1996

A Policy-Independent Secure X Server

Kirk Joseph Bittler
Portland State University

THESIS APPROVAL

The abstract and thesis of Kirk Joseph Bittler for the Master of Science in Computer Science were presented December 9, 1996, and accepted by the thesis committee and department.

COMMITTEE APPROVALS:

[redacted]

Tom Schubert, Chair

[redacted]

John McHugh

[redacted]

Mike Driscoll, Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]

John McHugh, Chair
Department of Computer Science

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 11 February 1997

ABSTRACT

An abstract of the thesis of Kirk Joseph Bittler for the degree Master of Science in Computer Science, presented December 9, 1996.

Title: A Policy–Independent Secure X Server

This thesis demonstrates that a secure X system can be designed and implemented to be independent of a particular security policy. The advantages and costs of a separation of security policy and enforcement are examined by developing a large scale application, the DX windowing system, on a DTOS platform.

DTOS is a high assurance operating system that isolates policy decisions in a Security Server. A security conscious process, such as DX, eliminates policy considerations from the code. The process instead consults the Security Server and enforces the decisions that server derives from the policy.

The DX architecture is described and its internal design examined. A discussion of X Windows security issues and an evaluation of the DX response is included. The performance of DX is analyzed and future work in the area of secure X systems is considered.

A POLICY–INDEPENDENT SECURE X SERVER

by

Kirk Joseph Bittler

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University

1997

Contents

List of Tables

List of Figures

1. Introduction

This thesis will show that a secure X Windows System can be designed and implemented to enforce a variety of security policies without change to the code. X has become the standard windowing system in Unix environments, so an X system that would support a wide range of the various security policies in use today, without modification to the code, could be used in many different commercial and government settings. The National Security Agency Synergy research project provides an operating system model on which we propose to develop a policy independent secure X system. Synergy is designed to support a large variety of security policies. This is accomplished by separating policy from its enforcement, with the former being localized in a Security Server..

It is widely held that dividing policy from mechanism has significant advantages. Among the benefits are the ease with which policy may be replaced or maintained, the portability of applications, and higher assurance that applications are enforcing the same policy. This separation also has potential efficiency costs in both design and execution. The Synergy architectural model strictly separates security policy from its enforcement by isolating security policy decisions in a Security Server. A security conscious application first contacts the Security Server to ascertain if a particular operation is allowed and then enforces the server's decision. The design appears elegant, yet has not been demonstrated through the construction of significant security sensitive software. This thesis proposes that the Synergy architecture provides a reasonable model for designing a large scale policy neutral application.

The X Window System[1], or X, by design, fully supports sharing of resources and thus is inherently insecure. While most secure systems require some type of isolation so that the limited interaction of processes can be strictly monitored, the X

1. Trademarks: X Window System is a trademark of the Massachusetts Institute of Technology. UNIX is a registered trademark of X/Open.

system promotes the sharing of data and resources among applications. The clients of an X server are meant to cooperate with each other. There are codes of "proper behavior", but no enforcement of them. This makes for an attractive and versatile windowing system in the absence of security concerns. Reconciling this approach, however, with the needs of a secure system is a significant problem.

X is defined by an asynchronous protocol that supports, but does not dictate, graphical user interfaces. An *X server* controls the hardware involved and implements graphics functionality by operating on resources such as windows, fonts, and cursors. *X clients* are application programs that communicate with the X server by requesting operations upon these resources. A client that implements a specific graphical user interface is called a *window manager.*

X is not able to restrict access to its resources. Any resource can be used by any client, whether or not that client was responsible for its creation, by simply supplying the appropriate resource ID in the X protocol request. A malicious client, thus can not only access the entire display region of another client, but can also sabotage other clients in numerous other ways. Indeed, all clients have equal rights — there is no notion of privilege.

By a secure X system, we mean a system that prevents unauthorized information from being disclosed, altered, and ensures the availability of its services, yet allows most current X applications to run without modification. In particular, the following issues are of concern. This list is selected from [4] and represents, in part, an interpretation of issues involved in creating a secure X Windows System. The DX system should provide the means for authentication of connecting clients including clients on remote hosts, visible labeling of windows indicative of sensitivity, a trusted path mechanism to authenticate the system to the user, and elimination of the communication channels created by overlapping windows of different sensitivities. A further objective is to mini-

mize the size of the Trusted Computing Base while keeping its component modules both small and functionally consistent, to ensure that trusted modules do no more than they should.

One system designed with reference to some of these goals is TX. Separating resources of clients at different sensitivity levels is an obvious solution to some problems in securing an X system. The design of the TX system involves separation of the management of shared devices, the screen, the keyboard, and the mouse, from the rest of the X server code. The portion responsible for device management must be trusted to allow access only as prescribed by the security policy. The rest of the code is "polyinstantiated", or replicated once for each active sensitivity level. This ensures that all other X resources are shared only by clients at the same sensitivity level.

Built on the TMach[2] operating system, TX was created with a specific security policy embedded into the code: a DoD style multilevel secure policy. TX was a trusted application created to run on a B3 system. It was targeted to show that such an X system could achieve a B3 application rating. It was not certified as such, but was a prototype for future work.

In order to demonstrate the thesis, a prototype policy independent secure X system, DX, based on the TX design will be designed and implemented on the DTOS operating system.. DTOS is based on the Synergy architectural model. Though the high level TX design is publicly available, we have not had access to the TX code. Developing an internal design for a secure X system , then, is a major objective of this work. Implementing a policy–independent system on DTOS requires recognition of any policy decisions built into the design, removal of these decisions from the system, and the substitution of calls to the Security Server in their place.

2. Trademark of Trusted Information Systems, Inc.

Performance issues inherent in the design and specific to the DX prototype implementation will also be explored.

In order to understand the DX design and its goals certain background material is necessary. Therefore, before presenting the design of the DX system we will provide, in chapter 2, a brief introduction to computer security, and, in chapter 3, an introduction to the X Window System. Chapter 4 describes the TX approach to security folowed by an introduction to the target DTOS operating system in chapter 5. The security problems involved in creating a secure X system are investigated in chapter 6. Chapter 7 describes the design and implementation of the DX system in detail. An analysis of the DX system with the respect to the problems described in chapter 6 follows. There follows an investigation of the performance of the DX system and some possibilities for future work in this area. The paper concludes with a discussion of the Synergy security architecture in practical applications such as DX.

2. Computer Security

2.1 Overview

Computer security is concerned with the protection of data resources from inappropriate disclosure, alteration, or destruction. In addition, the ability to use system resources must be restricted and carefully controlled. The specific needs of a system will vary depending on, among other things, how a system is used, who has access to the system, what resources are available, and the concerns of the system owner. A system's security requirements can be expressed as a "policy" in terms of the following:

> Confidentiality: The assurance that sensitive information is not disclosed to unauthorized subjects.
>
> Integrity: The assurance that data and programs are not destroyed or modified in an unauthorized manner.
>
> Availability: Resources are usable when needed by an authorized user.

Before a security policy is defined the *entities* and *operations* of a system must be identified. Entities are divided into *subjects* and *objects*. A subject is an active computer entity that can initiate requests for resources and utilize those resources to complete some computing task, e.g. processes or process groups executing on behalf of some user. An object is a passive repository used to store information, e.g. files, directories, memory. At times, interprocess communication (IPC) mechanisms result in processes classified as both subjects and objects. A *policy* states under what conditions a subject may operate on an object. *Operations* are types of accesses include reading, writing, deleting, etc.

Certain portions of a system are trusted with enforcing the security policy. These make up the *Trusted Computing Base (TCB)*. The TCB should be protected from non–trusted parts of the system to assure its inviolability.

There are several mechanisms for enforcing confidentiality, integrity and availability. Among them are authentication, access control, penetration analysis, and covert channel analysis.

Access Control comprises those methods of enforcing which subjects are allowed to access which objects. There are two primary classifications of access control. **Discretionary Access Control (DAC)** is the enforcement of user specified access. For example, in a Unix system the owner of a file may set access permissions for owner, group, and world (anyone else). The access permissions are at the discretion of the owner. **Mandatory Access Control (MAC)** enforces access mediation at the discretion of a system administration facility, based on the security attributes of the subject and object in question. For instance, a file may be labeled "top–secret", and a process attempting read it may be labeled "confidential". Under these circumstances read permission is denied.

Authentication consist of the procedures and mechanisms that allow a system to ensure that the stated identity of some external agent is correct. The question asked is, "how do we know someone or something is what it claims to be?" For instance, passwords are an authentication mechanism for users, as are various "smart card" and biometric identification methods. Users may also wish that the system provide a direct means to communicate with the TCB. This can be supported by a **Trusted Path** mechanism. Frequently, a trusted path is implemented using a **Secure Attention Key (SAK)**. When the SAK is pressed, all processes running at the terminal are suspended except the one trusted listener. This provides a guarantee of direct access to the system. No process can persuade the user that it is the trusted system, causing him/her to reveal privileged

information. Such an attempt by a program is called "spoofing". The trusted path only works if the user regularly presses the SAK to begin a dialogue with the system.

There are also assurance techniques available for raising one's confidence in the security mechanisms of the system. **Penetration Analysis** attempts to find the flaws in a system by attacking it. Though useful, it can never prove the absence of problems. Greater assurance can be obtained by subjecting the system to **Formal Verification**. This includes creating both a formal model and a formal mathematical specification. Proofs are used to verify the correctness of the transition from one stage to another. This technique requires significant effort, but is required for some systems.

Covert Channel Analysis attempts to discover and analyze how processes can send information through means other than those intended for data transfer. **Storage channels** are exploited by a sending process changing some system attribute that acts as a "signal" to a receiving process that monitors the attribute. This results in a coded message being transferred. For example, process B can signal information to process A simply by changing the name of a file. As a result, even if process A cannot directly communicate with process B or read a file to which process B writes, A may still be able to see the file. If so, then This simple covert channel may be closed by making the file invisible to process A, but will other channels be introduced? For example, the amount of disk space used by the directory can change. **Timing channels** vary the time required by various operations, using the passage of time as a coding scheme. These are even more difficult to discover. Storage and timing channels, collectively known as **covert channels** are particularly insidious since they avoid the normal access control checks.

One type of program that exploits covert channels is a *Trojan Horse*. Such a program looks like a normal program that appears to work correctly, but with unobserved side effects. For example, a Trojan Horse disguised as a text editor may use a covert channel to pass some of the information entered by a user to an unauthorized process.

The effectiveness of this or any covert channel is typically measured by its "bandwidth" (i.e. bits per second). Since covert channels are difficult to eliminate, a covert channel with a bandwidth of .001 bits/sec may be considered tolerable. Depending on the data in question and the system goals, one bit each thousand seconds could be a disaster. In general, though, the higher the bandwidth, the more likely a covert channel is a cause for concern.

2.2 Security Ratings and the "Orange Book"

In 1983 the Department of Defense Computer Security Center released an official evaluation criteria for secure systems. The 1985 revision of the "Department of Defense Trusted Computer Evaluation Criteria", also known as the "**TCSEC**" or the "Orange Book" (because of the color of its cover), became a standard for evaluating trusted systems in the United States. Since 1985 other official standards have been defined, most notably the 1989 "Canadian Trusted Computer Product Evaluation Criteria", and the 1990 European "Information Technology Security Evaluation Criteria". Many, indeed, feel that the TCSEC is becoming quickly outdated. At least, it may be inappropriate for many new situations such as distributed systems. The "Common Criteria," or "CC", released in 1996, attempts to harmonize the various standards listed above into a new evaluation criteria. The CC promises to become the standard for evaluation criteria. The systems in this paper are closely concerned with the TCSEC rating system.

The TCSEC lists several fundamental requirements for secure systems. The ratings of systems depend on the degree to which these "requirements" are met. They include[6]:

> Security policy – There must be an explicit and well–defined security policy enforced by the system.
>
> Marking – Access control labels must be associated with objects.

Identification – Individual subjects must be identified.

Accountability – Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party.

Assurance – The computer system must contain hardware/software mechanisms that can be independently evaluated to provide sufficient assurance that the system enforces [the four] requirements above.

Continuous Protection – The trusted mechanisms that enforce these basic requirements must be continuously protected against tampering and/or unauthorized changes.

The TCSEC lists four basic security divisions, A, B, C, and D, with some subdivisions. The requirements are monotonically increasing, from lowest D to highest A. If a lower rated division has a requirement, then all higher divisions have it also.

D: A system in this division has been evaluated but fails to meet the requirements for any other division.

C: To be in the C division a system must enforce Discretionary Access Control. This division is intended for single sensitivity level systems. It is subdivided into

C1: A C1 rating requires user authentication and functional tests for assurance of the system.

C2: A C2 rating requires a login procedure, auditing of security–relevant information, and resource isolation.

