Roundhouse: A Security Architecture for Active Networks

Cynthia E. Irvine
Naval Postgraduate School
Center for INFOSEC Studies
and Research
Monterey, CA 93943

William R. Shockley
Cyberscape Computer Services
1885 Franklin Street
Lebanon, OR 97355

15 May 1998

Abstract

We describe a high-assurance framework for actively networked clients and servers. Called Roundhouse consists of the following elements:

1. *Pinkerton*, a comprehensive model for the implementation of distributed protection domains that provide for robust protection in an active networks environment;
2. *Iron Horse*: Functional and security design of a kernelized host providing essential ring-based protection, packet authentication, and cryptography services for higher layers.
3. *DEPOT*: Specification, design, and prototype implementation on a PC base of the framework and initial content of dynamically modifiable servers. The intent is that DEPOT clients and servers would take advantage of platform protected modes where available (e.g., Windows NT, Iron Horse) leading to client-server computing in a network of heterogeneously trusted hosts.
4. As a general facility for installing and managing application ``hooks" DEPOT incorporates the following key new ideas:
    - the division of sets of hooks by module,
    - the partial ordering of modules,
    - binding hooks to network names, and
    - the provision of a run-time model of module behavior with a visible state machine model that abstracts and externalizes the dynamic behavior of that module.

The architecture is unique in that it composes strong and weak systems securely and permits the dynamic retooling of executing software.

Executive Summary

We are targeting C^4I and OLTP networked systems because they are immensely valuable. We focus on these because they offer the greatest opportunity for cost reduction:

- Maintenance and retooling costs are extremely high and outweigh development costs.
- They are vulnerable, relatively unprotected national assets.

Our long-range goal is to permit an order of magnitude reduction in protection and retooling costs. (Protection cost is expected loss due to penetration.)

Our solution, Roundhouse, involves the following architectures:

- Pinkerton, a comprehensive model for distributed protection domains.
- DEPOT, a distributed operating system extension for C^4I/OLTP applications.
- Iron Horse, a high assurance kernel supporting DEPOT for use where "ground zero" points of vulnerability must be protected (e.g., master key repository)

Roundhouse employs a highly innovative blend of existing and new technologies to achieve its overall goals. Each of the following plays a critical role.

- Dynamic linking techniques (1960s)
- Multi-state hardware based protection (1960s, re-emerging)
- Transaction processing (1970s)
- Split address space technology (1970s)
- Event-oriented programming (unique to this architecture)
- Scalable distributed authentication (1990s)
- Active networks (1990s)
- Answers the question: where do you put the crypto keys? (unique to this architecture)

The Pinkerton architecture incorporates

- A sound, conservative access control model for protection of distributed resources
- Prevents critical resources from migrating to weakly protected domains or hosts
- Prevents critical resources from being contaminated or modified from weakly protected domains or hosts
- Permits strongly protected servers to service weakly protected clients
- Completes authentication architectures by specifying protection grades for functions

The DEPOT architecture supports

- Both queued and RPC-oriented transaction processing
- Continuous system operation
- Supports dynamically customizable system environment per server
- Provides strong support for encapsulation of legacy applications

The Iron Horse architecture includes

- Minimized security kernel optimized for transaction processing
- Includes essential TP support functions (e.g., logging, secure multi-threading)
- Offers optional support for military secrecy policies
- High degree of concurrency inside the kernel

In summary, this architecture is unique in providing

- An architecture that composes strong and weak systems securely
- Dynamic retooling of executing software

1 Introduction to the Problem

The following scenario is an example of the real world problem domain that can be supported with secure active networking:

The Universal Command and Control System (UCCS) is a highly distributed system involving numerous platforms and commands, supporting military operations worldwide. The Highly Reliable Software Production Facility (HRSPF), recently assigned responsibility for improving system performance, has generated update 3.01, an update to routing algorithms intended to be installed in thousands of routers throughout the system, including routers serving highly critical processing nodes. The update introduces a critically needed performance improvement, so that time is of the essence. Accordingly, it is determined that the systems' active network capability will be used to distribute and install the update. This capability permits information packets to be flooded throughout the network without requiring the creation of individual sessions between HRSPF and each router. Whenever a packet traverses a router the update is extracted, validated, and loaded. In some relatively short time, most routers will have been updated.

