_scope
JVM1	16	12	14	16
CV	12	14	12	12
ULS	10	16	7	9
JAN	14	7	10	14
FCM	8	10	10	14

Table 5.6 Primitive-wise score for cumulative score - 2

	Op_mode	Sbox_ttl	Sc/In/NS	Sbox_no
JVM1	2	7	8	4
CV	4	8	1	4
ULS	2	6	5	3
JAN	2	5	6	4
FCM	2	4	7	4

5.4 Primitive based comparison

5.4.1 Sandbox Number (Sbox_no):

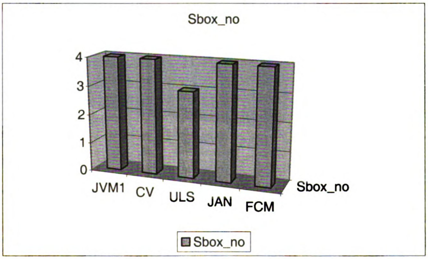

Figure 5.3: *Sbox_no*

We clearly see that of the different sandboxes ULS offers a more rigid *Sbox_no*. A really low value will indicate a more rigid *Sbox_no*, with really low values (say less than 2) reflecting a static nature of *Sbox_no*. While sandboxes here do not have a theoretical upper limit, the performance will likely decrease when the value of *Sbox_no* increases. Another reason for a high score on *Sbox_no* is when a single sandbox places restrictions on Interprocess communication. JVM is such an example, where in the strictest sense we only have a single sandbox. However, the restrictions (recall from the PB value assigned for JVM) in place ensure the safety of processes executing inside the sandbox, and those executing outside. The permission for external access to services can be selectively obtained for different processes.

5.4.2 Dynamic Policy Enforcement (Dyn_rst):

Figure 5.4 corresponds to the primitive *Dyn_rst*, for all the five

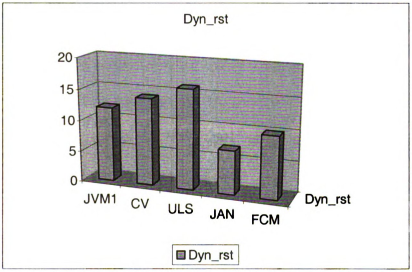

Figure 5.4: *Dyn_rst*

sandboxes that we compare. *Dyn_rst* offers more variation within its class. Clearly we see that Janus offers a lower degree of *Dyn_rst*. As discussed while introducing the primitives in Chapter 2, lower degree *Dyn_rst* is both an advantage as well as a disadvantage. On the positive side, it offers lower scope for damage in case of a flawed policy. ULS offers the highest level of *Dyn_rst*, but cannot be interpreted in isolation, as a significantly strong security measure. We can do that only with the help of the objective for designing the sandbox. A major reason is that ULS, owing to its user level implementation can only constrain user level sandboxes. JVM, CV, and FCM have their values in varying degrees between ULS and JAN. A comprehensive discussion of these features is provided in the section describing the actual classification of the respective sandboxes.

5.4.3 Access Control *(Acc_ctrl)*

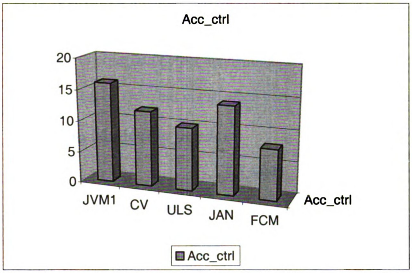

Figure 5.5:*Acc_cntrl*

Figure 5.5 corresponds to the primitive *Acc_cntrl.* Provision of access control is an important security measure. From a security standpoint the higher the value of *Acc_cntrl* the better. However, excessive access control can be detrimental to system performance. Almost all implementations of the sandboxes discussed here, discuss in their implementation the overhead in operating the sandbox. Here we see that FCM has the least amount of static *Acc_cntrl*. However, in practice the value is likely to go up if the set theoretic approach [26] to access control as specified in the implementation is followed. JVM we see offers absolute access control, since JVM does not allow the execution of both trusted and untrusted applets outside the virtual machine. ULS is an interesting case, since while it provides more access control than FCM, when looked at in conjunction with the *Op_mode* primitive, we will find that FCM is the more restrictive of the two sandboxes. The additional restriction in case of FCM is because FCM offers the capability to sandbox kernel Apps, compared ULS which itself is a user level application. But as we have mentioned earlier, we reiterate that *Acc_cntrl* alone is not an indication of a higher performance or security of FCM itself. The issue of higher security ability needs to be looked at from the requirements perspective. A sandbox which only needs to constrain helper applications at the user level can perform well, with respect to either ULS or JAN, but not so when kernel applications come into the picture.

5.4.4 Operation Mode *(Op_mode)*

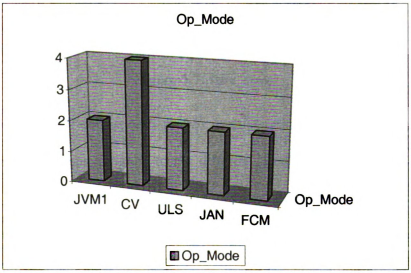

Figure 5.6: *Op_mode*

Figure 5.6 corresponds to *Op_mode*. The range of values, or the NPOD, for *Op_mode* is low and is reflected in the almost identical values for all the sandboxes except CV. As defined in Chapter 2, the operation mode can be either kernel or user level. Here again, differences arise with respect to whether it is possible to switch between the two in one implementation of the sandbox, or necessary to reinstall the sandbox for a different version. ULS for instance does not have the option of executing in kernel level, while FCM can be configured at start time to either execute in one mode.

5.4.5 Inter-sandbox communication *(Isbx_com)*

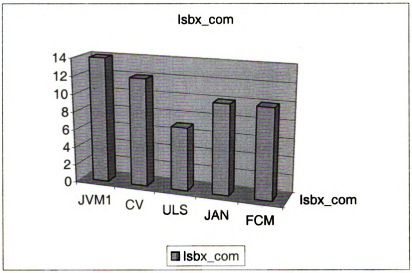

Figure 5.7: *Isbx_com*

Figure 5.7 corresponds to *Isbx_com.* Again; we are able to observe a good variance among the values assigned to the different sandboxes. A lower score indicates among other things, a more rigid means of communication, or even constrained communication. A low score could also be the result of insecure communication between sandboxes. However, we have avoided incorporating the idea while assigning values to limit the scoring to the extent of provision of communication. We observe that ULS provides the least amount of Interprocess communication mechanisms between processes contained within the sandbox and those outside. One reason is that while executing system calls, it will be impossible to exert influence over the actual executing of the system call by the sandbox. The maximum that a sandbox in such a case, can do is to limit the types of system calls that can be executed by the sandbox. JVM offers the highest level of communication between processes within the sandbox and those outside the sandbox. CV, JAN and FCM provide varying degrees of *Isbx_com* in between.

5.4.6 Sandbox Time to Live *(Sbox_ttl)*

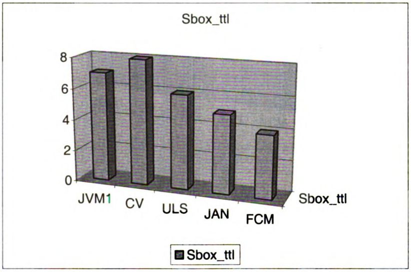

Figure 5.8: *Sbox_ttl*

The following figure corresponds to *Sbox_ttl*. FCM clearly, has the lowest *Sbox_ttl* of the different sandboxes. The positive implication is that the sandbox when not in action will represent lesser overhead. Thus, during execution of either trusted, or safe code, the sandbox could be switched off. However, such a system presents inherent risks along with the flexibility offered. On the other hand we have CV, which exists for as long as the system uptime (Sys_upt). The additional lifetime provides more security, though with additional overhead. JVM is closer to CV, and can be considered a more persistent entity, but the difference is attributed to the fact that JVM is loaded only when necessary. ULS, JAN and FCM take up values in varying degrees in between.