B: To be in the B division, a system must enforce Mandatory Access Control. It is intended for systems with multiple sensitivity levels. The subdivisions of the B division are as follows:

> **B1**: A B1 rating requires a specific informal security policy statement, sensitivity labels for all data, and mandatory access control.
>
> **B2**: A B2 rating requires a formal policy model, labels for all subjects and objects, a trusted path mechanism for purposes of authentication, covert channel analysis, and a division of protection–critical and protection–non–critical components.
>
> **B3**: In addition to all the previously stated requirements, a B3 rating requires a security administrator and recovery procedures. In addition, all accesses must be through a reference monitor. A "reference monitor" has three requirements: It must be tamperproof. It must mediate every access. It must be small enough to be analyzed completely.

A: To be in the A division, a system is not required to provide additional security functionality than for a B3 rating. The level of assurance, however, must be higher. The requirements include a Formal Model, a Formal Top Level Specification (FTLS), and a proof of their consistency. Also necessary are an informal FTLS to code proof and a formal covert channel analysis.

3. The X Window System

The X Window System is a popular and versatile windowing system. It provides a hierarchy of resizable windows and support for high–performance, device–independent graphics. Windows, cursors, and colormaps are among the abstractions provided to applications.

The X architecture is based on a client/server model of distributed computing. The X server controls the screen (or screens), keyboard, pointer (usually a mouse), and perhaps other input devices. X clients are application programs that communicate with the X server by sending requests to operate on resources maintained by the server. The X server controls the input devices, sending input events to clients as appropriate. Figure 3.1 illustrates the situation.

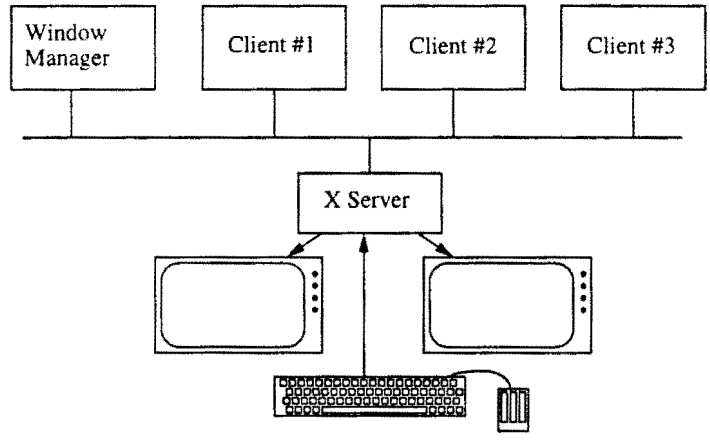


Figure 3.1: X Architecture

3.1 The X Protocol

X is based on the asynchronous X protocol. In fact, X *is* the X protocol. From its definition, an X system can be implemented. The versatility of X comes from the fact that it is defined only by a protocol and not a particular graphical user interface policy. The protocol is asynchronous for higher performance. Since a round–trip mes-

sage sent across a network takes considerable time, not waiting around for responses, which in many cases are not even needed, is much more efficient.

There are three types of X protocol messages:

Requests: an X client may send a request to the X server, consisting of an object id(some X resource such as a window, colormap, or cursor) and an operation (such as drawing a line, changing a color, creating a window, mapping a window — displaying it to the screen, or asking the size of a window).

Replies: the X server returns to a client information from some requests (such as asking the size of a particular window). Replies only are returned when a request asked for information or in response to errors.

Events: the X server sends events to clients informing them of something that they have expressed an interest in. For example, keyboard and pointer input create events. A client *registers* for events of which it wishes to be informed. An `expose` event is generated when a window, previously obscured, is uncovered. It is sent to the client which owns the window. This client is then responsible for sending the requests to redraw it.

Here is an example of a simple X protocol session [3]:

client	X server
Open connection	
	accept connection, reply describing server
CreateWindow request	
AllocColor request	
	reply to AllocColor request
CreateGC request	
MapWindow request	

Expose event

PolyLine request

etc.

3.2 The X Architecture

The use of X has become quite widespread because the protocol implies a device independent nature and because it is free. The X Sample Server may be obtained for no cost over the Internet or from the X Consortium. Vendors optimize, or rewrite, this server for their specific concerns, but the definition of the protocol is controlled by the X Consortium. We also used the sample server as a starting point for the DX system.

The code for the sample server distributed by the X Consortium is divided into three primary layers: the device independent or DIX layer, the device dependent or DDX layer, and the operating system dependent or OS layer.

The DIX layer does not depend on graphics devices, input hardware, or the host operating system. It controls the other parts of the server: the OS and DDX layers. In addition, it is responsible for dispatching client requests, managing the event queues and client data structures, and distributing events to clients.

The OS layer contains the higher level (non–DDX) functions that depend on the operating system. These include listening for client connections and requests, forwarding requests to the DIX layer, and memory management. The OS layer also contains routines that read font data from the font server.

The DDX layer contains procedures that manage the input devices and the graphics hardware. Among its duties are the creation and handling of pixmaps, colormaps, screens, fonts, and graphics contexts. The DDX layer is, itself, split into two layers. The top layer includes the mostly device independent frame buffer code. A *frame buffer* is a section of memory to which the CPU may write directly instead of writing to a graphics coprocessor. It is able to handle monochrome frame buffers (one bit per pixel) or color

frame buffers (2, 8, 16, or 32 bits per pixel are supported). Included are most of the routines that create and manage the frame buffer, pixmaps, colormaps, and graphics contexts.

The lower layer includes the code to initialize and probe input and output devices, as well as any graphics device dependent rendering code. Included in this layer are drivers for different operating systems and some specific graphics devices.

3.2.1 Flow of Control

The **main** routine, in the DIX layer, processes command line arguments, controls initialization of devices, data structures, and communications. Device initialization is done through the routines **InitInput** and **InitOutput** in the DDX layer. The OS layer handles communication set up.

When the server is ready for client requests, **Dispatch** is invoked by **main**. **Dispatch** reads and distributes requests from clients and device input events. If a client has requests ready in its queue, as determined by **WaitForSomething** (in the OS layer), they will be read and dispatched to the appropriate handling routine. Between client processing **Dispatch** handles input events by invoking **ProcessInputEvents** which clears the event queue.

3.2.2 Windows

Each window has a *parent*, which contains it, and possibly *siblings* and *children*. Siblings (children with the same parent) have a *stacking order*. Windows higher in the stacking order obscure lower ones if part of their respective areas overlap. Window attributes are maintained in a **WindowRec** structure, which contains pointers to the parent, the previous sibling in the stacking order, the next sibling in the stacking order, the first child in the stacking order, and the last child in the stacking order. These structures form a *window tree*.

One of the purposes of this organization is to facilitate the process of *clipping*. Clipping ensures rendering is done only to those portions of a window that are vis-

ible. Therefore, each window also has *clip–regions* associated with it. These contain lists of regions of the window that are currently visible. There are separate clip–regions for the window and the window with border. When a window is mapped (displayed to the screen), unmapped, moved, or resized it potentially affects the visibility of other windows. Therefore, when such an operation is performed, the window and border clip regions of any affected window must be updated. This process is called *window tree validation.*

If an area of a window is exposed during one of the operations listed above, after window tree validation occurs, an `Expose` event is generated and sent to its client, which is then responsible for sending the appropriate requests to repaint the newly exposed areas. Clients may, on the other hand, request *backing store* for a window. In this case, just before portions of a window are obscured, the server saves them in off–screen memory, so they may be immediately rewritten in the event of exposure. This may be much more efficient if a window's contents are updated infrequently.

3.2.3 Event Processing

Events fall into two categories: input events and server events. Input events are generated by input from the devices. The pointer, keyboard, and perhaps others, are monitored by the X server, which, upon receipt of data passes control to the respective routine, **ProcessPointerEvent**, **ProcessKeyboardEvent**, or **ProcessOtherEvent**. These routines are responsible for keeping track of what clients should receive notification for what events.

The server sends events to clients that have expressed an interest in them. For a specific event, it discovers what clients these are by traversing the window tree looking for windows for which a client has registered an interest. The server then sends an event notification to that (or those) client(s). There is, however, another way.

Clients sometimes *grab* a device. When this occurs, the input events for that device are sent only to the interested client. Grabs are of two types, each of which may be of two modes, an active or a passive grab. In an *active grab* the client takes over the device immediately. In a *passive grab*, the client specifies a combination of key and button pushes that initiate the grab. Each type of grab may be either synchronous or asynchronous. In a *synchronous grab* the device is essentially frozen until the client allows more events. For an *asynchronous grab* events are queued for later delivery to the client as usual.

Server events, on the other hand, are sent only to the client that created the window in which the events occur. These events are handles by the routine **DeliverEvents**.

4. TX

Trusted X, a prototype secure X system, is designed for the TMach operating system. Both TMach and TX are designed to enforce a standard multi–level secure policy.

4.1 TMach

The Trusted Mach system, or TMach is a highly secure operating system base. It consists of a microkernel based on Mach principles and a set of servers. It is evaluated to a B3 security rating. It may, variously, take on the personality of a Unix, DOS, or, in the future, some other operating system through changing some of the higher level servers.

TMach is a layered system. Each layer depends on the lower for functionality but may not tamper with it. The bottom layer is TMach kernel, which provides the basic abstractions in the same way as does the Mach microkernel (see chapter 5.2.1): tasks, threads, ports, messages, and memory objects, along with others such as I/O. It is certainly security conscious, but does no policy–specific mediation. The kernel provides the means for servers to construct various security functions of the system.

The rest of the TMach trusted computing base consists of a set of servers. These constitute layers above the kernel and provide multi–level secure services for other trusted servers as well as untrusted applications. They include a File Server, an Authentication Server, an Audit Server, a Printer Server, and the Trusted Shell Utilities. The last include a trusted path mechanism. The servers execute as separate tasks with independent address spaces. Communication between server and client is via IPC messages or explicitly shared memory. This isolation allows security properties to be more easily verified. Another server, the Root Name Server, handles all access mediation based on the security policy.

Outside the TCB are the single–level OS servers. These include servers for NFS, sockets, and pipes. Finally, on the top layer are the user processes.

4.2 X on TMach

TX was designed as a prototype multilevel secure X windows system with a target B3 security rating. The premise of the design is the encapsulation of untrusted functionality to limit the size of trusted code, together with polyinstantiation of the untrusted code per sensitivity level. This allows isolation of untrusted portions from their counterparts at other levels.

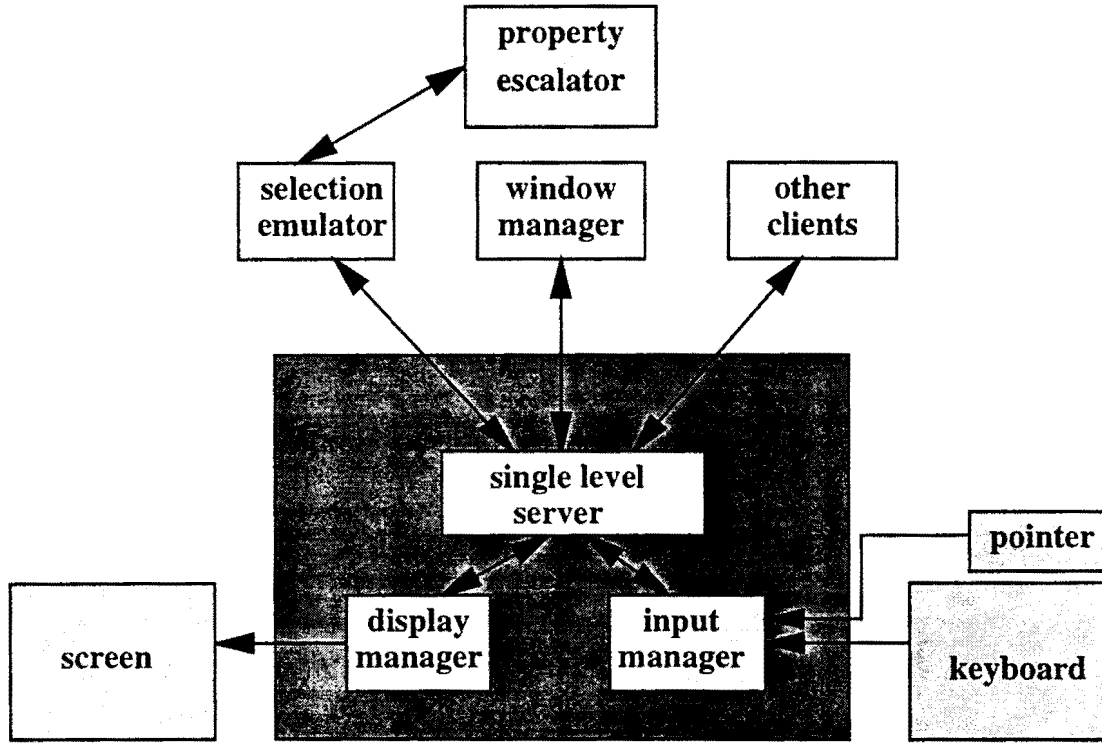


Figure 5.1: TX Architecture

The majority of the X server remains untrusted. The input functionality (keyboard and pointer input) and output functionality (the display), however, require trust because input and output each involve managing data from different security levels. An

Input Manager was created for the handling of keyboard and pointer input, and a Display Manager was created for handling the actual display of information on the screen.