Our approach addresses the following needs:

1. It is highly undesirable to be required to remove a router from service before installing an update. The DEPOT component will define an architecture for flexible servers that can be safely and dynamically retooled, without risking loss of a packet or message, entering an inconsistent state, or having to disable the server in order to recompile or re-link program code. We reach this goal by creating a useful blend of transaction processing (TP), dynamic linking, and object-oriented technologies.
2. It is unrealistic to suppose that attempts to introduce spurious active packets into the network will not occur. In order to permit the use of such technology in support of mission-critical processing, a sound, highly reliable approach for differentiating valid from spurious packets must be available. Although cryptography and access controls provide the needed technologies, these must be incorporated into a workable protection architecture that can be independently reviewed and is demonstrably sound. Moreover, the architecture must accommodate the employment and inter-operation of host platforms having a wide range of trust characteristics. Our architecture addresses this need by providing a detailed model addressing protection in a distributed system composed of heterogeneously trusted host platforms.
3. Cryptographic solutions addressing the trusted distribution of software rely on the existence of some repository where highly sensitive keys may be stored. Ultimately, the security of any cryptographically protected system is only as good as that afforded to its most sensitive keys. Our architecture therefore includes architectural work for a high assurance host, Iron Horse. Roundhouse refers to the combination of DEPOT running on the Iron Horse kernel.

The problem domain addressed by Roundhouse is not confined to military command and control (C^4I) systems. Roundhouse and DEPOT hosted on conventional platforms is intended to support On-Line Transaction Processing (OLTP) applications of all varieties. This style of application design is widely used today in many commercial sectors such as energy, banking, finance, transportation, and telecommunications to perform critical business functions. Of course, the protection and reliability of these sectors is of national concern [2].

1.1 Military and Civilian applications for DEPOT

Similar requirements have driven civilian OLTP and C^4I military systems towards remarkably similar architectural styles, which we call the Transaction Processing (TP) paradigm.

Its characteristics are as follows:

- A unit of work is not accomplished by one process from start to finish. Rather, it moves through the system from one specialized process to another. Each process is furnished with just the resources it needs when initialized to save OS calls for requesting resources.
- Two control strategies are frequent for TP applications. One strategy is to manage a work item using Remote Procedure Calls (RPC) or similar facilities: i.e., a "master" server process divides tasks into subtasks, farms them out to subordinate servers, then collects and collates the results. The difficulty with this strategy is that it often unnecessarily centralizes control and leads to unnecessary network traffic. Accordingly, a second control strategy, modeled as a set of workstations coupled by queues, through which work flows as through an assembly line, is also common. This strategy has the benefit of decentralizing control and rendering the application more robust, but can make it hard to track work.
- Multi-threading – the multiplexing of multiple independent execution points (each with its own stack) onto a single address space – is very common. The usual reason given for multithreading is that it permits one to substitute lightweight context switches for heavyweight switches and thereby increases throughput. (This is true, however, only if multithreading is used intelligently.) It is universally understood that multi-threading poses a significant problem for operating-system level security, where address space isolation is the basic technique for enforcing run-time access controls.
- Generally, typical performance goals might be stated as:
    1. To achieve stated end-to-end average latency.
    2. To preserve this latency under high load conditions (e.g., up to 80% or so of theoretically maximum capacity).

The optimization techniques used for TP, some of which are described above, work well when the work to be done is fairly stereotyped, because it then becomes possible to tailor processes into "specialists" and route work to the processes specialized to do it. Unfortunately, because the TP paradigm and time-sharing paradigms are not very similar, the relationship between TP applications and time-sharing OS are often fairly uneasy. It is not uncommon to find the application overriding, disabling, or working around operating system resource allocation, scheduling, and security policies and re-implementing them as part of the application.

Several general forces then converge to render many TP application systems very inflexible and costly to change or maintain:

- They are intrinsically large programs: too costly to replace very often.
- The application logic tends to be heavily interlaced with calls to functions associated with resource management, security, scheduling, and so on. Maintenance is error-prone, costly, and time-consuming because a change potentially impacts these system functions in mysterious ways.

- Because the application is mission- or business-critical, it is costly (or deadly) to shut the application down for maintenance.

As well as incurring significant costs for industries already using TP, this general lack of flexibility has frustrated the use of TP techniques by industries that have less stereotyped work flows: e.g., hospitals, pharmaceuticals, information providers, professional services and the like.