5.4.7 Sandbox Scope *(Sbox_scope)*

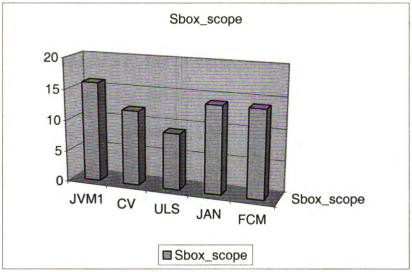

Figure 5.9: *Sbox_scope*

Figure 5.9 corresponds to *Sbox_scope*. We observe that JAN, FCM, and JVM offer a high value for *Sbox_scope*, with JVM offering the highest among these. The implication is that JVM is able to constrain more processes and in more ways than other sandboxes. An applet cannot execute outside the JVM. In the case of JAN and FCM the scope is slightly less restrictive in nature. ULS offers the least *Sbox_scope* which is a direct consequence ULS being executed in the user level. As a result, it will not be able to extend its scope to kernel level applications

5.4.8 Sc/In/NS

Figure 5.10 corresponds to a combination of three Boolean primitives, with only one NPOD. Thus we have combined *Nsss, Int_op, and Scl* into one graph. These three primitives have been clubbed together as they are not as essential to the definition of a sandbox as the other primitives. However, scalability, interoperability and using native security services are good features to have. Clearly, CV offers the lowest level of *Sc/In/Ns*. JVM performs the best in terms of both being scalable, and as well as being interoperable. ULS, JAN and FCM are almost close to each other with respect to *Sc/In/Ns*.

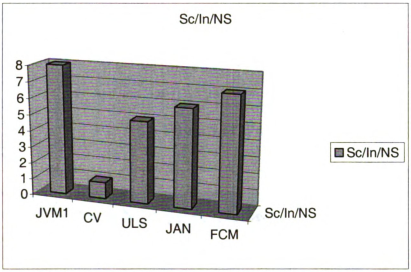

Figure 5.10: *Sc/In/Ns*

5.5 Primitive Subset Comparison

In order to illustrate the difference between the most important primitives we differentiate them into two different groups, and try to observe how the presence or absence of one can affect the overall differentiation among the primitives. We pick the two groups randomly.

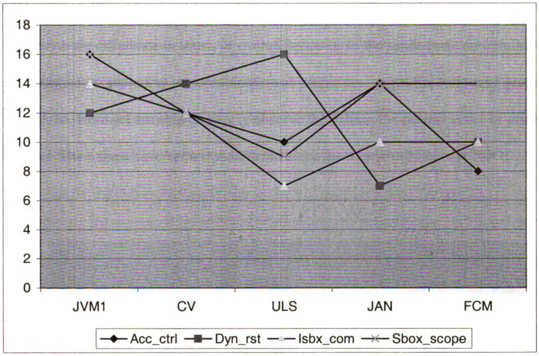

Figure 5.11: Primitive set 1

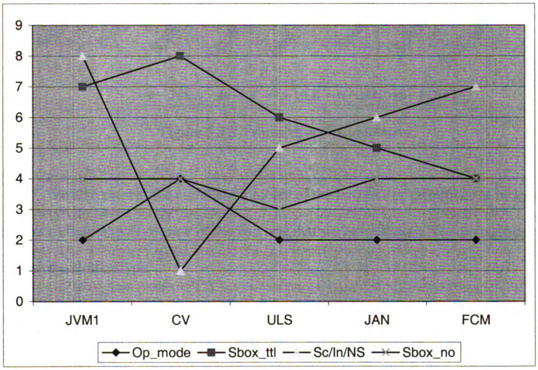

Figure 5.12: Primitive set 2

While in both figure 5.11 and figure 5.12 we observe outliers, and no distinct pattern, and thus very little difference in terms of overall behavior of the sandboxes, combining all the eight parameters including the combination of primitives we get the following graph in figure 5.13. Here we are able to clearly observe the pattern the correct order of primitives gives us. The values have been standardized since the metric used here is NPOD.

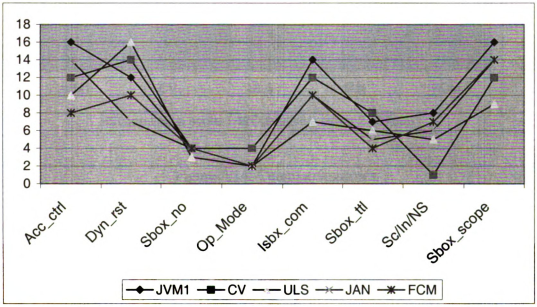

Figure 5.13: Primitive set - Cumulative

We also present the cumulative scores of the two sets of primitives in order to present a comprehensive view of the two sets that we have chosen. We have grouped values of a similar type.

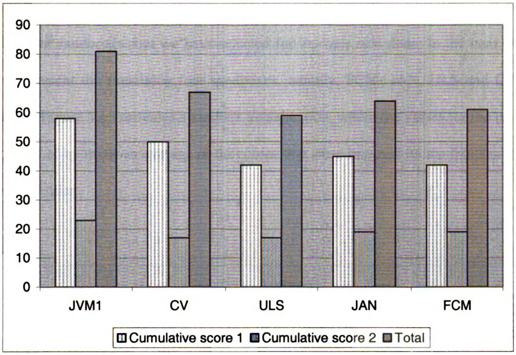

Figure 5.14: Cumulative scores

Figure 5.14 gives the cumulative comparison between the two sets of primitives, and a combined value for the different sandboxes. Assuming that a sandbox can be represented by a Polygon, whose vertices are represented by the different primitives, we will be able to find out a difference between the sandboxes when they are represented as a polygon with the different vertices representing the different primitives. Here again, including or excluding certain primitives will indicate the behavior of the sandboxes with respect to the different combinations of primitives. We will be able to observe them in the figures given in the next chapter along with the individual sandboxes.

5.6 Conclusion

In this chapter, we presented additional information about the visualization of the polygonal representation of sandboxes. We also discussed the translation of policy to sandbox design. A comprehensive primitive-by-primitive comparison between the different sandboxes that we have selected for analysis was done. In the next chapter we will present the remaining four sandboxes, namely, FCM, JAN, ULS and CV in more detail. Figure 5.15 provides a sandbox visualization, where the values for the primitives is arrived at by using an average of the scores that were imposed on the five sandboxes that we consider.

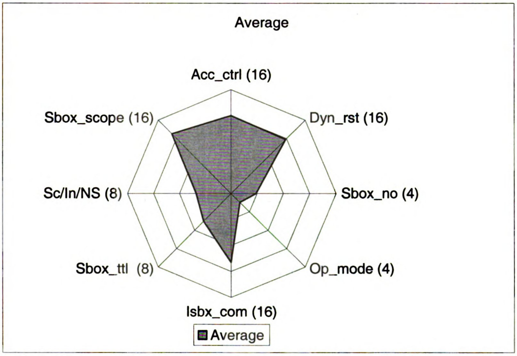

Figure 5.15: Visualization of sandboxes - Average

6. Classification of Sandboxes

In this chapter we provide a detailed analysis of the remaining four (JVM was discussed in chapter 4) sandboxes using techniques that have been presented in the previous chapters. We follow an approach like that in chapter 4. We first describe the fundamental idea behind the sandbox in question, and provide pointers to the actual implementation. We then present an overall diagram representing the sandbox, using our visualization process, along with a summary of the values for the individual primitives. We will also describe the numerical values imposed on each of the primitives. We also provide a brief commentary on the design category of the sandbox. Finally, we present a summary of the values identified for the different sandboxes. We have presented Chakravyuha, ULS, Janus, and FCM in that order in Sections 6.1 though 6.5

6.1 Chakravyuha

A sandbox Operating system for environment for controlled execution of alien code from IBM [34], Chakravyuha (CV) addresses the same problems addressed by other systems such as the JVM. At the core of the CV is a Resource Control List (RCL), which is a set of permissions and resource access privileges, which can be specified dynamically. The untrusted code and the RCL are verified by a trusted third part, before delivering them to a client. The client also gets the option of implementing either all or a subset of the RCL, based on a negotiated protocol for privileges. Figure 6.1 (adopted directly from the paper on Chakravyuha [34]) describes CV. We will explain the functionalities of the CV with respect to the primitives that we have identified.