The modified X server, called the **Single Level Server (SLS)**, no longer directly receives input from the keyboard or mouse, but from the Input Manager. Likewise, it no longer writes to the screen, but instead, sends the information to the Display Manager which renders it. The Single Level Server may now be untrusted. It is replicated once per security level and works independently from all of the other instantiations.

The **Input Manager (IM)** is responsible for routing keyboard and mouse input to the correct Single Level Server, or, in response to the Secure Attention Key sequence, to the MiniServer. When an input destination is specified by the user — a Single Level Server, the MiniServer–Trusted Shell, or none, all subsequent input is sent to that server only, until the Secure Attention Key is pressed again.

The **Display Manager (DM)** is the largest and most complicated of the trusted modules. It is responsible for accepting the output of the various SLSs, combining them into a coherent image, and displaying the result. This also requires labeling the windows according to their sensitivity levels. Each window has a label on all four sides, outside any window manager dressing, indicative of the sensitivity level to which it belongs.

For efficiency as well as to minimize the size of trusted code, the DM does not do low–level rendering. Rather, it composes the screen image from "frame buffers" sent in messages from Single Level Servers. The DM keeps track of all top–level windows, i.e. children of the root window. All other visible windows are contained within these. When a SLS maps (or updates) a window, it sends a Mach message to the DM. The message includes information such as the frame buffer (a buffer containing the pixel map for the top–level window), and the position, size and stacking order of top–level windows. The DM maintains all this information for each window. In addition, it records the colormap used at each level, along with the position, image, and color of the cursor.

When the MiniServer instructs the Display Manager to change the active Single Level Server, the correct colormap is installed and the correct cursor image is placed in the position it previously occupied when this level was last active.

When a level is inactive it may not map or unmap windows (i.e. cause them to be drawn to the screen or removed thence). Since the Single Level Servers do not know what the others are doing, mapping a window from an inactive server would not cause a security problem, but it could obscure a window currently in use. Even if placed in the background it could be disconcerting. TX resolves this problem by distinguishing between operations that should occur immediately and ones that can be delayed. The DM has two ports on which it receives requests: the "hold" port and the "non–hold" port. Mapping, unmapping, and some other requests are sent to the hold port. There they are "held" until the server which sent them becomes active. Requests sent to the "non–hold" port are fulfilled in the usual fashion. For example, if a clock were running at an inactive level, it should still be updated.

An area of the screen is reserved for use by the MiniServer to provide users with security state information. The MS sends messages to the DM which are immediately displayed in this area. This is part of the trusted path mechanism. The Single Level Servers know nothing of it.

The **MiniServer (MS)** and **Trusted Shell (TSH)** are concerned with the trusted path mechanism. As has been mentioned, the MS is part of the trusted path. It communicates with the Input Manager and the Display Manager and provides simple buffers for the DM to display. The Trusted Shell interacts closely with the MS, providing it with the information it renders for display by the DM. The TSH is the user interface for certain administrative functions such as creating a new SLS, selecting an active SLS, locking or unlocking the screen, displaying the sensitivity level of a window on the screen, or exiting TX.

So far, all these modules have served to keep clients at different levels separate from each other. Some interaction, though, is desired. In particular, cutting and pasting is a useful and characteristic function of X, but must be mediated by some trusted module. The **Property Escalator (PE)** is an implementation of such a server. The **Selection Emulators (SE)** are untrusted clients that grab cut buffers at their level and pass them on to the PE. Conversely, when a paste is requested at the level of an SE, it forwards the request to the PE. The PE then makes a decision as to what cut is allowed to be pasted to the requesting level. From its store of such cuts, the PE returns to the SE the one it deems appropriate. To reiterate, there is one PE, which is trusted, and multiple untrusted SEs, one per sensitivity level.

These are the primary modules involved in the TX system. There are also a **Server Initiator** that starts up the Single Level Servers when asked to by the DM, a **Client Initiator** that starts the SE, and a Window Manager for each SLS. Once communication has been established between the X client and the SLS, subsequent X protocol messages to pass directly between client and server.

It would be well to note here that the TX system restricts the drawing of "help–lines" by window managers when placing a window on the screen. Instead, the DM renders these lines. An extension to the X protocol was made for this purpose.

In summary, the TX system provides a very solid security design, targeted at B3 for a multi–level secure policy. It does not provide trusted graphics (Single Level Servers are untrusted), but enforces a separation of different sensitivity levels, with only mediated cut and paste interaction. The extra overhead of the various trusted modules does have a negative effect on performance. On the average TX responses are about half the speed of ordinary X.

5. The DTOS Operating System

5.1 The Synergy Model

Synergy is a National Security Agency research project for developing a "distributed, microkernel–based security architecture that will allow for flexibility in the security policies implemented and enforced[14]." This is a much different approach than TMach, which was designed specifically for multilevel secure policies.

The architecture divides functionality into servers allowing each part to be conceptually simpler, and increasing the flexibility of the system. The design also isolates security policy in a separate Security Server with enforcement by security conscious servers. The consequence is a system that can support a wide variety of security policies with relatively minor adjustments. The following servers are included in the original Synergy model.

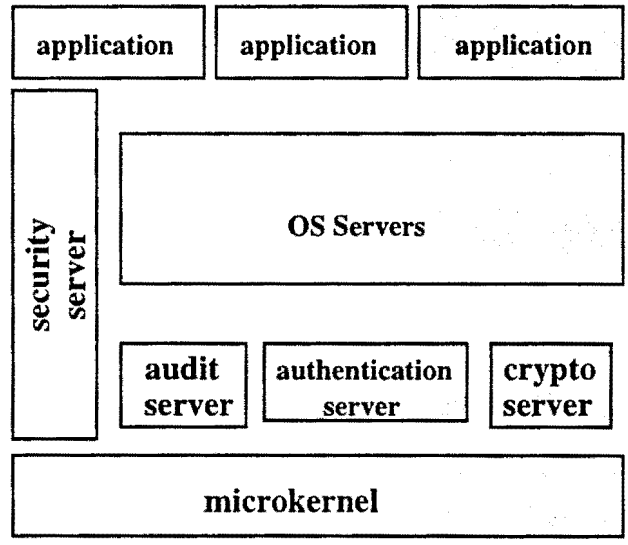


Figure 6.1: The Synergy Model

1. A microkernel provides the basic lowest level operating system functions (see the discussion of Mach to follow in 6.2.1). It also enforces all system access control decisions.

2. A Security Server is responsible for all access control policy decisions.

3. An Audit Server accepts and logs audit messages from all parts of the system.

4. A Cryptographic Subsystem is provided for encryption services.

5. A Network Server supports multi–level secure communication across physically unprotected networks.

6. An Authentication Server provides authentication for users and other systems.

7. OS Servers are responsible for all normal OS functionality not included in the microkernel.

5.2 DTOS

DTOS is an implementation of aspects of the Synergy model. Its stated goal is "..directed toward making an experimental microkernel–based secure operating system generally available to support further research and development in a number of different aspects of secure distributed computing[15]."

DTOS uses a modified version of the Mach microkernel with a `LITES` server to provide a Unix personality. We will first consider the Mach system. Then we will consider the security features added by DTOS.

5.2.1 The Mach Operating System

The Mach operating system was designed with the following objectives.

1. Provide a base for other operating systems.

2. Support large sparse address spaces.

3. Allow transparent access to network resources.

4. Exploit parallelism in both system and applications

5. Portability.

Mach is designed around a microkernel. The microkernel is much smaller than a monolithic kernel such as Unix. Indeed, its scope is much more limited. The responsibilities of the microkernel include only simple process management, memory management, communication, and I/O. All remaining duties, including file system management, are handled by servers running in user space.

In accordance with the first stated goal above, Mach may be used as a base for other operating systems by providing an *emulator*. An emulator typically consists of a set of servers to handle various OS functions and an emulation library. For example, consider the BSD Unix emulator. When a Unix system call is made, the application traps to the microkernel. The microkernel "bounces" this back to the emulation library, which is part of the application binary. The emulation library redirects the system call as a remote procedure call to the appropriate Unix server, which uses Mach primitives to satisfy the system call. In this way Mach takes on the "personality" of BSD Unix.

Some current releases of Mach use a 4.4 BSD Lite server called `LITES`. Other emulators could be created with the personalties of other operating systems. Indeed, one could run multiple OSs simultaneously on top of Mach by providing the appropriate servers and emulation libraries.

The Mach microkernel supports five basic abstractions: tasks, threads, ports, messages, and memory objects.

Tasks: A task is the environment in which program execution can occur. It is a passive entity consisting of the address space for the program (program text, data, and stacks), one or more threads, and some ports (including the process port, the bootstrap port, the exception port, and registered ports).

Threads: A thread is the executable member of the task, or its "active entity". It contains a program counter and registers, executes the instructions, and manipulate the task's address space and registers. Threads are managed by the Mach kernel. A thread is associated with a particular task, but more than one thread may belong to that task. In a multiprocessor environment threads may execute on different processors in parallel, while in a single CPU environment they are timeshared. Threads common to a task share the same address space and the special ports owned by the task of which they belong. In essence, a Unix process is a task containing a single thread. The "lightweight" nature of a thread (i.e., the fact that creating a new thread does not involve a new address space and special ports) means that it is relatively cheap to create multithreaded programs in comparison to Unix processes.

Ports: A port is a protected "mailbox" inside the kernel. All interprocess communication in Mach is done via unidirectional ports. There are three basic port rights:

1. A *receive* right — Each port has one and only one receive right associated with it. The task with this right is the only task which can get messages from the port.

2. A *send* right — Each port may have multiple send rights associated with it. Each task holding a send right may send messages to the port.

3. A *send–once* right — Each task holding a send–once right may send one message to the port, after which the right disappears.

There are also rights, more properly associated with the messages than the ports, that relate to the transfer between tasks of these basic rights.

Associated with each port is a message queue, a count of rights to that port, and limits on the amount of virtual copy memory, port rights and data the port can receive.

All Mach entities are referred to by ports. For example, a task always has an associated *task port*. Threads executing on behalf of a task use the task port to send requests to the kernel. A task also has a *bootstrap port* (receive right) for task initialization, an *exception port* (receive right) on which it receives errors, and a set of registered ports for communication with standard system servers. Likewise, threads are referred to by a thread port and abstract memory objects by a memory object representative port.

Messages Mach interprocess communication (IPC) is based on message passing. In fact, there is only one message passing system call, albeit a very complicated one. Messages are sent by a thread to a port. To send a message to a specific port, the thread or its task must possess a send or send–once right to that port. Messages are sequenced (queued on the port) and reliable (guaranteed to be delivered).

There are different types of messages. Messages may be used to send small amounts of data, large amounts of data, or port rights. Messages consist of a message header containing, among other things, the type and size of message data, the port to which it is sent, and the data itself. If the amount of data is very large, an "out–of–line" message may be sent. In this scenario the data consists of a pointer to a region of memory. The system copies this area of memory into the receiving task's address space. For efficiency, the memory object backing this region (see discussion below on memory objects) is mapped into the receiving task's address space. Only when a write occurs on a page in this region will an actual copy be made. This policy, called "copy–on–write", is used throughout Mach and is responsible for much of the efficiency of its virtual memory system. If a task has the appropriate capabilities, a message may be used to transfer or copy (in the case of a send right) a right to another task.

A message based IPC system is well suited for a client–server paradigm. The Mach OS uses it internally, and applications are easily written this way. The **netmsg-server** is largely responsible for the ability to access network resources transparently, providing a network–wide ASCII name registration of ports and a network transparent delivery of messages.

Memory Objects The virtual memory system is divided into three parts. This division improves Mach's portability. The parts are:

1. The **pmap** that manages the hardware memory management unit.

2. The machine–independent kernel code that processes page faults, manages address maps, and replaces pages.

3. The **memory manager** or **external pager** that manages the backing store (disk) through the abstraction of memory objects.

A memory manager manages memory objects. For each type of object there may be a different memory manager. A memory object is referred to by a port that associates it with a particular manager. It may consist of one or more pages, a file, a device, or another specialized data structure. Each is treated the same by the outside world. The difference lies in the memory manager used. For example, a device may have a different memory manager than a file.

Consider an external pager (a memory manager for a disk). Any region of physical memory is backed by an abstract memory object. The memory manager defines how the memory is organized, and how it can be used. One could say that a memory manager gives an area of memory its semantics. When a task references an area in its address space not presently in memory, a page fault occurs. The kernel sends a message to the memory object's port requesting the correct page. This message is received by the

memory manager for this object and handled according to the data in the memory object structure.

Programmers may write their own memory managers. There is a well–defined, asynchronous protocol with the kernel to which they must adhere. In this way, a user may create a customized paging policy for a specialized class of memory objects.