Considering these factors, we have identified the following as important long-range research goals for Roundhouse:

- To achieve a radical improvement in the maintainability of new TP applications by divorcing computational functions as completely as possible from logic integrating the function into a larger system.
- To embrace (not merely tolerate) the requirement to effectively encapsulate legacy servers and applications.
- To provide a malleable TP-oriented framework that permits dynamic reconfiguration of work paths, and retooling of individual system environments for servers on a per server basis
- To investigate hardware support for secure multi-threading
- To investigate what functions must be kernelized to support TP

These goals have strongly influenced our technical approach.

2 Roundhouse Sketch

Roundhouse involves the use of active networking techniques to support Command, Control, Communications, and Intelligence (C^4I) and On-Line Transaction Processing (OLTP) application systems. We have targeted this class of systems because it includes most so-called critical infrastructure systems. The security architecture includes the following elements:

- *Pinkerton*, a comprehensive model for the implementation of distributed protection domains that provide for robust protection in an active networks environment;
- An extensible, object-oriented interface architecture for a class (*Depot*) of objects encapsulating individual C^4I/OLTP server processes.
- An architecture and prototype design for a high-assurance run-time kernel (*Iron Horse*) hosted on an ISA or extended ISA-compliant hardware base tailored to support C^4I/OLTP processing. *Iron Horse* is used in the *Pinkerton* architecture to support security-critical management and storage functions (e.g., key management, and storage of root keys, and storage of critical certificates).

The following sections outline our technical approach to developing a high assurance active networking architecture.

2.1 Nomenclature

The primary facility seen by designers and programmers is called DEPOT. DEPOT is an acronym standing for "Distributed Extensible Processing Objects for Transactions" and refers to both the software implementing the DEPOT object class, and objects of the class itself. Roughly speaking, a DEPOT object is an encapsulated server with a server-specific, extensible execution environment. Programmers are able to define *dynamic installed service sets* (DISSs) for the

Depots. The remaining architectural names (Iron Horse, Roundhouse) are not acronyms, but simple code names.

Since one of the primary goals is usability on a variety of platforms, DEPOT is portable. The degree to which DEPOT objects and their contents are protected will naturally vary from platform to platform, depending upon the underlying protection mechanisms made available by the host. Iron Horse is a specialized platform intended to provide DEPOT with strongly enforced, hardware-assisted local protection capabilities. Roundhouse is the code name for our project, encompassing the entire set of architectures to be produced for both DEPOT and Iron Horse.

2.2 Roundhouse Security Overview.

Roundhouse security is decomposed into two parts that are handled quite differently.

- **Protection** refers to the enforcement of access controls to protect *Roundhouse* components (i.e., host system and *Roundhouse* code and data) using the strongest protection mechanisms provided by the host hardware and operating system. A key feature of our protection model (*Pinkerton*) is that it explicitly accommodates hosts with heterogeneous protection and authentication mechanisms (including those providing no hardware-supported protection at all). Our strategy thus differs in kind from many other efforts. Rather than trying to design a software-only protection solution [3] that will provide the same protection quality wherever it runs, we provide a means for administrators to grade the systems they have, allocate critical services and data to hosts of appropriate grade, and then operate the heterogeneous network safely. The protection model provides optional support as well for the enforcement of military-style mandatory access control policies [1].
- **Application-level security** refers to the enforcement of controls, including, but not limited to, traditional discretionary access controls (DAC) protecting user-level data maintained or processed by *Roundhouse* applications. By definition, such access controls are type-specific [5]. In order to provide a general and extensible environment for application-level security, we do not wish to constrain application security policy or its administration. Therefore *Roundhouse* provides a flexible framework that permits suitably authorized administrators to install or replace such controls. As a special case, *DAC* and audit for *Roundhouse* objects themselves are provided as administrator-replaceable software units using exactly the same mechanisms available for any *Roundhouse* application administrator. A key notion driving the application security framework design is that the modules making the access control or audit decision are dynamically linked to the application proper, which is responsible only for identifying when an access check is required. The currently linked access control module evaluates the request and passes to the application an *action indicator* telling the application whether to grant or deny the access. Audit log events are transparently generated if required at the same time. Thus, a properly designed *Roundhouse* application can achieve a rather high degree of policy transparency.

  The architecture will support ``generic" ACL-based controls for objects encapsulated by the DEPOT core (queues, modules, and so on). Appropriately authenticated active packets may be used to modify, remove, or replace the default controls with application-specific controls on these objects.

  A problem with identity-based controls is that each layer of application software creating new applications must re-implement them for new abstractions. The DEPOT architecture facilitates this by permitting application-level agents to expose transitions representing the need for an access check. This affords the application designer the opportunity to perform the check in a specialized ``add-on" agent subscribing to the access check transition. The advantage of using this capability is that the access policy is not ``hard-wired" into the application logic and so can be easily replaced or modified. Naturally, user-level requests do not receive enough privilege to modify the ring-protected identity-based security agents.