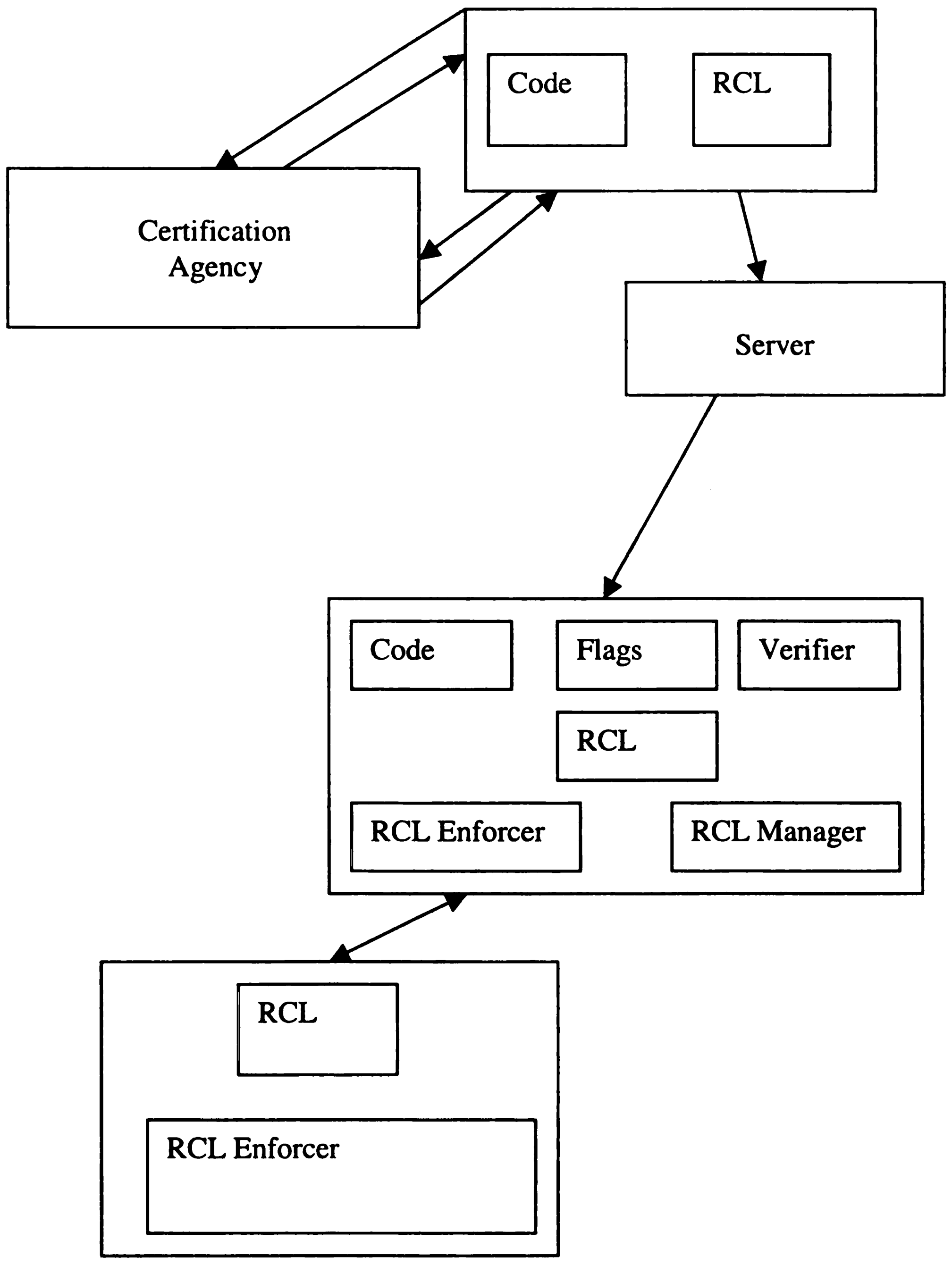

Figure 6.1: Chakravyuha system

6.1.1 Summary of results

In figure 2, we have provided the sandbox based on our visualization scheme. One important feature that stands out is the lack of CV's ability to scale or be interoperable with other systems, or utilize native security services effectively. On all other parameters the behavior of the sandbox is on expected lines, based on the average that was represented in Chapter 1 and Section 5.6. In Sections 1.2 through 1.8. In Section 1.9 we discuss the sandbox design category (described in Chapter 3).

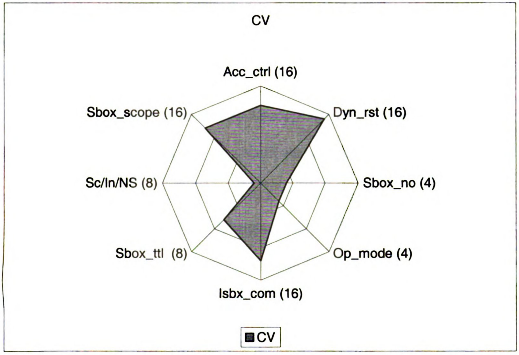

Figure 6.2: Chakravyuha visualization

6.1.2 Access Control *(Acc_ctrl)*

We have provided the value of PBA/ Uproc – PBA/ UTproc for *Acc_cntrl*. The RCL enables the policy based *Acc_cntrl*, and is uniform for both Uproc and UTproc. We have imposed a numerical score of 12, which is a relatively high score for *Acc_cntrl*. The fact that Uproc and UTproc are not differentiated serves to reduce the value of *Acc_cntrl*.

The CV has *Acc_cntrl* at its core, in the form of the Resource Control lists. The resource control lists perform the trust and authentication necessary for the code before delivery to client services. The RCL uses a public key cryptography [35] based system for enforcing the verification of code, resembling a system such as one that uses proof carrying code [36]. The RCL is combined with the code and a digital signature, which in turn is verified with the certificate issued by the certifying authority. A client can then verify the two in order to determine access to system resources. The access control modules can thus be classified into the client stub and the server stub, with the two components together providing the necessary verification.

6.1.3 Dynamic Policy Enforcement *(Dyn_rst)*

We have assigned a value of DH/ Uproc -DH/ UTproc for *Dyn_rst* for CV. CV provides a higher dynamic *Dyn_rst*. As a result we have assigned a higher value of 14 for *Dyn_rst* for CV.

The policy can be translated dynamically through the RCL. Modifications to the RCL are the means to enforce policy changes in the system. One problem the system encounters is that the policy cannot be enforced once a third party client has accessed the code/RCL combination. To overcome the problem of enforcing policy while a third party client has already accessed the code/RCL combination, CV has used a method to split the RCL into different parts, such as capabilities for specifying physical resources, and capabilities for logical resources such as files and network ports. The access itself is controlled by methods that operate on the various part of the RCL. The RCL can be changed in parts, thus allowing for dynamic change in policy representations.

6.1.4 Sandbox Numbers *(Sbox_no)*

We have provided a value of D/PB for *Sbox_no* for CV. D/PB is a common value found among most sandboxes. Since D/PB is also the highest value that can be assigned for *Dyn_rst*.

The number of sandboxes is not limited here. The definition of a sandbox is dependant on the changes presented to the RCL. Therefore, the number of sandboxes is not confined to any specified number. In CV the sandbox itself is a logical entity in the form of RCL, and only restrictions are placed on the execution of the processes, while there are no guarantees to the processes and applications. However CV allows the flexibility to fix a static minimum and a static maximum number of sandboxes to be hardwired during development or installation time.

6.1.5 Operation Mode *(Op_mode)*

We have assigned a value of SboxC/KL – App/UL for CV. A numerical score of 4, which is the highest possible value for *Op_mode,* has been provided. *Op_mode* for CV can be categorized into two parts. First part (App) represents the RCL that is changed dynamically, and accompanies the code in the form of a signature. The second part is the sandbox components (SboxC) like the code verifier which check the signature provided with the code and the certificate from the CA. The RCL operates in the user mode, to provide flexibility to the user, while the sandbox components function in the kernel mode, and behave more like OS components. Thus the operating mode of the RCL based sandbox itself is only a user level, while several components that enable the execution of the sandbox work in both user and kernel level. A typical example of a process in the kernel level is the component for modifying the system call interface.