At a slightly higher level, a programmer is given much greater flexibility over the use of virtual address space than in a most operating systems. For example, memory objects can be mapped into a task's address space at pre–chosen locations. This provides support for large sparse address spaces. Sharing of memory between tasks can be achieved by mapping a memory object into each task; perhaps one with read and one with write permissions.

5.2.2 DTOS and Security

DTOS has altered the Mach microkernel with the following objectives [15]:

1. Provide policy–based control over all Mach services.
2. Minimize the impact of security on performance.
3. Maintain compatibility with the existing Mach interface.
4. Support a wide range of policies, including dynamic policies.
5. Provide a platform on which to experiment with secure applications.

To achieve these objectives the Mach microkernel and Lites server have been modified. DTOS is designed to provide a policy–independent operating system with all security policy decisions made by a Security Server. Mechanisms to support a wide range of security policies are available.

Two types of modifications were necessary to the microkernel. It was altered to perform security checks on all accesses. Also, for performance reasons, a cache was added to store recent security policy decision information.

The Lites server was modified to label all files with a security context, enforce policy control over file operations, and allow security aware applications to specify a label on a process.

In DTOS, security subjects consist of threads executing in a task. Security objects may be either ports or memory regions. With each type of object, a set of *services* and corresponding *permissions* is associated that indicate the access modes and permissions allowed for that object.

The Security Server "embodies a specific system security policy and provides security relevant information[15]." The prototype security server released with DTOS implements a policy of Type Enforcement, though other policy styles are possible. Type Enforcement is based on the enforcement of permissions to which a subject is entitled with respect to an object. It establishes the rules that decide if an action that a subject requests to perform on an object should be permitted. Type Enforcement restricts access to data using *domains* and *types*. A domain defines a role which users or programs may assume. For instance, in the DTOS prototype, all tasks responsible for initiating the system run in the `bootstrap` domain. Types are classifications of objects. An object may be of type regular file, Mach task port, host port, or many others. A permission such as create_file might be granted to a subject in domain A to create an object of type regular_file.

Security is implemented in the DTOS prototype by adding permission checks to all services provided by the microkernel. This is done by consulting the Security Server, which provides the information in accordance with the security policy. The microkernel checks this information and enforces the decision. The Lites Server behaves

similarly with regard to the file system permissions and the Network Server with regard to network security permissions.

The particular security policy is defined in a *security database*. For the prototype Security Server, the database defines permissions for all subject–object pairs. Included are such permissions as those for reading or writing a file, possessing various Mach port rights, creating a new task, and accessing various devices. All subjects are labeled with a *security context*. That context consists of a domain, a level, one or more categories ("none" is acceptable), and, perhaps, a classifier.

Levels and *categories* are orthogonal within domains. The terminology is taken from standard multilevel secure systems, where a level may be unclassified, confidential, secret, top–secret, etc. Domain–level compartments are separated into categories. These categories may or may not be accessible to each other. An object *classifier* creates a type from a subject domain. For instance, if a task running at user:unclassified:none creates a regular file, the context of the file will be user:unclassified:none:reg_file, where reg_file is the classifier. The type of this file is then user_reg_file.

These are the various forms that security contexts take in the database.

Subjects

Subject: Domain:Level:Category1,Category2,...,CategoryN

e.g. Unix:unclassified:none

Object: There are different types of objects.

1. Objects Related to Subjects (Derived Objects)

   Domain:Level:Category1,Category2,...,CategoryN:Classifier

   e.g. user:secret:nato:reg_file

This refers to an object which is a regular file in the user domain.

Nato is a standard MLS (multi–level secure) category.

2. Root Objects (not related to a domain)

   Type:Level:Category1,Category2,...,CategoryN

   e.g. dev_iopl_port_sid:unclassified:none

3. Objects Related to Root Objects

   Type:Level:Category1,Category2,...,CategoryN:Classifier

   e.g. dips:unclassified:none:device_pager_port

The type "dips" is the short name for dev_iopl_port_sid.

The database consists of subject–object pairs. For instance, here is an example take from the database that defines the permissions for a subject in the `security` domain and an object of type regular file in the `user` domain (`user_reg_file`).

```
/* This is a subject domain */
domain:security
/*  This is the object type. The type here refers to a
regular file.  The permissions are all file service per-
missions which will be enforced by the Lites Server. */
type:user_reg_file
/* The following are lists of the specific permissions for
the subject to act upon the object.  For example, a
fsv_create permission, in this context, allows a task in
the security domain to create a regular file in the user
domain. */
/* First come the permissions when subject and object are
at the same security Level (and in the same category) */
```

```
perms:fsv_create,fsv_link,fsv_unlink,fsv_append
perms:fsv_truncate,fsv_visible,fsv_exec,fsv_write
perms:fsv_read_fsv_chflags,fsv_chmod,fsv_chown
perms:fsv_utimes,fsv_stat,fsv_access,fsv_revoke
perms:fsv_chcontextfrom,fsv_chcontextto
perms:fsv_rename,fsv_rmdir,fsv_setlock,fsv_getlock,fsv_un-
lock
/* Next come the permissions when the subject Level domi-
nates the object Level. */
dom_perms=fsv_visible,fsv_exec,fsv_read,fsv_stat,fsv_ac-
cess
/* These are the permissions when the subject Level is
dominated by the object Level. */
domby_perms=fsv_create,fsv_write,fsv_truncate,fsv_ap-
pend,fsv_chcontextto
/* There are, in this instance, no permissions for incom-
patible Levels, i.e. the Categories field is different. */
incomp_perms=
```

The Security Server loads a policy that consists of permission lists such as those above. When the microkernel, or, in this case, the Lites Server queries the Security Server with subject and object security contexts, it replies with an access vector that contains the permitted actions of the subject upon the object.

Since the policy may be changed to fit the goals of a specific system in a fairly straightforward manner, DTOS can reasonably claim to be policy–independent.

Since the policy decisions are localized in the Security Server, separation of policy and enforcement is effected. This, in turn, makes a change of policy reasonable to implement.

6. X and Security

Let us look at some specific security problems involved in creating a secure X system and proposed solutions to them [4]. The problems mentioned here are general, but the solutions assume a design featuring a single X server. Our multiple X server solution will be presented later. Throughout this discussion an assumption is made that on any "secure" X system, "well–behaved" applications can run unchanged. The following list represents, in part, an interpretation of the B3 requirements for an X system.

Authentication: Who can connect? Some control is needed over what clients are able to connect to the X server. Currently, there are two mechanisms: the host access list, providing access control per host, and the MIT "magic cookie" authentication, providing access control per client. The former does not provide very fine grained control, and the latter is not very secure. This is often improved upon by using Kerberos or some other authentication protocol to mediate access to the X server.

Privileges: X has no notion of privileges. There is no means for limiting the functionality of clients on a per client basis. Of course, a wide range of possible privileges could be implemented. Are privileges static or dynamic? How does the X server discover the privileges of a specific client? How are they handled in a networked environment? If the privileges are dynamic, how is the buffering of requests handled? These are only some of the questions which need to be considered. One could define separate privileges for each class of X operation and give every protocol event a privilege label. This is the course of action suggested by [4].

Mandatory Access Control: How is access to private resources limited? How is access to shared resources limited? Every resource needs to have an attached sensitivity label. This may still cause problems, because any access to even a local resource could result in events being sent to other clients. This opens information channels. Resources which

are normally shared must be polyinstantiated or have access to them limited by privileges — or both.

Discretionary Access Control This is only an issue if different users utilize the same server at the same time. For B3 assurance, user–based access control lists are necessary. Maybe even per–client based access control is desired.

Secure Networking This is really an outside issue, i.e. an underlying system must already be in place to guarantee that the network is secure. At least the sensitivity label of a connecting program must be provided when a connection is established; perhaps including specific X security information such as privileges, DAC, information levels, etc.

Visible Labeling The TCSEC can be interpreted as requiring visible labeling of windows (specifically of top–level windows). This can be accomplished using a modified window manager to display labels for each of these windows. The concern then arises of label spoofing: when a client pretends to be at another level by creating what looks like a security border around itself. Can this be avoided?

Trusted Path A trusted path mechanism is required for B3 certification. This mechanism should allow the user to determine the sensitivity of all windows and to change the input information label. It must provide non–spoofable access to the trusted system.

Auditing It must be determined which actions should be audited and how they should be identified and characterized. The natural security subject for the X server is a client, not a thread or process. Whenever a client makes a request for a resource that it does not own, therefore, it should be logged. Connection requests should also be logged. Some ambiguity remains.

Cut and Paste How do we regulate the flow of information between clients? A mandatory access control policy does much of the work. A covert channel exists, however, when

the cutting client is at a lower level than the pasting client and a request is received from the pasting client declaring in what format the cut information should be sent. One solution would be to require a trusted intermediary which converts the formats itself. Other solutions seem less feasible.

Denial of Service A trusted path mechanism can aid against some types of denial of service attacks. If some client is inhibiting normal operation of the system by dominating resources, the SAK allows the user instant access to the system, from which it can kill the client.

Input Processing How are grabs controlled? Recall that a *grab* occurs when a client registers for, and receives all input from a device. How do we keep a client from grabbing input intended for another security level? Input events could be labeled according to the MAC policy and only delivered appropriately. The trusted path could be used to change the MAC label as needed. Then, when a grab occurs, it only applies to input to which the client has access under the MAC policy.

Overlapping Windows Overlapping windows create very complicated problems. Information channels abound because clients redraw their own windows when they are exposed. The X server must let a client know when another client at a different sensitivity level covers and exposes the first one's window. This provides a means of communication between the two clients. Possible solutions are:

1. The server could keep *backing store* (backup buffer containing the the image of the window) for all windows, so that the client need never know when it has been exposed. This, unfortunately, is a big memory drain. On a typical screen nearly two megabytes of memory would be needed for backing store.

2. Slow down the covert channels by requiring exposed clients to redraw their entire windows. Aside from not solving the entire problem, this requirement would slow down everything else as well.

3. Use a tiling policy to make sure that windows do not overlap. This is not popular with users. In addition, if clients can determine how much tile space is available, that could be used for a covert channel.

Window Managers Must window managers be trusted? In the single X server case covered here, it seems difficult, if not impossible, to avoid. Who is going to be responsible for labelling? If the window manager must, then it needs to be trusted. The trusted portion can, perhaps, be separated from the untrusted portion so as to minimize trusted code, but this may be a very difficult task.

TCB Size and Structure In order for the X server to meet B3 requirements, the trusted portions of it must be minimized. The trusted and untrusted portions are then separated. The Trusted Computing Base of the system thus would contain parts of the X server and parts of the window manager.

7. The DX System

The DX design separates out the portions of the X server that must be trusted, namely, the input and the output operations. The keyboard, mouse, and screen are shared by all clients regardless of sensitivity level. The remaining untrusted portions of the X server are polyinstantiated when required by the security policy.

As in TX, the untrusted portions are in a separate task called the **Single Level Server (SLS)**. It is modified to accept input from an **Input Manager (IM)** instead of directly from the keyboard and pointer. Since there may be multiple SLSs running simultaneously, the IM is responsible for directing input to the correct instantiation.

The **Display Manager (DM)** accepts the output of SLSs, in the form of buffers containing output images, which it combines into a coherent screen image. The **Trusted Module (TM)** provides a trusted path mechanism and a user interface for DX administration tasks.

The DX design differs from the TX model in the handling of security decisions. The **Communications Manager (CM)** handles client connections, directing the X protocol messages to the correct SLS. Instead of providing a built–in policy, the DTOS Security Server is consulted for each of the decisions. The CM is also responsible for starting new SLSs.

Cutting and pasting are accomplished by the trusted **Property Escalator (PE)** with assistance from the untrusted **Selection Emulators (SEs)**. The PE mediates the transfer of cut data between different SLSs. The SEs are SLS applications that transfer cut data to and from the PE. Again, these act similarly to their TX counterparts, the major difference being how security decisions are made. The PE makes requests of the Security Server for policy information regarding permissions to paste data, and enforces those decisions.

Each of these modules, together with **Init**, the program that starts all the trusted modules, is a separate Mach task with a separate address space. They communicate with each other soley through IPC mechanisms. An attempt has been made to keep the size of the trusted modules as small as possible for greater assurance.