The event-based architecture can be used in much the same way to accommodate audit: one can dynamically install an audit agent that captures and stores information from the current event and request context transparently to the application agent inducing the event.

3 Pinkerton Protection Model

Pinkerton is a high-level protection model that specifies locally enforceable rules for building a *protection-safe* network from individual *Roundhouse* hosts. A key feature of the *Pinkerton* model is that it permits the use of heterogeneous hosts (i.e., using different internal protection mechanisms, including none). What is meant by “protection safe” is that a more highly protected server, running on a node such as *Iron Horse*, can safely provide services to all of its clients, even those running on a host with weaker protection. In order for a less-trusted client to subvert a server, the server's internal protection mechanisms would have to be overcome: it cannot be done simply by sending the server a spurious "active packet", management request, or service request containing a Trojan Horse. *Roundhouse* core services uses cryptographic authentication mechanisms to label incoming packets and requests and carefully segregates system-level and user-level packets and requests: the locally available protection mechanism is used thereafter to separate them. Ultimately, protection is based upon key distribution: a "weak" node never is given a key that would allow its traffic to be labeled elsewhere as from a stronger node, and it cannot gain such a key without subverting a node of the stronger protection class.

3.1 Protection Policy Design

Pinkerton actually models an enumerably infinite class of potential protection policies. The system security officer selects one of these policies by defining a set of abstract *protection classes,* defining a partial ordering over these classes representing their intended trust relationship, and allocating nodes and internal protection domains to each class. (Relatively obvious constraints are imposed by the model on the allocation based upon intrinsic privilege relationships, software dependencies, and so on: one cannot make an allocation that permits a more trusted domain to depend upon a less trusted one.)

Additionally, for each class, one defines an authentication and encryption profile that *Roundhouse* will require for management-level operations on entities (including *Depots*) of that class. *Roundhouse* distinguishes between client requests/replies to a *Depot* (which are simply presented to the *Depot* on its queues) and *Depot* management requests (which are interpreted and executed by the *Roundhouse* core).

Pinkerton addresses the issue of cooperation under conditions of mutual suspicion [4,6]. Administrators may set up new, shared classes, or agree to treat designated classes (one from each administrative domain) as equal, or both. The modified policy settings will then permit the desired degree of sharing.

Pinkerton permits servers to service clients not previously known by assigning such requests to an administratively defined protection class (typically "protection-low").

3.2 Intended use

It is possible to define a fairly elaborate, application-specific system with a large number of distinct protection classes using the *Pinkerton* model. Experience with systems incorporating similar integrity controls has proven that this is usually a mistake. For the administration of a small enterprise one might think in terms of only a few groups of users (e.g., insiders and outsiders) and define a correspondingly small set of protection classes. A good motto is "less is more, but none is too little".

3.3 Protection Policy

In general, it is expected that most aspects of security, both in the narrow sense of traditional access controls and the broader senses being discussed by the active networks community will be implemented as application-specific *dynamically installed service sets* (DISSs). However, the issue of self-protection is such an important one that controls addressing this need are built into Core Services.

An important consideration is that the quality of protection available for DISSs and applications varies from platform to platform, depending upon the underlying host. On the one hand, we want to exploit and leverage strong isolation and protection capabilities where they exist (e.g., on Iron Horse, Windows/NT, VMS, etc.). On the other hand, we do not want to unnecessarily prohibit communications between stronger and weaker machines. We therefore need a coherent, demonstrably correct model that prescribes which requests are to be honored, and which are to be ignored in order to maintain the integrity of strongly protected DISSs. This model must meet some common-sense objectives:

- **Support management requests**. We would like to be able to dynamically modify, replace or install DISSs, using active network techniques, if, and only if, the packet or request implying such an action emanates from a sufficiently trustworthy source.
- **Support service requests**. If a request packet does not imply modification of the DEPOT configuration, but simply requests services, the range of sources we are willing to service is presumably broader. Meeting the first objective by requiring all packets to meet the criteria enforced for "management" packets is therefore inappropriate.