6.1.6 Inter-sandbox communication *(Isbx_com)*

We have assigned a value of MISC/ UTproc for *Isbx_com*. We have assigned a relatively high numerical score of 12.The sandbox here is a logical restriction on access to resources. While the sandbox description does not discuss the topic of interactions between processes contained in different sandboxes, CV developers have provided the flexibility to place additional restrictions on the process being contained. Even then, the idea will be only whether or not a process can communicate with another process, and there is no restriction placed on the communication channel itself. *Isbx_com* will be an area for research for CV, in order to be able to function as a full-fledged security infrastructure and is particularly necessary, in the light of the fact that the system has suggested sweeping changes to Operating System development process. The high numerical score is due to the MISC value for UTproc. However, we also reduce some points since CV does not have any mandatory *Isbx_com* requirements for Tproc.

6.1.7 Sandbox Time to Live *(Sbox_ttl)*

Sbox_ttl has been assigned a value of Sys_upt referring the fact that CV continues to exist throughout the duration the operating system executes. The score imposed (8) is also the highest for *Sbox_ttl*. The restrictions placed here live for the lifetime of the code. Since changes are being made to the fundamental behavior of the operating system and the protocols for communication between trusted and untrusted system components the RCL and the signature stays in place. However, the sandbox can cease to exist if the RCL itself does not have any entity in it.

6.1.8 Sandbox Scope *(Sbox_scope)*

The value for *Sbox*_scope is L/ Tproc - G/ UTproc. The scope of the CV is global because all untrusted code must have the RCL and the signature verified against the certificates. For Tproc the scope is local, and we have therefore assigned a numerical score of 12 which accounts for the G value for UTproc and the reduction (of 4 from maximum value of 16) accounts for the L value for Tproc. A summary of the features of the CV is given in table 6.1.

Table 6.1: Chakravyuha summary

Features Available	**Available**	**Value**
Acc_ctrl	PBA/ Uproc – PBA/ UTproc	12
Dyn_rst	DH/ Uproc -DH/ UTproc	14
Sbox_no	D/PB	4
Op_Mode	SboxC/KL – App/UL	4
Isbx_com	MISC/ UTproc	12
Sbox_ttl	Sys_upt	8
Sbox_scope	L/ Tproc - G/ UTproc	12
Scl	N	1
Int_op	N	
Nsss	N	

6.1.9 Design Category

CV will be a clear case of layered design category. There are well-defined units and services provided by the different units built on top of other services. There is also a linear relationship between the different units. At the most fundamental level is the certification agency that will be responsible for the certification of various programs and their signatures. On top of the certification agency, the code production system works to ensure that the certificates and the digital signatures are compatible. The server, which is seen as the provider of services which need to be secure uses the services of the code production system. The client in turn uses the services of the server. At the top of the layer, we find that the client also interacts with the database maintaining the RCL which forms the core of the CV security mechanism.

However, within the individual layers, the implementation has been on a component based approach. We refrain from classifying the whole process as a component based approach because any one layer does not provide the services sufficient to contain any or all part of a specific functionality. It is only with the cooperation of all the layers, and the service provided thereof, that a sandbox could be visualized.

6.2 User-level resource constrained Sandboxing

In this section, we describe analyze a user level Sandboxing mechanism put forward in [37]. All our discussion about the sandbox will be based on the description presented in [37]. The system is directed towards Sandboxing in shrink-wrapped Operating systems, where access to propriety code is denied. The sandbox instead builds on other operating system components such as a CPU/process monitoring tool to monitor applications in the system.

6.2.1 Summary of results

From figure 6.4, we are able to interpret the meaning of the sandbox compared to the ideal sandbox. Clearly we see that ULS is one of the smallest in terms of area covered. The small size implies that there must be a limiting factor(s) for ULS. One prominent limitation is that the sandbox is constrained to execute only in the user level.