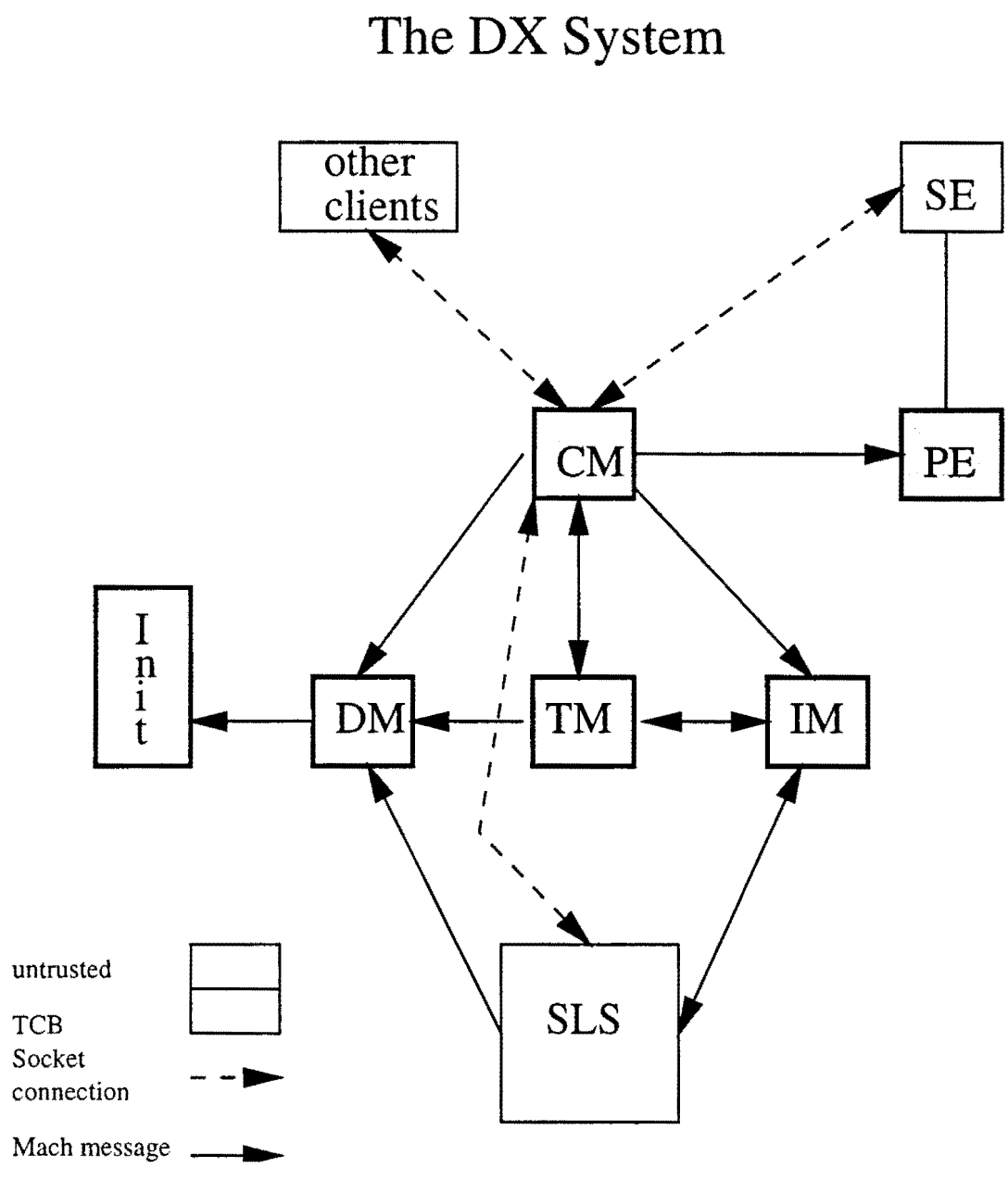


Figure 7.1: The DX System

7.1 Init

Init starts the CM, DM, IM, and TM. First the DM is invoked and the Init task then waits for the DM to send it configuration information such as the physical screen size and color depth, which will be needed by the SLSs. When the CM is created, this information is passed along so that the CM can forward it to the Single Level Servers when they are started. The TM also uses this information when forming its frame buffers to send to the DM. Certain ports are created and rights to these ports are inserted in the newly created tasks. These ports are used for communication between the various DX modules. The ports are named `cm2dm`, `tm2dm`, `tm2im`, `im2tm`, `cm2im`, `cm2tm`, `tm2cm`, `cm2pe`, and one unnamed port for sending the configuration information from the DM to Init (see figure 7.1).

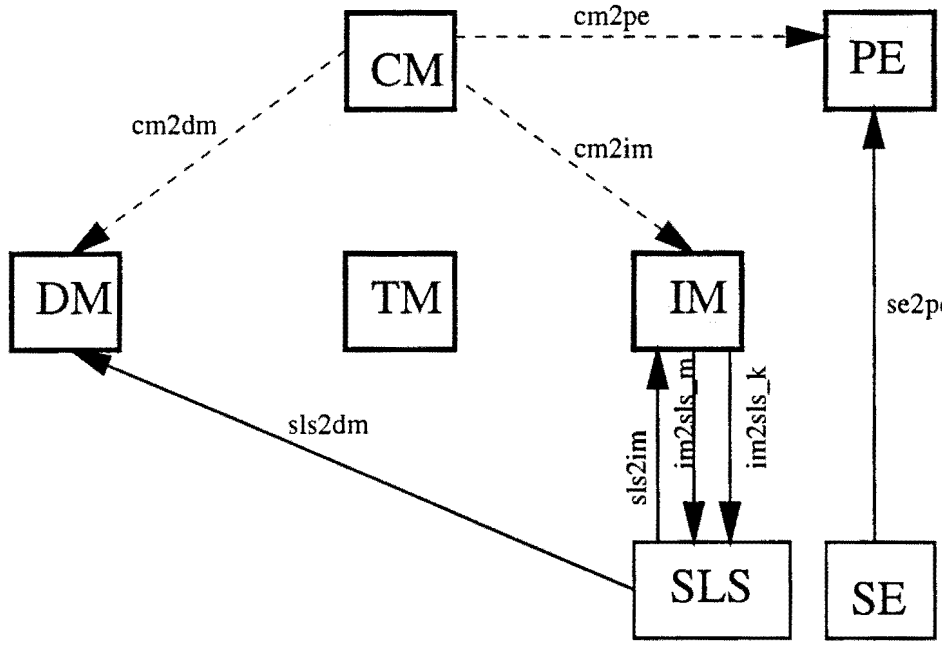
7.2 The Communications Manager

The Communications Manager has three primary responsibilities: 1) It connects X clients to the appropriate Single Level Server, 2) It consults the Security Server to see when a new Single Level Server should be started, and 3) It starts Single Level Servers and Selection Emulators.

The CM is started by the program Init at DX startup time and listens on port 6000, the default X server port, waiting for clients to connect. Upon such a connection request, the CM makes an `accept_secure` Lites call in order to obtain the security context of the client. A list of running SLSs with their connection ports and security contexts is kept in the CM. First, these security contexts are checked to see if any match that of the client. If so, this is the first chosen. If not, the choice is made to check the oldest server first. The Security Server is consulted by asking if the subject security context (for the client) has `dxv_share_server` permission to the object security context (the SLS as an object has the SLS subject context with the classifier, *dx*. See chapter 7.7). If the

permission bit is set, the CM opens a Unix socket connection with that SLS and begins forwarding all messages from the client to the SLS and vice versa. If the permission bit is not set, then each subsequent SLS is checked in turn in the same manner.

If the client is not allowed to connect to any running SLS, the CM starts a new one at the security context of the client. An SE also must be started at that context. Certain port rights must be granted at this time by the CM for communication between the new servers and other system components. These include `sls2dm`, `sls2im`, `im2sls_m`, `im2sls_k`, and `se2pe` (see figure 7.2).



A port xx2yy implies that xx has sends rights to the port and yy has the receive rights to it.

Figure 7.2: DX Ports and Port Rights

7.3 The Single Level Server

The **Single Level Server (SLS)** is an X Server modified for use with the DX system. To maintain as much compatibility as possible with future X server releases, our objective is to minimize any changes to the X server for use as an SLS.

We begin with the Xvfb server, which is part of the core X distribution for X11R6. This server emulates a dumb frame buffer using virtual memory. All rendering is done to this frame buffer instead of writing to the screen. It has no dependencies on either input or output hardware since it does not use them. It accomplishes this by replacing the standard X server output code with stubs or simple substitutions that use the virtual frame buffer instead. The standard input code is replaced by stubs that do nothing, since Xvfb does not take input from devices.

For our purposes a number of changes needed to be made in order to use Xvfb. These included re–introduction of the standard X input code, which was then modified to accept its input from the Input Manager rather than directly from the devices.

7.3.1 Modifications of X Server for use with Input Manager

The changes necessary to allow the X server to work with the Input Manager are at a low level and fairly limited. This serves the dual purpose of simplicity in the reworking of the X server and minimization of code in the trusted Input Manager. The Input Manager simply forwards input to the X server which acts upon it in much the usual manner.

At the lowest level of the X server reside the operating system specific functions for opening, closing, reading from, and writing to devices. Functions requiring alteration for use with the Input Manager include those routines associated with the input devices. For instance, the code to open a device was replaced by a Mach IPC request to the Input Manager. The Input Manager opens the device and replies with a (dummy) file

descriptor — the return type of the replaced function. The case is similar for `close` and `ioctl` calls. On the other hand, device read and write calls are replaced by routines that check Mach ports for the arrival of input events. These functions are not request–driven (which would be most natural) for the sake of functionality and efficiency. If a request were made of the Input Manager each time the X server checks for input, either a) the X server would block waiting for a return message, retarding its operation severely if it is non–active, or b) the Input Manager would need to be able to simultaneously manage requests from all the Single Level Servers, adding to its complexity and degrading performance. All of these routines, **xf86KbdOn**, **xf86KbdOff**, **xf86KbdEvents**, **xf86MouseOn**, **xf86MouseOff**, **xf86MouseEvents**, are in the operating system specific `Mach_io.c` file. In addition, a few routines exist that write to the mouse and get the keyboard type. These reside in a common directory and were changed in much the same way.

7.3.2 Modifications of X Server for use with Display Manager

Changes made to the Xvfb server in order to work with the Display Manager include the addition of message passing functions to enable communication between the Xvfb server and the Display Manager. Specifically, the virtual frame buffer itself was sent in a Mach "out–of–line" message.

Most messages to the DM originate in the Xvfb DIX routine `Dispatch`, in dispatch.c. These are sent from the primary dispatch loop under certain conditions:

1. The request from the client is one of: MapWindow, MapSubwindows, UnmapWindow, ConfigureWindow, CirculateWindow, WarpPointer, SetInputFocus, SetDashes, SetClipRectangles, ClearToBackground, CopyArea, CopyPlane, PolyPoint, PolyLine, PolySegment, PolyRectangle, PolyArc, FillPoly, PolyFillRectangle, PolyFillArc, PutImage, PolyText, ImageText8, ImageText16, Crea-

teCursor, CreateGlyphCursor, RecolorCursor, SetScreenSaver, KillClient

2. The request refers to a window, specifically an Input–Output window (as opposed to an Input–Only window, which is not visible).

3. The window in question is mapped (displayed on the screen).

4. The "active" top level window has changed since the last request. That is, the ancestor of the current window that is also a child of the root window is different than that of the last request. A message is also sent if a fixed time has elapsed since the last request in the same top level window, during which time at least one of the above requests was made.

In effect, a message is sent once per fixed time interval (small enough to be reasonable) if requests relating to a top level window have been made during that time. In addition, when one of the requests listed in point 1 occurs for a window in a different top level window than the previous one for which a message was sent, the old window is updated and the new one is sent. The use of this timing mechanism was implemented for reasons of efficiency. The large number of messages required for many applications were deemed unnecessary and a serious drain on computing resources. The timing policy served to eliminate a large portion of these messages, significantly improving performance.

The contents of a message to the Display Manager are as follows.

1. The virtual frame buffer passed "out–of–line". Since the frame buffer can be quite large, for example, this prevents a resource–consuming extra copy. Mach simply passes the address of the virtual frame buffer, allocates memory for it in the virtual address space

of the receiving task, and maps that memory to the same pages containing the data. Mach's copy–on–write policy ensures that the frame buffer is actually shared by the two tasks until it is written again.

2. The window id of the top level window.

3. The "extent" of the descendent of the top level window in which the drawing actually occurs. That is, we send the coordinates of the upper left and lower right corners of the smallest rectangle containing the window referred to by the request. This window is contained in the top level window referred to in 2.

4. The stacking order of all top level windows that are currently mapped.

5. The operation performed on the window. The following operations are defined: MAPPING, UNMAPPING, CURSOR, CONFIGURE, and UPDATE. These are the only ones the Display Manager chooses to differentiate.

The last four items in this list are sent "in–line" in a single structure.

Messages related to the cursor are handled a bit differently. They are sent from the routine `miSpriteMoveCursor` in misprite.c every time the routine is invoked. Each time the cursor is moved, the new location is sent to the Display Manager. The contents of the message are the same as above, except that there is a dummy window id. The "extent" of the cursor is identical in concept to the extent of windows.

7.4 The Display Manager

7.4.1. Initialization and Connections

The Display Manager is started by the program, Init. When it is invoked it is given certain port rights: dm2init — a send right to Init (i.e. to a port to which Init has receive rights), cm2dm — a receive right from the Communications Manager (i.e. a

port for which the CM has send rights), and tm2dm — a receive right from the Trusted Module.