The key problem that must be addressed for any framework hosted on many different platforms is that of understanding how strongly protected instances (e.g., hosted on Iron Horse) may successfully provide services to poorly protected instances (e.g., hosted on a PC running MSDOS) without excessive risk to its own integrity. Three major issues (and a host of minor ones) must be addressed:

- At the local level, how do we prevent the less trusted client from stealing any privileges the more trusted server might possess?
- At the level of an administrative domain (i.e., a set of nodes managed by a single enterprise) how do we know that the composition of individual solutions is coherent and meaningful?

- At the level of communications among different administrative domains, possibly with quite different notions of what it means to be trustworthy or approaches for protecting data, how may one arrange for the safe sharing of information and services without exposing either domain to uncontrollable risks? (This is the problem of mutual suspicion.)

As the model tackles a difficult problem in some detail it is necessarily somewhat complex. The remainder of this section describes some of its salient features. Space precludes sketching the implementation approach in great detail.

The key insight is that what must be accomplished in the distributed environment is exactly what *is* accomplished locally by a ring mechanism. Rings provide a means whereby service requests by less trusted subjects (i.e., service calls) can be safely honored (after the appropriate set of argument validations is performed) without compromising the integrity of a properly implemented trusted subject (the same process executing in a more privileged ring). The primary difference is that in the distributed environment, it is impossible for a local protection domain in one node to be *privileged* (in the sense of having direct access) with respect to objects in another node: everything must be passed as argument or result. But this difference is a difference in the safe direction. Ultimately, our argument that the model is correct is based upon the belief that rings, properly utilized, work, as our model is in the large a "distributed version of rings". Most of the actual complexity derives from the need to show how authentication and cryptography fits into the picture and specification of an adequate set of constraints to deal with differences in mechanism strength.

3.4 Pinkerton Details

The following section (adapted from the proposal) describes the *Pinkerton* model in greater detail.

3.4.1 Local Protection Domains (LPDs)

Local Protection Domains are the basic building blocks from which the desired network–level abstractions are built. Intuitively, an LPD is an execution domain or its like. Unlike more traditional models, we accept LPDs that are protected by mechanisms of various strengths. Policy constraints are eventually defined that permit administrators (not applications, users, or programmers) to take these differences into account. For example, on one node we might have LPDs protected by address space separation with hardware assistance (e.g., Iron Horse); on a second, local protection domains built with server-owned ACLs (e.g., VMS or Windows NT); on a third, virtual domains managed by a closed language interpreter (e.g., a Java machine); and on a fourth, a single monolithic LPD (e.g., an ordinary PC).

Within each node, a partial trust ordering is given for its LPDs based upon privilege and dependency relationships, in the natural way (i.e., if A is privileged with respect to B, and/or B depends upon A, A must be as or more trusted than B).

3.4.2 Security Administration Domain (SADs)

A security administration domain is a dynamic set of nodes (and by inclusion, their LPDs) that shares and enforces an identical, administratively defined protection policy. Services may be extended to nodes outside the SAD. No implication is intended that the SAD have any purpose other than to establish the range of nodes enforcing the same administratively defined policy.

3.4.3 Extended Protection Domain (EPDs)

EPDs are network-visible objects representing groups of LPDs defined by the SAD administrator to be equivalent, for the purposes of protection – that is, equally trustworthy. The set of EPDs is partially ordered, with an ordering consistent (in the natural sense) with the partial orderings of the LPDs within nodes. We denote by the expression *E(A)* the unique EPD containing LPD *A*. Basically, two rules must be enforced.

- LPD *A* may use services provided by LPD *B* only if *E(B)* is as, or more trusted than *E(A)*. Obviously, if this rule is violated, the trustworthiness of LPD *B* will be compromised because it now depends upon a less trustworthy component.
- LPD *A* may accept management commands (including, but not restricted to, commands to install or modify the existing DISS configuration) from LPD *B* only if *E(B)* is as, or more, trusted than *E(A)*.

In addition to assigning LPDs to EPDs, the administrator defines for each EPD a set of indicators flagging the cryptographic and authentication, and other protection checks that must be associated with communications involving *managing or servicing* the LPD (e.g., for adding an DISS that executes in the LPD).

Thus, enforcement of the rules broadly expressed above involves several ideas:

- Each and every flagged criterion must be passed before a packet will be accepted as a valid service response or management directive.
- Included among these are source authentication, determining that the desired cryptographic protocol was applied, and whatever other criteria might be built into the model (exactly how many criteria will be applied is a research issue).

We believe that this approach is consistent and sound, because it can be mapped to abstract models of ring-like protection systems. In our interpretation of the model for distributed protection, these operations (in part) turn into the operations of "selecting and using indicated authentication and encryption protocols".