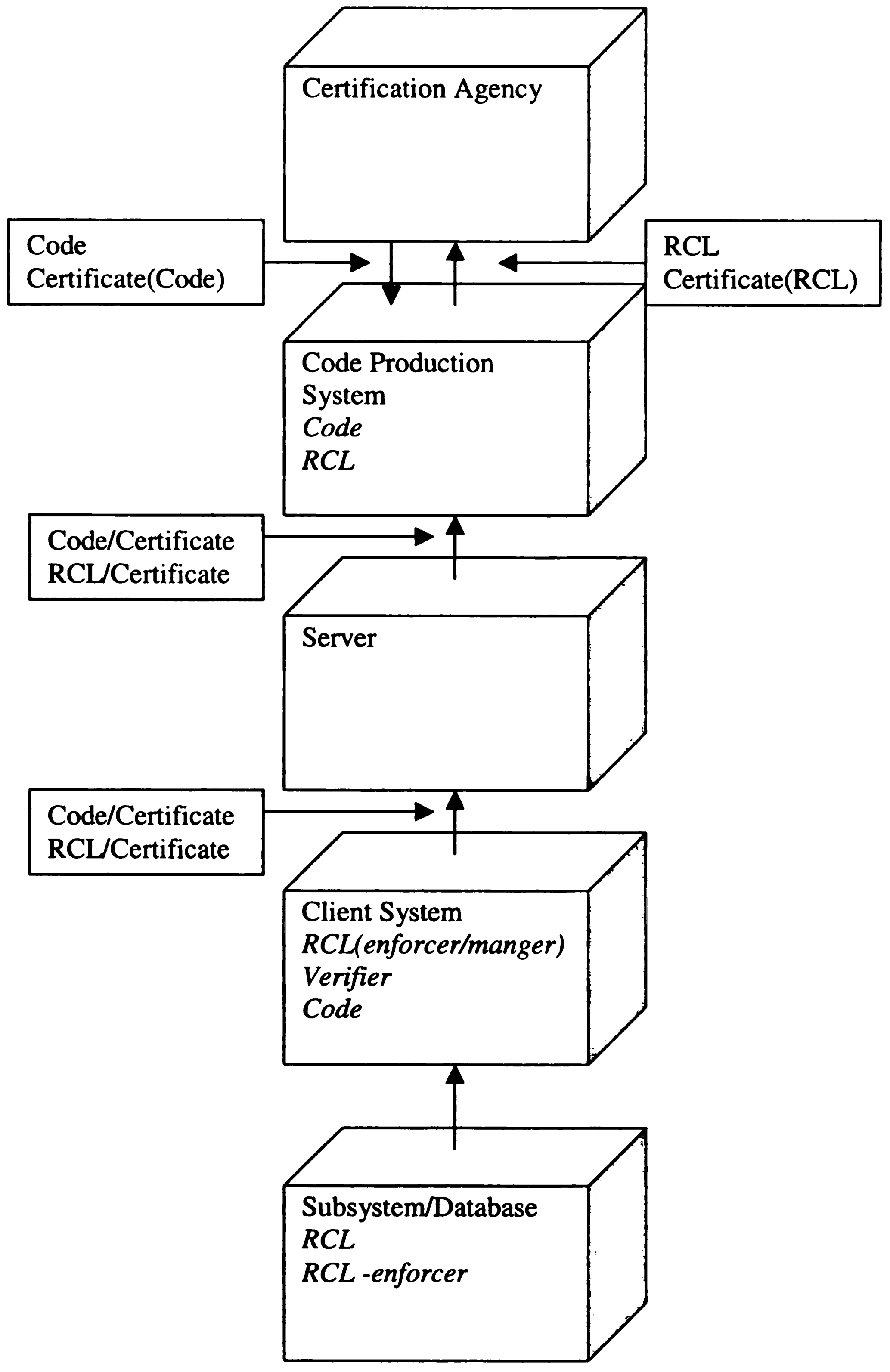

Figure 6.3: Chakravyuha – Layered design category

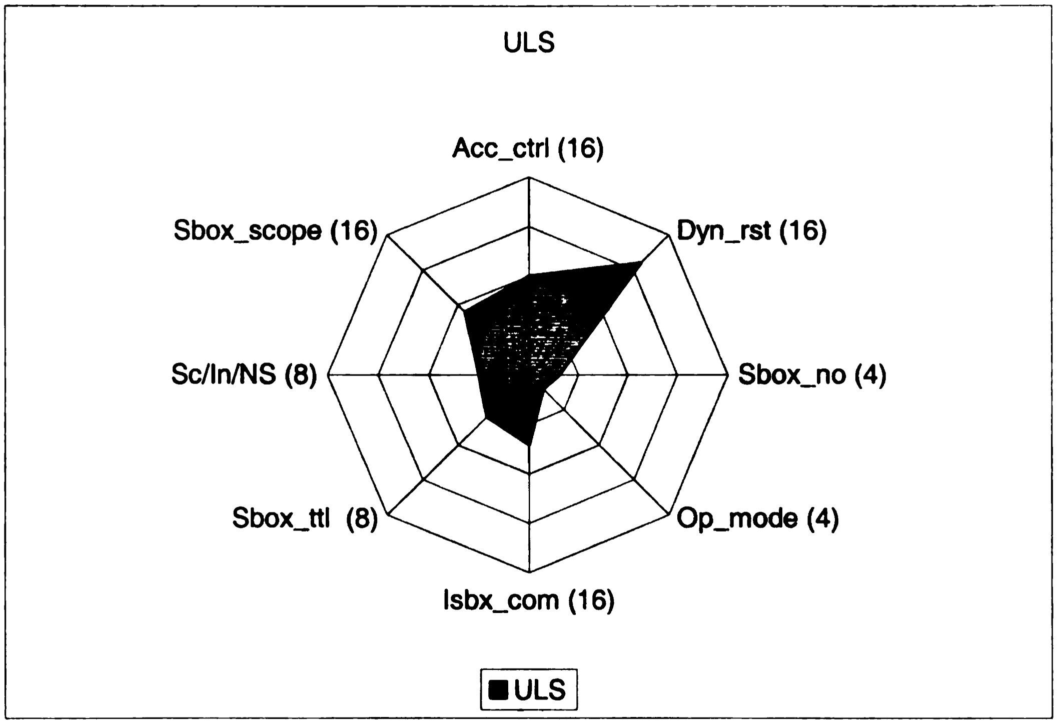

Figure 6.4: ULS visualization

The user level constraint in turn has a cascading effect on the other primitives. But ULS can translate policy during runtime; we see a higher value for *Dyn_rst*. We will present a feature-by-feature discussion in this section. Finally we will present the tabulation of the individual values with the final score.

6.2.2 Access Control *(Acc_ctrl)*

We have assigned a value of PBA/ Tthread – PBA/ UTthread for *Acc_cntrl*. *Acc_cntrl* is defined at the thread level, which results in greater overhead, even while additional security is provided. Therefore we impose a reduced value of 10 for *Acc_cntrl* for ULS. The sandbox uses an access control mechanism such as explicit restriction of physical resources in direct contrast to the approach taken in CV. *Acc_cntrl* has been divided into quantitative and qualitative restriction on the use of resources. The former represents restrictions such as limited access to specified portions of the files system [37], or restricting the use of some files. The latter represents the restrictions placed on accessing certain amounts of different components in the operating system. Another example will be restricting the process to consume not more than a certain specified percentage of the CPU.

6.2.3 Dynamic translation of policies *(Dyn_rst)*

We have provided a value of DH/ Tthread -DH/ UTthread for *Dyn_rst* since both trusted and untrusted threads are treated in the same manner in the case of *Dyn_rst*. Due to the dynamic nature we have assigned the highest score of 16 for ULS. While ULS design does not mention any specific aspect of the *Dyn_rst* that will be necessary to translate the policies in real-time, the design poses no hindrance for the use of such a mechanism. For example, a simple scripting tool can accomplish the task of changing the values that represent the quantitative restrictions, while updating a file to make changes to the qualitative restrictions can also be accomplished in a similar manner.

6.2.4 Sandbox Number *(Sbox_no)*

We have assigned a value of S/PB for ULS. ULS is more at a primitive level, and does not discuss the aspects of multiple sandboxes. However, based on the design, the ULS does allow for multiple policies to exist for different applications. Since we view the sandbox here as a logical entity, they certainly can be viewed as multiple sandboxes. Due to the ambiguous nature of *Sbox_no* specification in ULS we have provided a value of 3, which is lower than the that for all other sandboxes.

6.2.5 Operation Mode *(Op_mode)*

We have provided a value of SboxC/UL – App/UL for *Op_mode*. Clearly, one of the stated goals of ULS is to provide security against external threats, while minimizing dependency on the operating system for the installation of such security specific components. The Op_mode of ULS is directly in relation to the design objectives for ULS, which is to minimize or completely eliminate the need for integrating sandbox code with the operating system kernel. ULS operates in the user level throughout the lifetime of the system. Based on the discussion in Section 2.5, we have assigned a value of 2 for ULS.

6.2.6 Inter-sandbox Communication *(Isbx_com)*

We have assigned the value of S/mbox / UTthread for *Isbx_com* for ULS. A secure mailbox type communication can be visualized for untrusted threads. Since the possibility of different or multiple independent sandboxes is not available with ULS, the issue of inter sandbox communication is not discussed extensively. ULS, based on its specification is definitely extensible to independent implementations that support different values of *Isbx_com* discussed in Chapter 2. We have provided a score of 7 for *Isbx_com* which is also the lowest among all the sandboxes that we have considered.

6.2.7 Sandbox Time to live *(Sbox_ttl)*

Sbox_ttl is assigned the value Prcs_lt/App_lt. That is the sandbox lives until the process or the application lives. However, a direct consequence of the presence of the dynamic translation of policies is that the sandbox can cease to exist due to a user action. The abrupt termination of ULS as a consequence of high *Dyn_rst* presents a vulnerability to the systems, wherein, a malicious process that gains access to the critical files maintaining the user policy can sabotage the entire sandbox mechanism, by changing the policy. Therefore we have assigned a value of 6 for *Sbox_ttl* for ULS.

6.2.8 Sandbox Scope *(Sbox_scope)*

We have provided the value of L/ Tthread - L/ UTthread for *Sbox_scope* for ULS. We consider *Sbox_scope* to be limited to local measures. While global access is still possible, very rarely can they be implemented for reasons of complexity and reducing complexity at the implementation level is one of the design goals of ULS. The primitives for ULS is summarized in table 6.2.

Table 6.2: ULS summary

Features Available	**Available**	**Values**
Acc_ctrl	PBA/ Tthread – PBA/ UTthread	10
Dyn_rst	DH/ Tthread -DH/ UTthread	16
Sbox_no	S/PB	3
Op_Mode	SboxC/UL – App/UL	2
Isbx_com	S/mbox / UTthread	7
Sbox_ttl	Prcs_lt/App_lt	8
Sbox_scope	L/ Tthread - L/ UTthread	9
Scl	N	5
Int_op	N	
Nsss	Y	

6.2.9 Design Category

The architecture for ULS can be classified into a layered architecture. The primary reason for doing so is that the sandbox is not an isolated entity, but depends to a large extent on the services provided by the operating system and related components such as the system (operating system) monitors. We can visualize the sandbox as riding on a basic set of services, which are not a part of the sandbox itself. The Basic service provided by the kernel is used by the system call interceptors, which in turn are used by utilities like the system monitors, whose services are used two layers of the sandbox. The first layer is responsible for the specification of the policy while the second is responsible for the enforcement of the policy.

6.3 Janus – A secure environment for untrusted helper applications

Named after the roman god of the entrances and the exits, Janus (JAN), presents a secure environment for untrusted helper applications. Janus is presented in [39], and we base our discussion on the sandbox presented here. Janus works like a sandbox, usually of helper applications (such as the applets that are loaded for a browser to perform certain actions). The original implementation was based out of Solaris operating system. Janus is developed around a threat model based on web-based applications. The threat model here is that the applications themselves are trusted, but use untrusted components, which are the helper applications referred to earlier. Janus is based on the idea that "*an application can do no harm if its access to the underlying operating system is restricted appropriately*" [39]. We will discuss the Janus sandbox based on our primitives and provide a brief discussion on the architecture classification of the sandbox also.