The Display Manager begins by examining a modified version of the XF86Config file, which excludes the usual keyboard and pointer sections. It then probes and initializes the output devices. This part of the code is taken directly from the X Sample Server. The DM then executes the following initialization procedures:

1. A message is sent to Init (using the dm2init port) containing the screen dimensions. Init passes these on to the modules that need them: the CM, TM, and SLSs. The CM does not use this information directly, but distributes to the TM and SLSs. They need it to compose their respective frame buffers to be sent to the DM.

2. A port set is allocated to be used for receiving messages from SLSs and the TM, and moves tm2dm into it. The port set is a convenience used to listen at all DM ports simultaneously, rather than each individually.

3. A new thread is created (NewConnectionThread) to accept messages from the CM to transfer receive rights to ports for which SLSs have send rights. These are the ports to which the SLSs send requests and frame buffers.

4. A frame buffer is allocated to be used to store the "background" of the screen.

The NewConnectionThread begins in DMAcceptConnections, which waits for messages from the Communications Manager. Such a message contains receive rights to a new port and its receipt indicates that a new SLS has been started with send rights to this port. This port is renamed for the convenience of the DM and then added

to the port set created in step two above. Finally, the port is added to an array of ports for each running SLS.

7.4.2 Normal Operation

After the initialization steps above are completed and the NewConnectionThread has received a message from the Communications Manager indicating that a Single Level Server is running, the Display Manager is in normal operational mode. The routine, *main*, loops, reading requests from SLSs. A request from a Single Level Server consists of:

1. a virtual frame buffer
2. the colormap of the virtual frame buffer
3. the stacking order of all top–level windows (children of the root window) controlled by the SLS
4. the extent (dimensions) of the window upon which the operation is to be performed
5. a unique identification number for the top–level window that is an ancestor of the window upon which the operation is to be performed
6. the operation to be performed on the window: MAPPING, UNMAPPING, CONFIGURE, CURSOR, ROOT, UPDATE.

The first three operations are performed only on top–level windows.

When a request is received from an inactive server three situations may occur. If the operation is MAPPING, UNMAPPING, CONFIGURE (movement of a window or a change in its size), or ROOT (any operation pertaining to the root window),

the requests (minus the frame buffer) are queued for later delivery when the server again is active. If the operation is CURSOR (i.e. a cursor movement) it is ignored. If the operation is UPDATE, it is processed normally.

If the request is from the active SLS and the operation is MAPPING or ROOT, the colormap information is stored using the appropriate X StoreColors routine. Otherwise, the routine `vfbUpdateScreen` is invoked.

`VfbUpdateScreen` is the heart of the Display Manager. If the operation is MAPPING, the new window information is added to the global array **winList**, which keeps track of all windows mapped for each SLS (note that the DM only keeps track of top–level windows). Then the security border for the window at that level is printed to the screen. This is a simple bitmap that is replicated to fill rectangular areas on the screen — one for each side of the window. Finally, the window is drawn to the screen. More precisely, the box, whose extents are sent with the request, is copied from the virtual frame buffer to the screen.

Copying a box from the virtual frame buffer to the screen becomes much more complicated when the window referred to is not on top of the stacking order. In this case, and others we will see, we would like to draw to the screen only those portions of the window that are currently visible. In order to do so we find the "clip area" of the window. The clip area is a series of rectangles that, taken together, form the visible portions of the window. These rectangles have the following constraints:

a) rectangles do not overlap,

b) rectangles are sorted by the vertical coordinate of the upper–left corner and then by the horizontal coordinate of the same corner,

c) rectangles are further sorted into bands — rectangles in the same band share the same vertical coordinates, only their horizontal coordinates differ.

The routine `FindClipArea` iterates through the list of windows higher in the stacking order than the window to be written, finding the clip area with respect to each one (including the security border). The resulting clip area's coordinates are stored in a linked list, boxList. The members of this list refer to the coordinates of boxes in both the real frame buffer and the virtual frame buffer of the window in question. Then the rectangles described by this list are in turn copied from the virtual frame buffer to the screen.

If the operation requested by the Single Level Server is UPDATE, the request may be from either the active SLS or an inactive SLS. In each case, the clip area must be calculated. If the window to be updated is from the active SLS, it is clipped to each window above it in the stacking order for the active SLS. On the other hand, if the window is from an inactive SLS, it is clipped to each window above it in the stacking order of its SLS, and, also, to each window in each virtual frame buffer higher in the SLS stacking order. (The stacking order of these different "levels" is determined by how recently they have been active. The most recently active Single Level Server will be second in the stacking order to the currently active SLS, etc.) The clip regions, in both cases are copied from the appropriate virtual frame buffer to the screen. When the window is from an inactive SLS, however, the regions are also copied to the background virtual frame buffer[3].

3. Clip areas of windows are not, in general, calculated each time a window is updated. Instead, for efficiency, these regions are stored in the **winList** structure along with a clip_valid field. A clip–area is marked invalid when a window intersecting the one in question is either MAPPED, UNMAPPED, or there is a change in the active Single Level Server. A clip–area, then, is calculated almost only when it needs to be.

The root window of the active server is handled slightly differently. On a ROOT request, `vfbUpdateScreen` checks to see if the window is already mapped to the screen (i.e. if it is in **winList**). If not, the window is added to **winList**. The root window is then written to the screen after being clipped to all (non–root) windows at ALL levels. This will put it in the background, behind every other window as the user would expect. In order for this to work correctly with the cursor (as we shall shortly see), the window must also be written to the background virtual frame buffer, again clipped to all (non–root) windows at all non–active levels.

When the pointer device is moved, the active Single Level Server will send a CURSOR request to the Display Manager indicating the extents of the box containing the cursor image. The Display Manager then must remove the old cursor image and write the new one.

The first task is a rather complex one. The cursor must be in some subset of the set of three areas: 1) in a window of the currently active SLS, 2) in a security border of one of these windows, or 3) in a window of an inactive server. When the old cursor image is replaced, with what do we replace it? The portion in an active window should be replaced with the image of that region from the active virtual frame buffer. The portion in active security borders should be replaced by the appropriate security border bitmap. Finally, the portion not in one of the aforementioned regions necessarily must be in a window of an inactive SLS, and should be replaced by that region of its virtual frame buffer.

In order to facilitate cursor image replacement and the unmapping of windows (which will be examined below), the Display Manager creates a background virtual frame buffer. This buffer contains the image of all windows in all inactive servers, with their security borders, in the correct stacking order, as well as the root window of the current active server in the background. This background buffer acts analogously to the root window of a normal X server.

Now let us consider in more detail how the cursor image replacement is accomplished. When a CURSOR request is received by `vfbUpdateScreen`, this routine stores the location of the new cursor image for later use. When the next CURSOR request arrives, the location previously saved is the one that must now be replaced. This entire region is copied from the active virtual frame buffer to the screen. This will ensure that region 1 above is on the screen, but also, perhaps, some extraneous areas. Next, a check is made to see of the old cursor region intersects any security borders of a window from the active Single Level Server. If so, the borders affected are redrawn. (Recall that the security border of each window is divided into four sections: top, bottom, left, and right. Only the affected ones are redrawn.) This takes care of region 2. For region 3 we clip the area to be replaced to all (non–root) windows in the active virtual frame buffer, then copy these clip regions from the background virtual frame buffer to the screen. In this way all the non–active virtual frame buffers are handled at once.

Finally, when the old cursor image has been replaced, the new one is copied from the active virtual frame buffer to the screen. Note that this causes an interesting side effect: the area in the cursor image that is not the cursor itself (the cursor is often an X or an arrow, but by the *cursor image* we mean the rectangle that contains the cursor) is always from the active SLS. This means that when the cursor is in an inactive window or in a security border, this area around the cursor will appear as a hole or "window" into the active virtual frame buffer. This gives an extra means of determining which windows are from the active SLS, aside from the security borders.

When a request is for the UNMAPPING of a window, the sequence of events is similar to that of replacing a cursor image. The area that formerly contained the window is copied from the active virtual frame buffer. Next, the region, clipped to the current level's windows, is copied from the background frame buffer to the screen. Any security borders that have been affected in this process are subsequently redrawn. In addi-

tion to these steps, the window entry is deleted from **winList** and the clip–areas of any windows in the list that intersected the deleted window are marked "invalid".

Lastly, a CONFIGURE request (indicating a window movement or resizing) is satisfied by executing an UNMAPPING request on the window, followed by a MAPPING request with the new coordinates.

7.4.3 Changing Levels

In addition to the requests from the Single Level Servers, the Display Manager listens for requests from the Trusted Module. These are of two types: DISPLAYWINDOW, and TM_ACTIVE.

In the first instance, the TM message consists of a buffer and a height. This buffer is copied to the screen, starting at the top and occupying the first "height" lines. Most commonly "height" corresponds to a reserved area of the screen to which only the TM has access. Otherwise, if the buffer covers an area greater than that reserved, the area of the real frame buffer that is to be overwritten is saved into a new buffer, to be copied back at the appropriate time.

Upon receipt of a TM_ACTIVE request (when the Secure Attention Key is pressed) the DM enters a mode in which it accepts requests only from the TM[4]. These requests are DISPLAYWINDOW again or a request to change the active SLS. The latter request indicates the SLS number to be activated and signifies that the DM is no longer in TM_ACTIVE mode.

The routine `ChangeLevels` is invoked upon return from TM_ACTIVE mode if the server requested to be active is different than that which was most recently active. The duties of this procedure are as follows:

1. Change the stacking order of the connected Single Level Servers. The new active server is given the highest stacking order pre-

4. This requires removing the port right tm2dm from the port set to listen on it alone.

cedence, and each SLS between the top of the stacking order and the previous position of new one is moved down one notch, For example, if **server_stack_order**, the variable which keeps track of this, was {0,1,2,3,4,empty,...} and server 3 is to become the active SLS, **server_stack_order** would be changed to {3,0,1,2,4,empty,...}.

2. Set the global **current_server** to **new_active_server.**

3. Copy the (non–root) windows from each non–active virtual frame buffer to the background virtual frame buffer, adding security borders, in order from lowest to highest in server stacking order.

4. Copy the root window of the new active virtual frame buffer to the background virtual frame buffer, clipping to all (non–root) windows in each inactive frame buffer.

5. Copy the entire background virtual frame buffer to the screen.

6. Copy each (non–root) window of the new active virtual frame buffer to the screen in order from lowest to highest in window stacking order.

7. Change the active colormap to that of the last request from the new active Single Level Server.

8. Satisfy the requests that had been stored while the current server was inactive.

The result of these operations is a screen with a "background" consisting of the active root window with all the (non–root) windows of inactive servers on top. The "foreground" is made up of all the (non–root) windows from the active SLS. All windows

are labelled with security borders to indicate from which Single Level Server they derive. The Display Manager is then returned to normal operational mode and is ready to process all requests from the new active SLS.

7.5 The Input Manager

The Input Manager is responsible for opening, initializing, and reading the keyboard and pointer devices. It interacts with Single Level Servers, the Trusted Module, and the Communications Manager. The input devices are polled and, when input is present, the input events are sent to the active SLS or the TM. The TM receives all input after the Secure Attention Key is pressed until the TM determines the new active SLS. The CM informs the IM when a new SLS has been created.

By allowing input events to one Single Level Server at a time, as determined by the Trusted Module, the Input Manager effectively separates all input per SLS without making any security decisions.

7.5.1 Initialization and Communication Threads

The Input Manager is started by the program, Init. When it is invoked it is given certain port rights: a receive right on the port cm2im — for which the Communications manager has a send right, a receive right on the port tm2im — for which the Trusted Module has a send right, and a send right on the port im2tm — for which the Trusted Module has a receive right.

The Input Manager begins operation by creating a thread dedicated to listening for the Trusted Module. This TM thread awaits messages on tm2im, which indicate to whom the input events should be sent: either the TM itself or a Single Level Server. This signifies a change in the active SLS.

Next, the IM creates another thread which listens on the port cm2im for messages informing it of the creation of a new SLS. Port rights are contained therein:

a receive right for one, and send rights for two others. The new SLS has the corresponding send and receive rights for these ports. One port is reserved for special requests from the SLS. The two send rights correspond to SLS ports for the transmission of keyboard an pointer events. These port rights are placed in arrays indexed by the SLS number (determined by first come first assigned).

Lastly, before normal operation begins, `KbdOn`, `MouseOn` and `MouseIOCTL` are called. `KbdOn` open the keyboard device and sets it into non–blocking event mode.

7.5.2 Normal Operation

After the initialization steps are complete the Input Manager is ready to accept requests from, and send input events to, SLSs. When the Trusted Module indicates that a Single Level Server is now active, the IM will accept requests from the designated SLS. These requests are: KBD_ON, KBD_OFF, KBD_GETMAP, MSE_ONE, MSE_OFF, MSE_WRITE, and MSE_IOCTL A KBD_GETMAP request is usually first to arrive. The type of keyboard (always VANILLAKBD) is returned in a reply message. A KBD_ON request is responded to by returning the keyboard file descriptor (the keyboard was already "turned on" during initialization). Likewise, a MSE_ON request is followed by a reply containing the mouse file descriptor. The KBD_OFF, MSE_OFF, and MSE_IOCTL requests essentially do nothing but provide a dummy interface for the usual X server calls.