3.5 A Note on Application Security

Security for Roundhouse supported-applications is decomposed into two parts, the first of which has been described as "Protection". We have performed this decomposition because we are convinced that protection concerns are of sufficient importance, and of sufficient complexity in the network environment, to demand a single point of administration where a knowledgeable administrator can make decisions regarding the protection structure of the SAD that cannot be overridden by local node administrators, programmers, or users. The protection model described can, if used thoughtlessly, quickly lead to undesirable complexity: practical SAD administrators will typically define only a few EPDs.

The remaining security concerns (and there are many) we treat as "application-specific" security. The default security DISS will provide ACL-based discretionary security for DEPOT objects proper. These controls, as for any DISS, are dynamically replaceable (provided one is authorized under the current ACL to make the replacement and the replacement software passes the protection checks it must to satisfy the protection policy.

One can imagine creating specialized DISS modules permitting cooperating instances to enforce many of the more exotic policies being discussed within the active network community. The notion of enforcing context-sensitive policies based upon flows (considered as threads of execution in their own right) is also very intriguing: we believe that this research direction leads to enforcement of typical business rules.

4 Iron Horse

Iron Horse is a high-assurance platform intended to support critical security functions in our active network architecture. It will provide significant protection for instances of DEPOT, the framework for dynamically modifiable servers. Iron Horse provides a basis for establishing protection domains and is, itself, self-protecting. As a DEPOT host, it is intended to support the entire spectrum of DEPOT processing. From the point of view of protection, the Iron Horse system can be decomposed into the following subsystems, beginning with the most privileged and proceeding to the least.

4.1 Non-discretionary Integrity Domains

IRON HORSE is intended to enforce locally a coarse-grained non-discretionary integrity policy, primarily to afford strong assurances that hosted DEPOT servers cannot be corrupted by spurious active packets or server management commands. Users may wish to take advantage of the capability to partition user-level data into non-discretionary protected subsystems.

It is useful to state a litmus test for evaluating the overall success of a security architecture:

> *It must be possible to protect, with a strong degree of confidence, critical enterprise data and the servers manipulating that data while continuing to offer less-protected service to clients executing on less trusted, or untrusted platforms.*

We believe that a system approach failing to meet the first criterion cannot be called *secure* in any justifiable sense, while a system approach failing to meet the second criterion cannot be called *operationally useful* in any justifiable sense.

In order to meet the litmus test, it is essential that DEPOT-encapsulated servers be able to receive and process queries emanating from less trusted clients. This implies that the servers must be trusted with respect to a pure Biba [7] integrity policy.

The Iron Horse solution is based upon the observation that the transfer of a service request to a DEPOT server is analogous, in every respect, to the transfer of gate call arguments from a less to a more privileged ring. How to build a gatekeeper that effectively sanitizes arguments is well understood. In the architecture contemplated, IRON HORSE and DEPOT agents executing in highly privileged (and thus, highly protected) rings will coordinate to perform this function, resulting in a high degree of confidence that spurious privileged active packets are processed in the correct ring and that untrustworthy user-level requests are executed in an outer ring not privileged to modify critical data nor initiate management or reconfiguration actions. The partial ordering of DEPOT modules is key to making this architecture work, as it allows us to allocate modules to execution rings in a meaningful way while drawing on experience in extensible protection architecture.

4.2 Hardware Base

A primary goal of the Iron Horse effort is to investigate the exploitation of multiple hardware-supported address spaces or split address space technology to reduce the cost of inter-domain switching. The Intel iAPX 86 CPU architecture already supports a simple form of MAT with two per process address spaces (i.e., the LDT, or local descriptor table, and GDT, or global descriptor table), although few commercially-available general purpose Operating Systems have chosen to exploit this capability in the way we envision. Accordingly, the initial hardware target for Iron Horse will be ISA-bus compatible systems (e.g., ordinary IBM-compatible PCs) as this hardware configuration is widely available, relatively inexpensive, and supported by numerous development environments. Although we are targeting IBM-compatible hardware, there is no current intent to design a DOS- or Windows-compatible API into Iron Horse.

Ultimately, CPUs supporting a larger number of independent address spaces may become available. Such CPUs would become an attractive target for further experimentation.

4.3 Iron Horse Kernel

Residing in the most privileged protection level (Privilege Level 0) is the Iron Horse Kernel. This might concisely be described as a relatively traditional kernel that virtualizes access to basic system resources and provides a virtual environment for higher-level processes. The kernel will not be “monolithic” in the normally understood sense (i.e., a single monolithic “critical section”). Instead, it will support a high degree of concurrency. The principal researchers have considerable practical experience in designing and implementing non-monolithic kernels.