6.3.1 Summary of results

As shown in figure 6.5 we see that Janus outperforms the best sandboxes in different categories such as *Acc_cntrl*, and *Sbox_scope*. Janus shows behavior resembling the average case scenario for all other primitives. Janus also offers significant scope for *Scl*, *Int_op* and *Nsss*. As presented in Figure 4, Janus has a layered architecture for sandbox design. In later sections we will provide a primitive-by-primitive analysis of Janus. We have finally presented a tabulation of the different primitives and the final scores associated with each.

6.3.2 Access Control *(Acc_ctrl)*

We have assigned the value of UA/ TApp – PBA/ UTApp for *Acc_cntrl* for Janus. Janus presents an *Acc_cntrl* module. *Acc_cntrl* in Janus works exactly like ULS, but in a different operating environment. The fundamental idea is the same. There is a component (specifically a file) that will describe the policy with respect to how untrusted applications can access the underlying resources. Restrictions in Janus are more rigid, however, in that the sandbox entirely deals with web based helper applications which represent the high threat level concentration. We have assigned a score of 14, which is relatively high for *Acc_cntrl* for Janus because UTApp is provided with PBA. Since UA is provided for TApp, we reduce (2 points) from the maximum value of 16 that is possible for *Acc_cntrl.*

6.3.3 Dynamic translation of policies *(Dyn_rst)*

We have provided the value of SL/ TApp-SH/UTApp for *Dyn_rst.* For *Dyn_*rst, since the policy is specified in a separate module, changes can be made dynamically and Janus will use the most updated version of the policy at all points of time. Therefore, we have provided a value of 7, which is the lowest among the sandboxes that we have compared.

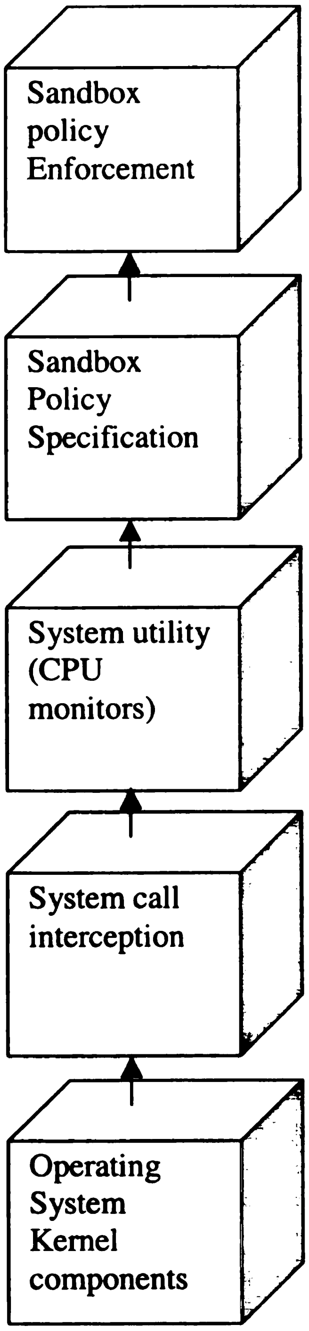
Sandbox policy Enforcement
Sandbox Policy Specification
System utility (CPU monitors)
System call interception
Operating System Kernel components

Figure 6.5: Janus block diagram

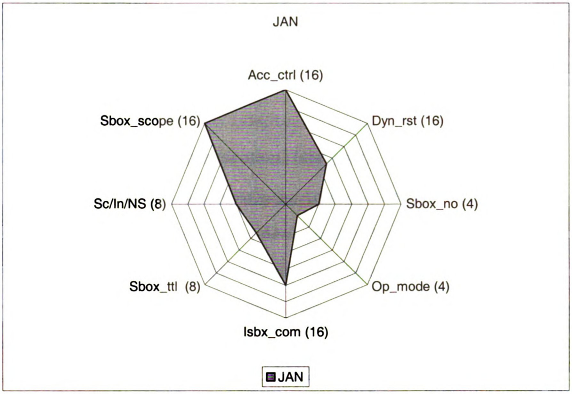

Figure 6.6: Janus - visualization

6.3.4 Sandbox Numbers *(Sbox_no)*

The *Sbox_no* in this case is dynamic and therefore we assign the value of D/PB. The modularized representation of the sandbox policy enables Janus to restrict different classes of processes in different ways. Thus several sandboxes can simultaneously exist. Multiple sandboxes is also enabled by the fact that the overhead resulting due to the different number of sandboxes is reduced significantly because of the modular representation, wherein unused components can be taken away from a sandbox for the different uses. We have assigned a score of 4, which is the highest for *Sbox_no*, for Janus.

6.3.5 Operation mode *(Op_mode)*

We have assigned the value of SboxC/UL – App/UL for *Op_mode* for Janus. Janus operates in the user level, even though some of it's components make use of system call interceptions. These are unobtrusive interceptions, and become obtrusive only when necessary, i.e., when the system security, as represented in the policy, might be violated. We have assigned a score of 2 for *Op_mode* for Janus, which is also the many other sandboxes (JVM, CV, FCM).

6.3.6 Inter sandbox communication *(Isbx_com)*

We have assigned a value of P/Comm. /UTApp for *Isbx_com* for Janus. *Isbx_com* is an issue that is not addressed in the development of Janus. The Sandboxing system here is seen more as a means of isolating individual suspects on a case-by-case basis, rather than in a generic manner and has the advantage that the overhead of providing separate communication channels to processes contained within the sandbox is done away with. The disadvantage is that once the sandbox is breached, either because of a policy flaw or an unknown exploit, the sandbox is considered breached. We have assigned a score of 10 for *Isbx_com* since policy based communication is provided only for untrusted applications (UTApp), while nothing is mentioned for TApp.

6.3.7 Sandbox time to live *(Sbox_ttl)*

We have assigned the value of Prst/UTApp for *Sbox_ttl.* The sandbox time to live here is the system lifetime. While options have been provided to disable the sandbox, anytime the user wants, when applied to a process, it (the sandbox) stays on for the system lifetime. Changes in policy, in a dynamic fashion, can result in there being a zero policy system, which effectively is equivalent to disabling the sandbox; however, the system is still (theoretically) in place as a functioning sandbox. We associate *Sbox_ttl* as persistent (Prst) rather than Sys_lt, since the former encompasses the latter. Since there is a possibility of disabling the sandbox we have assigned a lower score of 5 (compared to the highest value of 8).

6.3.8 Sandbox Scope *(Sbox_scope)*

We have assigned a value of L/TApp - G/UTApp for *Sbox_scope* for Janus. *Sbox_scope* for Janus is for all user level processes, which clearly indicates a global (G) value. However at the same time kernel level processes cannot be sandboxed. Therefore

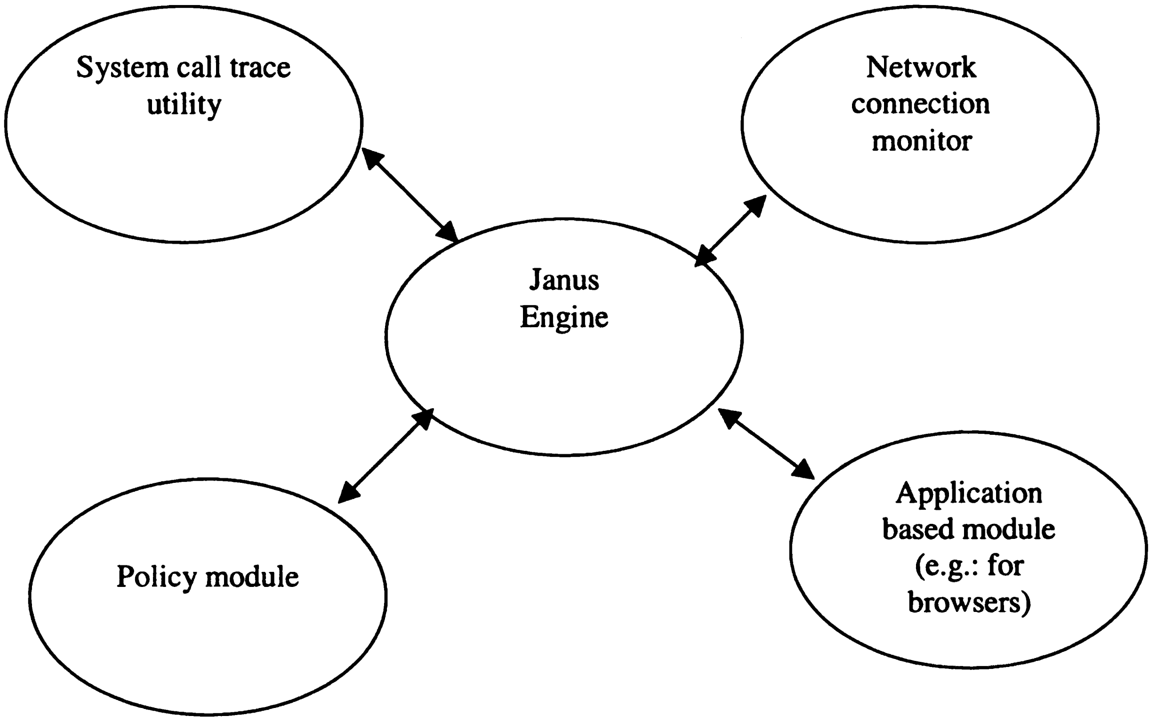
System call trace utility
Network connection monitor
Janus Engine
Policy module
Application based module (e.g.: for browsers)