After the active Single Level Server has made a KBD_ON request, it begins to receive keyboard input when it is available. Similarly, after the active SLS has made a MSE_IOCTL request, it begins to receive pointer input. Since the SLSs require that input take the form of events, and the device drivers require that this be polled, the Input Manager polls the keyboard and mouse according to a simple timing mechanism that linearly increases the time between polling if no event has occurred. The interval

increases until a set limit is reached. When input is discovered, the interval is reset to a lower value (what value depends on the type of input received).

When an input event is discovered, the IM first checks to see if it was the result of hitting the Secure Attention Key. If so, all subsequent events are sent to the TM until further notice. If not, the event structure (containing a list of events in the case of pointer input) is sent to the appropriate SLS port. The SLS does its own polling on the input message queue. Due to the asynchronous nature of this interaction, combined with polling, it is conceivable that some input could be lost. To minimize this risk, the size of the input message queues is set to the maximum allowable by the Mach OS. This technique makes it unlikely that keyboard input will be lost. Pointer events may exist up to thirty two per event structure. making loss of these equally unlikely.

Requests from the active Single Level Server and polling for input are done in the main event loop. In order to keep the IM from blocking on a message receive, a timeout value is needed. This is integrated with the timing mechanism mentioned above so that the timeout values become the polling interval. Likewise, a timeout value is used on message–sends to keep a Single Level Server from effectively suspending the IM by not reading its messages.

7.5.3 Changing Levels

When the Secure Attention Key is hit all input (including the actual SAK keystroke) is immediately redirected to the TM. When that entity sees fit to return control of input to a SLS, it sends a message containing the index of the SLS to be made active. The TM thread reads this message and sets the variable **active_sls** to it. This is simply an index into the various arrays of ports associated withe SLSs. At this point the IM has changed the active SLS.

7.6 The Trusted Module

The Trusted Module (TM) provides a number of important services. Foremost among these is a trusted path mechanism. When the Secure Attention Key is pressed, the TM receives all keyboard and pointer input. The TM is then said to be "active".

The DM reserves an area on the screen for the use of the TM. Frame buffers, sent in messages to the DM, are displayed there. The information the TM displays includes:

1. A message indicating the security context of the active SLS.
2. The security border of the active server.

When the TM is active, an expanded menu is displayed, utilizing a larger portion of the screen. In this larger area the following menu is displayed:

TM Active

0. Startup new single level server
1. Shutdown single level server
2. Activate Shell
3. Exit DX
4. Sec. Context of running SLS
5. Sec. Context of running SLS

.

.

.

The top line indicates that the TM is active. Menu items 4,5, etc. designate the security contexts of running SLSs. Selecting one of these items will make that SLS active. The TM sends messages to the IM and the DM informing each of the new active SLS (these are the only other modules which have a notion of an active SLS).

Selecting menu item 0 will begin the process of starting up a new SLS and SE at a requested security context. It does this by informing the CM of the request. The

CM is responsible for both checking with the Security Server for the validity of such a request and for actually executing the programs. Selecting menu item 1 will initiate the shutting down of a SLS. This is done, again, by sending a message to the CM which kills the SLS and informs the the DM, IM, and PE that the SLS is dead. Menu item 2 activates a trusted shell at the security context of the shell which started up the DX system. This shell may be used to start clients or perform many other tasks within a trusted path. When the shell is activated, it is displayed by overwriting the menu above. The shell is ended by typing "exit". Selecting menu item 3 will initiate the process of shutting down the entire DX system (this is not yet implemented).

7.7 The Property Escalator and Selection Emulators

The ability to transfer information between windows is an essential part of the X Windowing system. Much like TX, the DX system uses a Property Escalator (PE) to mediate the transfer of information between clients of different Single Level Servers, and Selection Emulators (SEs) to aid in that transfer. There are some essential differences, however.

A **Selection Emulator (SE)** is a small untrusted X application. Its purpose is to pass selections (cuts) back and forth between the Property Escalator and a Single Level Server. It is replicated, once per SLS, of which it is a client.

An SE is started by the Communications Manager when it determines (by consulting the Security Server) that a new SLS should be started. The SE asserts ownership over the current selection so that it will be notified (through Xlib routines) when another client makes a new selection. When the SE receives this notification, it requests the data of the new selection and then re-asserts ownership of the selection. This selection data is converted to text format and forwarded in a Mach message to the PE.

As the owner of the "current selection" the SE receives notification when the selection is pasted. The target X application for the paste negotiates with the owner to copy the selection data. The SE, instead of returning the current selection data (for this SLS), sends a message to the PE requesting the latest overall selection. The PE returns either the most current *appropriate* data from another SLS, or a message indicating that the selection already owned by the SE is the most current one.

Upon receipt of the data from the PE, always in text format, the SE converts it to the format requested by the target client, and registers the data as the new selection with itself as owner.

This method of handling conversions is necessary to avoid covert channels. An SE (untrusted, remember) at one security context should not know the format of a selection at another. Converting all selections to text for storage in the PE circumvents this problem. Also note that it is somewhat different and more versatile than the way TX handles the situation. The TX PE stores each selection in *all* formats which are likely to be used.

The **Property Escalator (PE)** is responsible for storing selection data from the different Selection Emulators and mediating the transfer of that data.

The PE is started by Init at DX startup time. It receives selections from SEs and stores them in chronological order in a linked list, together with their security contexts. One selection is stored for each SE. When a request for a selection is received by the PE, a check is made to see if the requesting SE already has the most recent selection. If it does, the PE will inform it so in a message. On the other hand, if the most recent selection was from another SE, the PE will consult the Security Server to see if the requesting SE is allowed to paste. This is done by asking if the subject security context (for the requesting SE) has `dxv_paste` permission to the object security context (the selection gets the security context of the cutting SE with classifier `dx`, created for the DX sys-

tem. See chapter 7.7). If this permission bit is set for these pairs the PE will return the selection to the requestor. If not, the next most recent in the list is considered, etc. The process repeats until an acceptable selection is found or the possibilities are exhausted. In the latter case, the one the SE owns is deemed most recent.

7.8 DTOS Security Changes to Accommodate DX

A number of changes were necessary to allow the DX system to run under DTOS security mechanisms.

1. A new security domain, the `xserver` domain, was created in which to run the trusted portions of the DX system. Permissions relating to this domain were given where appropriate. It was patterned after the privileged Unix domain, which has access to all other domains currently available. System programs run in the Unix domain, so they need access to devices and files, etc., at many security contexts. Conversely, programs in other domains sometimes need to access system information. Much of this is true with the `xserver` domain as well. It does not need access to, say, the `bootstrap` domain, but it does need to communicate with all the domains in which users normally execute programs. Likewise, Init and the CM, running in the `xserver` domain, must be able to execute programs in other domains. The CM needs to be able to communicate via Unix sockets as well as TCP sockets. The IM and DM must be allowed to access certain devices such as the mouse, keyboard, and graphics hardware. All of these permissions were added to the security database for the `xserver` domain. In addition, a macro was created to facilitate the addition of permissions between the `xserver` domain and any new domain that might be created.

2. A new classifier, `dx`, was created so that specific permissions could be available for use with the DX system. Creating this classifier required two very small alterations to the DTOS kernel. Then a service permission access vector was created for the `dx` classifi-

er. The access vector returned by the Security Server when presented with a subject and an object consists of a set of IPC permissions and a union structure of mach_services permissions. The union is resolved by the classifier of the object. The new access vector is one member of this union.

```
struct dx_service {
        unsigned char;
        dxv_paste:1,
        /* can subject paste object? */
        dxv_shareserver:1,
        /* can subject run on this SLS? */
        dxv_pad:6
        /* padding */
}
```

Example permissions for all existing user domains were set up and a macro for creating multilevel secure style permissions for any new domain was created.

3. The context of `/dev/tty` was changed from user:unclassified:none to tmp:unclassified:none. When the application, *xterm*, runs it attempts to open `/dev/tty` with read/write permissions. Very few security contexts may both read and write a `user` domain device under the prototype MLS policy. These restrictions are not applied to the `tmp` domain.

8. DX Security Analysis

The security issues relating to the DX system have been treated throughout the foregoing descriptions. Here we summarize those features, analyzing with respect to the security issues described in chapter 6.

8.1 Authentication

The Communications Manager determines what clients are allowed to connect to what Single Level Servers. The Security Server is consulted to get the policy information for these decisions. The determining factor is the security context of the client. When a client connects with the CM, the context of each SLS is considered in turn, and the Security Server is asked if a client (identified by its context) is allowed to connect to the SLS (identified by its context with the classifier dx). When the permission is granted the CM opens a connection to the SLS. If no permission is granted for any SLS, the CM starts a new SLS and SE at the security level of the client.

In the future, DX will not automatically begin a new SLS, but, instead send a message to the TM requesting confirmation. The TM will indicate to the user at the console that a new SLS is requested. The user then will be responsible for either confirming that a SLS should be started, in which case, the TM will initiate the process, or ignoring this request.

8.2 Privileges

Each client that connects to the DX system is allowed the full range of privileges within its own sensitivity level. That is, clients have access to all of the SLS X resources that they would under a normal X server. The limitations are imposed on the resources to which the *SLS* has access. Access to keyboard and pointer input is restricted by the Input Manager. Access to the output screen is in the control of the Display Manager. The client has no privileges to resources in any other SLS. Access to any resource of a different SLS is prevented entirely by their complete separation. The only exception

to this rule is cutting and pasting. This is mediated by the trusted Property Escalator that enforces the permissions allowed by the Security Server as set forth in the security policy.

8.3 Mandatory Access Control

Every resource in the DX system has an attached sensitivity label inherited from the underlying DTOS operating system. When a client connects to the CM, the most common situation is that a connection will be opened to a SLS at the same security context as the client. Each SLS has a specific security context, so all its resources are labeled accordingly by DTOS. Every resource, including ports and sockets, has one and only one security context under DTOS. No resource is shared between different SLSs directly.

The DM controls the screen real estate. It is a trusted intermediary between the SLSs and the graphics hardware. It has an internal labeling scheme for frame buffers sent from SLSs. The DM can determine the security context of any pixel on the screen. None of this information is available to the SLSs.

8.4 Discretionary Access Control

Any user running in a context that is allowed to send information to a TCP socket in the `xserver` domain may currently connect to the system. The security policy limits which users may run under what contexts. The security policy of the operating system is thus responsible for limiting user access.

DX does not extend DAC to the X resource level. The primary reason for this is that the X server, in general, does not make distinctions based on the user. Such a policy would alter the whole flavor of X and require major modifications to the X server itself.

8.5 Secure Networking

All incoming data from the network is intended to go through the CM. There, it is directed to the appropriate SLS. The CM obtains the security context of a client through the DTOS `accept_secure` call. The context of the client, together with

the security policy determines what the target SLS is. Even if the CM were avoided, DTOS limits the connections to the SLSs. At the moment a client could bypass the CM and connect to a SLS. It could, however, only connect to a SLS with a context for which it has permission to write to a socket. This hole should be closed in the future. One way would be to establish the communication between the CM and SLS be via Mach messages. This would involve both wrapping the X protocol request in a Mach message and altering the SLS to accept and unwrap Mach messages instead of socket messages.

8.6 Visible Labeling

One of the key components in a secure windowing system is the labeling of windows displayed on the screen. The Display Manager attaches a security border to all four sides of every top level window displayed on the screen. This label consists of a simple bitmap replicated to surround the entire window. The TM displays, in its expanded menu, the bitmap for each SLS together with its context. It also displays the bitmap of the security border for the active SLS.

When a window is in the foreground (no other windows obscuring it) the entire security label is usually visible. If the window is situated flush against one or more of the sides of the display area (including top and bottom), however, that portion of the label will not be displayed. At least one side of the label will be visible under all circumstances except when the window in question takes up the entire display area (as the root window does). Two situations may occur. In the first, the active SLS has only the root window displayed and no other. In this case, another SLS could spoof the root window by covering the entire display area with its own window, because the root window of the active SLS is always in the background. This spoof is not considered to be a security risk. No information is transmitted to the spoofing SLS and the user has only to look at the active SLS field in the TM display to see the SLS to which input will be directed. In the second situation, the window in question is not the root window. In this case, the user

could determine the security context of the window simply by looking in the area reserved for the TM where the bitmap of the security border of the active SLS is displayed. Since all windows of the active SLS (with the exception of the root window) are "on top" of all other windows, any such window would necessarily be from the active SLS.

Spoofing windows at other sensitivity levels is limited by a number of factors in the design of the DM. Foremost is the labeling itself. Under almost all circumstances labeling makes it obvious from what SLS the window originated. Also, the windows from the active SLS are on top. This makes it impossible for a rogue SLS to overlap a window from the active SLS with the labels of its own windows, thereby convincing the user that a window of the active SLS is really from the imposter. A SLS has no idea where the windows of another SLS are located (or even if they exist). Therefore it would be extremely unlikely that one SLS could spoof a window of a different SLS by creating its own security border that looks like that of the level being spoofed. The DM would still put the real border on the outside of this one. To hide this, the rogue SLS could attempt to size and place the window so that active windows obscure the real security border. This would, however, require a peculiar configuration of active windows together with knowledge of the size and placement of those windows.