The security policy enforced by the kernel will be a highly-configurable label-based policy supporting the definition and protection of a set of partially ordered protection domains, with domain isolation based upon management of per process LDT and GDT images and the hardware-based ring mechanism local execution domains.

The primary innovations to be incorporated in the Iron Horse Kernel design are these:

- Exploitation of LDT management functions. This permits the implementation of secure multithreading in the DEPOT layer.
- Support for essential transaction-oriented functions. In particular, the kernel will export a logging function and a set of functions allowing it to respond (as a local resource manager) to transactional commands issued by a DEPOT level transaction manager (e.g., begin transaction, end transaction, commit, abort, checkpoint, savepoint).
- Support for lightweight context switches in architecture exploiting split address space technology. Transitions between high assurance domains have traditionally required full context switches, resulting in lowered system performance. Threads have been developed to avoid context switching. Because threads share address spaces, hardware mediation cannot be used to place threads into different protection domains. We intent to work on reducing the cost of context switches by exploiting the multiple descriptor tables offered by the iA86 CPU. This permits the implementation of secure multithreading in the DEPOT layer.
- Optimized to support the efficient scheduling and execution of DEPOT objects. Put another way, the kernel will be designed for TP-style processing rather than, for example, time-sharing.

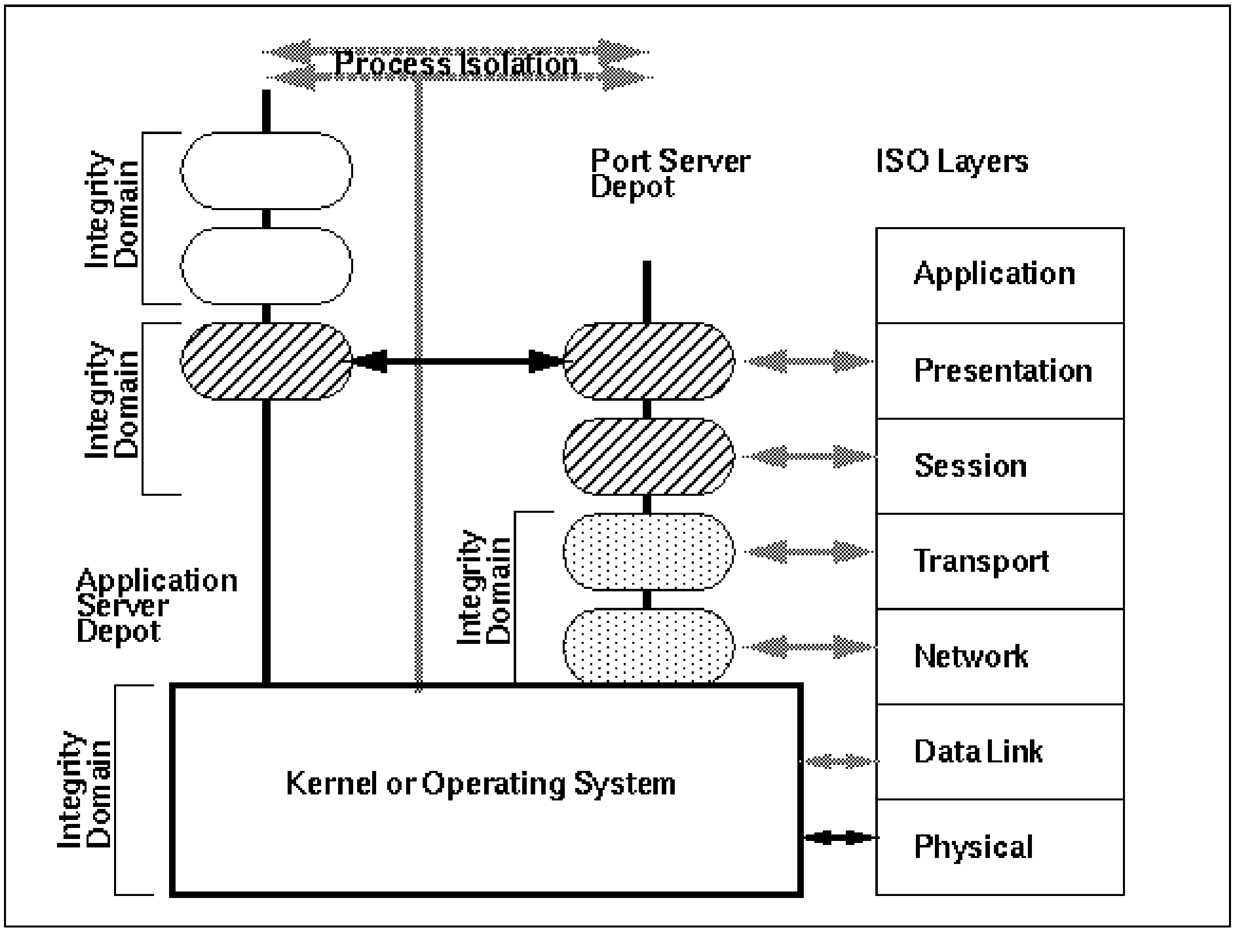

Figure 1. Iron Horse and DEPOT integrity domains.

- Although "low-level" synchronization primitives solving the secure reader-writers problem will be exported (e.g., eventcounts, sequencers) it is expected that synchronization of most applications will be via the DEPOT level transaction manager. The primary use of low-level primitives would be to support the development of application-level resource managers.

The second point deserves further elaboration, as the full import of the requirement may not be immediately obvious. Modern distributed transaction processing architectures are based upon the cooperation of a higher-level transaction manager (which will be part of DEPOT) and lower level resource managers providing access to durable and recoverable data. Both depend upon a basic logging capability. Making the kernel a resource manager means that when a failure occurs, it will recover (in coordination with other participating nodes) into a state that is not only secure, but consistent with all distributed transactions that are recovered. Figure 1 illustrates Depots on a base supporting integrity domains for applications.

5 DEPOT

As one means of providing isolation of DISSs, it is desirable to permit DEPOTs to execute in distinct LPDs if made available by the host machine. (How DISSs are allocated to these domains remains the province of the DEPOT administrator.) A DEPOT with its DISSs can span multiple