Figure 6.6: Janus Components

we have assigned a high score of 14, but reduced 2 points from the maximum possible score of 16. A summary of the primitives for Janus is given in table 6.3.

6.3.9 Design Category

Based on our observations with respect to the primitives, Janus, clearly falls under the component based architecture. The architecture is illustrated in figure 6.6. Major components include, a system call trace facility, a path tracing utility, a network component utility, and utilities based on different applications. The Janus engine is at the core driving the independent components.

Table 6.3: Janus Summary

Features Available	**Available**	**Values**
Acc_ctrl	UA/ TApp – PBA/ UTApp	14
Dyn_rst	SL/ TApp-SH/UTApp	7
Sbox_no	D/PB	4
Op_Mode	SboxC/UL – App/UL	2
Isbx_com	P/Comm. /UTApp	10
Sbox_ttl	Prst/UTApp	5
Sbox_scope	L/TApp - G/UTApp	14
Scl	Y	6
Int_op	N	
Nsss	Y	

6.4 A Flexible containment mechanism (FCM)

6.4.1 Summary of Results

The FCM is based on an open source implementation defined in [26]. As the name suggests, the method allows for flexible specification of policies. The FCM is almost identical to Janus both in design and implementation. Thus we will provide a longer summary of results, and avoid topic wise discussion for the primitives. Figure 7 shows the sandbox when represented using our visualization process.

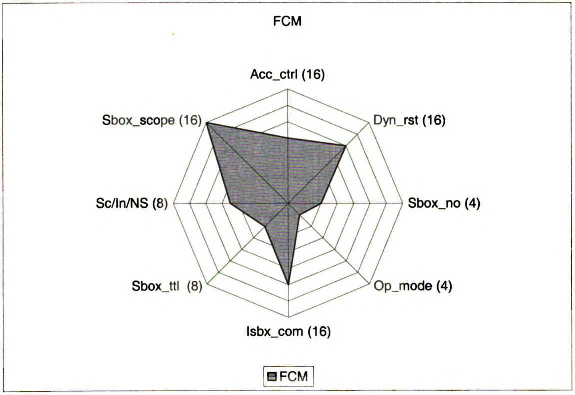

Figure 6.7: FCM - visualization

FCM [51], represents a sandbox designed to allow a high degree of *Dyn_rst*. FCM provides a system call driven sandbox mechanism, wherein the sandbox architecture in reality contains an API. A process creates a sandbox by invoking and later applying to itself a custom designed system call. The process then is provided with system calls to actually configure the sandbox. Custom designed system calls differentiates FCM from other sandboxes. The API also provides applications using FCM with the ability to sandbox child processes independently.

Table 6.4 : FCM summary

Features Available	**Available**	**Values**
Acc_ctrl	PBA/ TApp – PBA/ UTApp	8
Dyn_rst	SL/ TApp-DH/UTApp	10
Sbox_no	D/PB	4
Op_Mode	SboxC/KL – App/UL	2
Isbx_com	P/Comm. /UTApp	10
Sbox_ttl	App_lt/UTApp	4
Sbox_scope	L/TApp - G/UTApp	14
Scl	Y	7
Int_op	Y	
Nsss	N	

In the design of FCM, parent processes retains significant control over sandboxes of its children. The parent has the rights to decide on how many processes can execute within the child sandbox. However, the sandboxes cannot restrict other process. They only restrict themselves.

FCM also enables descendants to inherit it's own sandbox. FCM is designed as a transient entity. The implementation of FCM is more in tune with a component-based approach, rather than a layered approach. This is clearly seen from the individual components that the system has including separate systems for *Isbx_com*, and *Acc_cntrl*. Table 5 summarizes the results for FCM.

Table 6.5: Summary of all sandboxes.

Features Available	**JVM**	**CV**	**ULS**	**JAN**	**FCM**
Acc_ctrl	PBA	PBA/ Uproc – PBA/ UTproc	PBA/ Tthread – PBA/ UTthread	UA/ TApp – PBA/ UTApp	PBA/ TApp – PBA/ UTApp
Dyn_rst	DH/ TApp-DH/UTApp	DH/ Uproc - DH/ UTproc	DH/ Tthread - DH/ UTthread	SL/ TApp-SH/UTApp	SL/ TApp-DH/UTApp
Sbox_no	D/PB	D/PB	S/PB	D/PB	D/PB
Op_Mode	SboxC/UL – App/UL	SboxC/KL – App/UL	SboxC/UL – App/UL	SboxC/UL – App/UL	SboxC/KL – App/UL
Isbx_com	MISC/TApp – MISC/UTApp	MISC/ UTproc	S/mbox / UTthread	P/Comm. /UTApp	P/Comm. /UTApp
Sbox_ttl	Prcs_lt	Sys_upt	Prcs_lt/App_lt	Prst/UTApp	App_lt/UTApp
Sbox_scope	G/TApp - G/UTApp	L/ Uproc - G/ UTproc	L/ Tthread - L/ UTthread	L/TApp - G/UTApp	L/TApp - G/UTApp
Scl	Y	N	N	Y	Y
Int_op	Y	N	N	N	Y
Nsss	Y	N	Y	Y	N

Table 6.6: Summary of scores for all sandboxes

	JVM1	**CV**	**ULS**	**JAN**	**FCM**
Acc_ctrl	16	12	10	14	8
Dyn_rst	12	14	16	7	10
Sbox_no	4	4	3	4	4
Op_mode	2	4	2	2	2
Isbx_com	14	12	7	10	10
Sbox_ttl	7	8	6	5	4
Sc/In/NS	8	1	5	6	7
Sbox_scope	16	12	9	14	14

6.5 Conclusion

In this chapter, we applied our classification scheme to different sandboxes and observed the differences between the sandboxes in terms of the primitives that we have defined. Table 6.5, and Table 6.6 give the summary of the values and scores assigned to the individual primitives Clear differences can be observed in the sandboxes, and a relationship between the objectives and policies of the designers could be seen through the above table. Such tabulation will help in obtaining a birds eye view of the requirements for a new sandbox, enabling easier translation between policies and sandbox design for future designers of sandboxes. The study in this chapter also reinforces our classification scheme with respect to popular sandboxes, from a security perspective.

7. Conclusion and Future Work

7.1 Conclusion

In this thesis, we have discussed why sandboxes are an effective security design mechanism. We found that while there are several independent implementations of sandboxes, there is no way to assess which one to employ for a given security scenario. It will be prudent to take all security measures into consideration during the sandbox design process itself. Thus having an idea of how the implementation of the sandbox will look like at the design stage or perhaps even earlier will be useful. In order to do that we need to visualize the relative effectiveness of a sandbox with respect to other sandboxes while designing the security policy.

We developed a step-by-step classification and visualization process. Firstly we identified important primitives that define a sandbox from a security standpoint. We then identified common implementation paradigms with respect to sandboxes, and provided a brief discussion about the same. Subsequently we classified existing sandboxes and used a scoring scheme to differentiate between them based on the different security parameters. We specifically illustrated an example with respect to JVM; as to how even sandboxes that are close to each other can indeed be differentiated using the classification scheme.