8.7 Trusted Path

The DX system offers a trusted path mechanism involving the IM, TM, and DM. A trusted path operation is initiated by a press of the Secure Attention Key (SAK). As we have seen, the IM, upon reading this, redirects all input to the TM. The TM sends all relevant information to the DM for display. All SLSs are in an inactive state with respect to the IM and DM. Any time a user desires trusted functionality, the SAK will allow him/her to interact with the TM.

8.8 Auditing

The prototype DX system currently does no auditing. At least, in the near future, connection attempts and cut and paste attempts will be logged. The latter is necessary because a client may perform cuts and pastes without intervention from the user.

8.9 Cut and Paste

The discussion of the PE and SE in chapter 7.6 covered cut and paste thoroughly. Please refer to it for details.

8.10 Denial of Service

It is probably not possible to eliminate all denial of service attacks. They have been taken into consideration in the design of DX, however. The trusted path mechanism is of great use for this purpose. The user can always take control and kill SLSs if they are, for instance, flooding the DM with window update requests. Another possible attack is flooding the CM with connection requests. Both of these approaches could slow down the system enough to make it virtually unusable. More serious denial of service attacks have been avoided by careful design of the IM. The IM uses timeouts on all message to or from the SLS, so that it will not block for long periods of time waiting for message transmission to complete. It sends input data to the active SLS when the data is read rather than waiting for a request for input from the SLS. In this way, the IM is not unduly impeded by slow SLS.

8.11 Input Processing

The IM guarantees secure input processing. All input data is directed to exactly one SLS or the TM. The data is then effectively labeled with the sensitivity label of that module. When a client grabs the pointer or mouse, therefore, the grab applies only to input data directed at its SLS. It has no effect on the distribution of other input data.

8.12 Overlapping Windows

The problems arising from allowing overlapping windows are avoided in the DX architecture. Only the DM knows if windows from separate SLSs overlap and there is no communication pathway from the DM to a SLS. A SLS simply creates its own frame buffer. It has no knowledge and no means of knowing that any other window could overlap one it has created. In order to keep the system this simple, the DM must keep a backing store for all the windows it displays. It does this by retaining the frame buffer from each SLS until a new one is received or that SLS terminates. When a window needs to be redrawn, the DM copies the data from the appropriate frame buffer to the screen.

8.13 Window Managers

Window managers are not trusted in the DX system. Instead, each SLS has its own window manager. There is no interaction between them. They know no more than the SLSs of which they are clients.

8.14 TCB Size and Structure

The size of the various trusted modules is given below. Some effort has been made to keep the size of the trusted computing base components within reasonable limits.

subject	approximate lines of C code (LOCC)	
	DX	TX
DM	11,500(SVGA), 19,000(S3)	9,700
CM	1,500	800 (SIT + CIT)
IM	500	2,400
TM	2,000	5,300 (TSH + MS)
PE	500	300
Init	500	550 (Master)

Table 8.1: TCB Size

When using the DM built for the SVGA graphics devices the total lines of C code for the TCB is about 16.5K. For S3 cards the total comes to about 24K. Including a library shared by all tasks (9.3K LOCC), the total for the TX TCB is approximately 30K LOCC.

9. Analysis and Future Work

9.1 Performance

Performance of the prototype DX server has not been studied in detail. Using some standard X benchmarks, it was determined that DX operation responses were on the average just less than half the speed of ordinary X running on the same hardware under DTOS. This is comparable to the performance of TX. For small rendering operations the performance was worse (about 35% on ten pixel lines), and better for larger areas (about 80% on large rectangles). The greater the complexity of the operation, the slower the performance relative to ordinary X. For a complex test, including creating a window, creating a graphics context, clearing an area, drawing some text, scrolling the window, and finally destroying the window, the performance was only about 20% of the normal X server.

No working profiler is yet available for DTOS, making the identification of problem areas less precise and more difficult. The results above would appear to indicate that a significant portion of time is spent in message passing and context switches, since more complex operations involve more messages. It was also observed that disk accesses were more common than had been hoped. This seems to be caused by excessive swapping resulting from high memory usage.

One possibility to improve the performance of DX is to utilize shared memory between the Single Level Servers and the Display Manager. The frame buffers passed from a SLS to the DM occupy roughly one megabyte of memory. Sending the buffers in out–of–line messages, together with Mach's copy–on–write scheme allows the servers to share this region of memory until one task writes to the buffer. The SLSs may write to this area almost continuously, causing copies to be made of the pages containing the frame buffer. If the memory could be truly shared, so that the SLS had read/write ac-

cess and the DM read–only access to it, these copies would not need to occur. Performing the same tests as above using significantly different frame buffer sizes, however, yielded statistically identical results, indicating that sharing memory would not result in an increase in efficiency due to fewer copy operations. Sharing would still be beneficial because the corresponding reduction in memory use caused by one fewer frame buffer should cut down on expensive disk accesses resulting from swapping.

Despite Mach's elegant virtual memory system, where memory sharing may occur by simply mapping a memory object into more than one address space, sharing at the user task level is quite difficult. The memory manager provided with DTOS does not allow a user task to gain access to memory objects directly, nor does it provide any other means of sharing memory. To provide shared memory between an SLS and the DM, a new memory manager will need to be developed. Even if memory sharing were implemented, there are potential problems with the consistency of the frame buffer and the actual screen displayed by the DM. The DM uses the frame buffer of a particular SLS as backing store for the windows it displays. When the DM updates a window by copying it from the frame buffer, the DM may catch the window in an incomplete state — e.g. only partially drawn. This image could remain displayed on the screen for a considerable period of time until the next update of that area. The DM could be altered to utilize a more traditional, per window backing store.

Another approach to SLS–to–DM communication would be for the SLS to send only those portions of the frame buffer that have been updated. Aside from some changes to the SLS, this would also require that the DM use a different method of backing store, probably a different buffer for each window, including the cursor. This approach would require an extra copy of the region to be sent by the SLS, but would eliminate some copying of other regions of the frame buffer.

To reduce the amount of memory used by the DM and so reduce disk accesses, frame buffers could also be compressed. Only the portions of the compressed frame buffer copied to the screen would be uncompressed. Whether the overhead of compressing and uncompressing buffers would eliminate the gain from fewer disk accesses is currently unknown.

9.2 Variations to the DX Design

In the future DX should be enhanced to log security relevant information for the purposes of auditing. This would, at least, include connection attempts and cut and paste attempts.

The possibility of the DM creating the cursor itself rather than copying the cursor image from the active frame buffer has been explored . A simplification of cursor management would result, but the changes in cursor style that many applications provide when the cursor enters a particular area would not appear on the screen.

One could have a different user interface such that switching between active servers only requires a mouse click in the new active window rather than a press of the SAK and choosing the new level through the TM. For this to occur through a trusted path, the IM would need to track the pointer so that, when the correct button is clicked in another window (from a non–active SLS), the TM would be notified immediately to initiate the change. The SLS, however, controls the visible cursor location, creating the possibility that the user could be fooled into activating a different SLS than intended. If the DM creates the cursor image itself and obtains the location from the IM this problem is avoided. Adding cursor tracking functionality to the IM involves the addition of significant code. The resulting increase in size of the trusted IM could be a cause for concern.

Though the DX system works well under a large variety of security policies, certain types of dynamic policies are not supported. Specifically, the revocation of a permission for a particular client to connect to its SLS, once that connection has been

established, is not allowed with the current design. Revoking the permission would require cloning the SLS at a different security context, killing the client in the old SLS, killing all inappropriate clients in the new SLS, and redirecting client communication to the new SLS. When a dynamic policy dictates that clients of different SLSs should be together, the situation is worse. Major modifications would be necessary to separate or reproduce the exact state of the resources belonging the client in one SLS for use in another.

9.3 Single Server Approaches to Secure X

There have been attempts to create secure X servers using a single server instead of polyinstantiated servers. **Compartmented Mode Workstations (CMWs)** usually have a monolithic trusted server, a trusted window manager, and a few trusted clients to assist with security borders and cut and paste operations. Implementations of the model exist and are in use today. They are low assurance (B1) multi–level secure systems.

Another approach would be to have a single server and a trusted window manager that would display different sensitivity levels in different viewing areas (i.e. like "desktops" in the FVWM window manager). This has the advantage that it makes some resources easier to separate, and, most importantly, it eliminates many possible communication channels arising from overlapping windows. The major drawback to this approach, aside from the complicated mandatory access control involved in any single server system, is its rigidity. Users may not like being able to see only one level at a time. Though there would be freedom of movement within the area reserved for one sensitivity level (unlike a system with "tiled" windows), the ability to juxtapose windows at different levels would be severely restricted.

Single X server systems would likely have an advantage in execution speed when multiple levels of sensitivity are present. This, along with the ability to control access at a finer level makes the model worth further investigation, despite the many problems involved.

10. Conclusion

A prototype DX system is now operational in a DTOS environment. It supports the full range of security policies that can be implemented using the DTOS prototype Security Server, and is expected to support most other types of policies. Clients do not require alteration for use with DX and behave in the expected manner.

The prototype DX system demonstrates that a secure X system that is policy–neutral is possible. An operating system that separates security policy from mechanism allows such an application to be developed.

The separation of security policy from mechanism that the Synergy architecture prescribes serves a large security sensitive application such as DX very well. Many modifications to the policy have been successfully attempted with no change to the DX system. A system that provides this separation mechanism, such as DTOS, allows the system and its applications to function in environments with varying security requirements. Some questions remain about other types of applications. The performance cost of making DX policy–independent is minimal, as messages to the Security Server are relatively few. Applications that consult the Security Server more frequently will incur greater performance degradation. These costs may be partially offset, in DTOS, by a mechanism that allows individual applications to cache security decisions. The application is notified when the cache should be flushed. If a dynamic policy requires very frequent flushes of an application cache, the benefits of the cache may be diminished. We suggest, however, that application caches for security decisions is an important mechanism for any operating system of this type.

References

[1] Robert W. Scheifler, James Gettys. *X Window System, The Complete Reference to Xlib, X Protocol, ICCCM, XLFD, 2nd Edition, X Version 11, Release 4.* Digital Press, X and Motif Series, 1990.

[2] Elias Israel, Erik Fortune. *The X Window System Server, X Version 11, Release 5.* Digital Press, 1991.

[3] Adrian Nye. *X Protocol Reference Manual for X11 Version 4, Release 6.* O'Reilly & Associates, Inc., 1995.

[4] Jeremy Epstein, Jeffrey Picciotto, "Trusting X: Issues in Building Trusted X Window Systems or What's Not Trusted About X?". In *Proceedings of the 14th National Computer Security Conference,* October 1991.

[5] Epstein, McHugh, Orman, Pascale, Marmor–Squires, Danner, Martin, Branstad, Benson, Rothnie. "A High Assurance Window System Prototype". *Journal of Computer Security,* 18 Jan 1994.

[6] National Computer Security Center. Trusted computer systems evaluation criteria. Technical Report 5200.28–STD, Fort Meade, MD, DoD, December 1985.

[7] Andrew S. Tanenbaum. Case Study 4: Mach. Pages 637–680 in *Modern Operating Systems.* Prentice–Hall, Inc. 1992.

[8] Johannes Helander. "Unix under Mach, The Lites Server". Master's Thesis 1994.

[9] Boykin, Kirschen, Langerman, LoVerso. *Programming under Mach.* Unix and Open Systems Series. Addison–Wesley, Reading, 1993.

[10] Keith Loepere. *OSF Mach Approved Kernel Principles.* Open Software Foundation and Carnegie Mellon University, June 1993.

[11] Keith Loepere. *OSF Mach Approved Kernel Interfaces.* Open Software Foundation and Carnegie Mellon University, June 1993.

[12] Secure Computing Corporation. *DTOS Mach Kernel Interfaces.* Derived from *OSF Mach 3.0 Kernel Interfaces Document*, Edited by Keith Loepere. Secure Computing Corporation, December 1993.

[13] Spencer E. Minear. "Providing Policy Control Over Object Operations In a Mach Based System". Secure Computing Corporation, April 1995.

[14] Saydjari, Turner, Peele, Farrell, Loscocco, Kutz, Bock. "Synergy: A distributed, microkernel–based security architecture.' Technical Report, Version 1.0. INFOSEC Research and Technology, November 1993.

[15] Secure Computing Corporation. *DTOS Users Manual.* Secure Computing Corporation October 1995.

[16] Richard A. Kemmerer. "Computer Security". In *Encyclopedia of Software Engineering.* 1993.

[17] Edward Amoroso. *Fundamentals of Computer Security Technology.* Prentice–Hall, 1994.

[18] Secure Computing Corporation. *DTOS Formal Security Policy Model.* Secure Computing Corporation. January 1996.