The classification scheme could be used on two fronts. It could be used as a tool to effectively study and analyze existing sandboxes, as we have done in this thesis. Alternately, the results of this study could be used by the designer of a new sandbox to get a prior view of how their design fares compares to other implementations. What is important here is that the designer gets to compare his design with other implementations, before implementation.

7.2 Future Work

We have discussed a new scheme and provided a retrospective analysis on existing sandboxes. Using this framework to develop a new sandbox will be a possible future work. With the added insight into sandbox design, it will be exciting to take on some of the common security flaws or problems such as buffer overruns. We have the power of making decisions and visualizations before implementation. Another area of future work is the development of automated tools to help the scoring process. Automation could be in the form of a rigorous systems test, with specific benchmarks for the different primitives. Incorporating formal specification and verification within the classification framework is another area of future work.

Bibliography

[1] M. Jung and E. Biersack. A Component-Based Architecture for Software Communication Systems. *Proceedings of IEEE ECBS, Edinburgh, Scotland.* April 2000.

[2] D'Souza, D. Francis, Wills, and A. Cameron. Objects, components and components and frameworks with UML approach. Addison-Wesley, 1998.

[3] Trent Jaeger, Jochen Liedtke, Vsevolod Panteleenko, Yoonho Park, and Nayeem Islam. Security Architecture for Component-based Operating Systems *8th ACM SIGOPS European Workshop*, 1998.

[4] Sendmail utility exploits. `Http://packetstorm.linuxsecurity.com /unix-exploits/sendmail-exploits/`

[5] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, and Peat A. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *Proceedings of the 7th USENIX Security Conference*, 1998.

[6] V. Barghavan. Dynamic addressing in Wireless LANs. Proceedings of the *IEEE international communications conference*, 1995.

[7] Mark E. Russinovich, and David A. Solomon. Microsoft Windows Internals, third edition, Microsoft Press, December 2000.

[8] David A. Solomon and Mark E. Russinovich, Inside Microsoft® Windows® 2000, third edition, Microsoft Press, October 2000.

[9] James C. Foster. Buffer Overflow attacks Detect, Exploit, and Prevent. Syngress. 2004.

[10] Topics in Computer Security `http://www.computerworld.com/ securitytopics/security/story/0,10801,89861,00.html?SKC =security-89861`

[11] Wenke Lee, and Salvatore J. Stolfo. Data Mining Approaches for Intrusion Detection. *Proceedings of the 7th USENIX Security Symposium,* 1998.

[12] M. Jones and J. Regehr. CPU reservations and time constraints: Implementation experience on windows NT. In the *Proceedings Of 3rd USENIX Windows NT Symposium*, July 1999.

[13] Eric Bloedorn, Alan D. Christiansen, William Hill, Clement Skorupka, Lisa M. Talbot, and Jonathan Tivel. Data Mining for Network Intrusion Detection: How to Get Started. The MITRE Corporation, 2001.

[14] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth, M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the 5th USENIX Security Symposium*, pages 127–140, June 1995.

[15] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[16] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002.

[17] David Ferraiolo, Janet Cugini, and Richard Kuhn. Role-based access control (RBAC):Features and motivations. In *Proceedings of 11th Annual Computer Security ApplicationConference*, pages 241- 48, 1995.

[18] Ravi S. Sandhu, Lattice-Based Access Control Models, *Computer*, v.26 n.11, p.9-19, November 1993.

[19] R.S. Sandhu, E.J.Coyne, H.L.Feinstein and C.E.Youman. "Role Based Access Control Models" *IEEE Computer*, vol 29, Num 2, p38-47, February 1996.

[20] `http://www.linux.org`

[21] Yasushi Saito and Brian Bershad. A Transactional Memory Service in an Extensible Operating System. *USENIX Annual Technical Conference* (NO 98), 1998 .

[22] `http://www.techtutorials.info/lindistro.html`

[23] Tim Lindholm, and Frank Yellin. Java™ Virtual Machine Specification, The. Addison Wesley Professional, 1996.

[24] Asit Dan, Ajay Mohindra, Rajiv Ramaswami and Dinkar Sitaram. ChakraVyuha (CV): A Sandbox Operating System Environment for Controlled Execution of Alien Code. RC20742, IBM T.J. Watson Research Center, 1997.

[25] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing, *4th USENIX Windows Systems Symposium,* 2000.

[26] D. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," *Proceedings of the 11th USENIX UNIX Security Symposium* pp. 207-225, Aug. 2002.

[27] Dirk Balfanz, and Daniel R. Simon. WindowBox: A Simple Security Model for the Connected Desktop. Microsoft Research, `Http://research.microsoft.com/crypto`.

[28] `http://www.msdn.microsoft.com/netframework/technologyinfo/overview/`

[29] Raju Pandey, and Brant Hashii. Providing fine-grained access control for Java programs via binary editing. Parallel and Distributed Computing Laboratory, Computer Science Department, University of California, Davis, CA.

[30] Cormac Flanagan, Greg Nelson, K. Rustan, M. Leino, Mark Lillibridge, James B. Saxe, Raymie Stata. Extended Static Checking for Java. *Proceedings of the Programming Language Design and Implementation (PLDI)*, Germany, 2002.

[31] Li Gong, Gary Ellison, and Mary Dageforde. Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation. Addison Wesley Professional, second edition, 2003.

[33] `http://java.sun.com/j2se/1.4.2/ docs/api/java/lang/SecurityManager.html`

[34] Asit Dan, Ajay Mohindra, Rajiv Ramaswami and Dinkar Sitaram. ChakraVyuha (CV): A Sandbox Operating System Environment for Controlled Execution of Alien Code. RC20742, IBM T.J. Watson Research Center, 1997

[35] Michael Burrows, Martín Abadi, Roger Needham, and William Stallings. A Logic of Authentication. Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996.

[36] Andrew W. Appel. Foundational Proof-Carrying Code. *16th Annual IEEE Symposium on Logic in Computer Science,* June 2001.

[37] Fangzhe Chang, Ayal Itzkovitz, and Vijay Karamcheti. User-level Resource-constrained Sandboxing. In the proceedings of the USENIX Windows systems symposium, 2000.

[38] Ian Goldberg David Wagner, Randi Thomas and Eric A Brewer. A Secure environment for Untrusted Helper Applications. Proceedings of the 9th USENIX Security Symposium, 2000.

[39] Oracle® Application Server 10g Performance Guide 10g (9.0.4) Part No. B10379-01

[40] `http://bosna.usask.ca/pub/JXTABenchSuite_Report.pdf`

[41] Bill Venners. Inside the Java Virtual Machine. Published by Computing McGraw-Hill, 1998.

[42] Venners, Bill. Inside the Java 2 Virtual Machine. Published by McGraw Hill, 2000.

[43] `http://www-106.ibm.com/developerworks/java/library/j-javaevol/javaevol.html`

[44] `http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/`

[45] Li Gong. A Secure Identity-Based Capability System. IEEE Symposium on Security and Privacy, 1989.

[46] Richard Y. Kain, and Carl E. Landwehr. On Access Checking in Capability-Based Systems. IEEE Symposium on Security and Privacy, 1987.

[47] Samir Djilali, Thomas H´erault, Oleg Lodygensky, Tangui Morlier, Gilles Fedak and Franck Cappello. RPC-V: Toward Fault-Tolerant RPC for Internet Connected Desktop Volatile Nodes INRIA, LRI, Universit´e de Paris Sud, Orsay, France, 2000.

[48] `http://www.microsoft.com/resources/ngscb/`

[49] Mark E. Russinovich, David A. Solomon. Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000. Microsoft Press, 2004.

[50] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, and Jonathan Walpole. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In the Proceedings of the 7th USENIX Security Conference, 1998.

[51] D. Peterson, M. Bishop, and R. Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. *Proceedings of the 11th USENIX UNIX Security Symposium* pp. 207-225, 2002.

[52] Matt Bishop Computer Security: Art and Science Addison-Wesley Pub Co, 2002.

[53] S. Ghosh, A. Gupta. An Exercise in Fault-containment: Self-Stabilizing Leader Election. Information Processing Letters Vol. 59, No.5, pp. 281-288, 1996.