LPDs. However, intrinsic functional dependencies will constrain how this can be done. An important research goal will be to investigate such constraints.

5.1 The DEPOT Processing Paradigm

DEPOT is an alternative model to that of simply loading a statically linked, opaque code segments into the address space of a run-time process.

The DEPOT model replaces the notion of a process with that of a durable, recoverable active processing object called a Depot. It is intended that Depot objects be well-suited for implementations of remotely accessible servers. The internal process structure of a Depot is hidden from its clients, who communicate with it using queue-based messages.

A typical Depot makes the following major sub-objects visible to clients (although only clients involved in managing or reconfiguring the Depot will be interested in many of them):

- one or more message queues that handle all communications with the Depot. Schemas for each queue are also visible.
- a collection of agents (code objects) that collectively implement the behavior of the Depot. Agents are grouped into a partially ordered set of named modules.
- For each module, an externally visible state machine model (SMM) that abstracts and externalizes the dynamic behavior of that module. In a sense, SSMs are to behavior as schemas are to data. The transitions of the SSM are called events. Events are typically associated with internal calls for service (e.g., system calls such as alloc, enqueue, dequeue, etc.) Some events are induced by the Depot system (e.g., execute initial agent).

The most unusual characteristics of the above model are the following:

- Agents may be dynamically added, removed, or replaced by placing appropriately formatted management messages (i.e., active packets) on the input queue.
- Similarly, the SMM may be dynamically extended in various ways (just as new fields may be added to a data schema), subject to straightforward consistency constraints.
- Agents do not directly invoke one another. Rather, agents may subscribe to an existing event. An executing agent typically induces an event defined for its module by making a system call. It is possible for an agent to explicitly invoke a lower level agent by name: however, that is turned into an event by Depot so that subscribing agents may seize the thread of control. To the inducing agent, the call looks like an ordinary subroutine call and return: in fact, any number of subordinate agents may have executed between the call and return.

One might characterize the system as a general facility for installing and managing application ``hooks'': the key new ideas are the division of sets of hooks by module, the partial ordering of modules, binding hooks to network names, and the provision of a run-time model of module behavior with the SSM. It is just these new features, however, that give us a means for securing the Depot.

Figure 2 illustrates the flow of control within a Depot:

(a) Agent 1 makes a system call to allocate memory;

(b) as a result Agent 1 induces a state transition;

(c) the transition takes place in the SSM;

(d) Agents A and B are invoked as subscribers to the SSM of the Module containing Agent 1;

(e) additional state transitions and invocations may take place in a chain of modules;

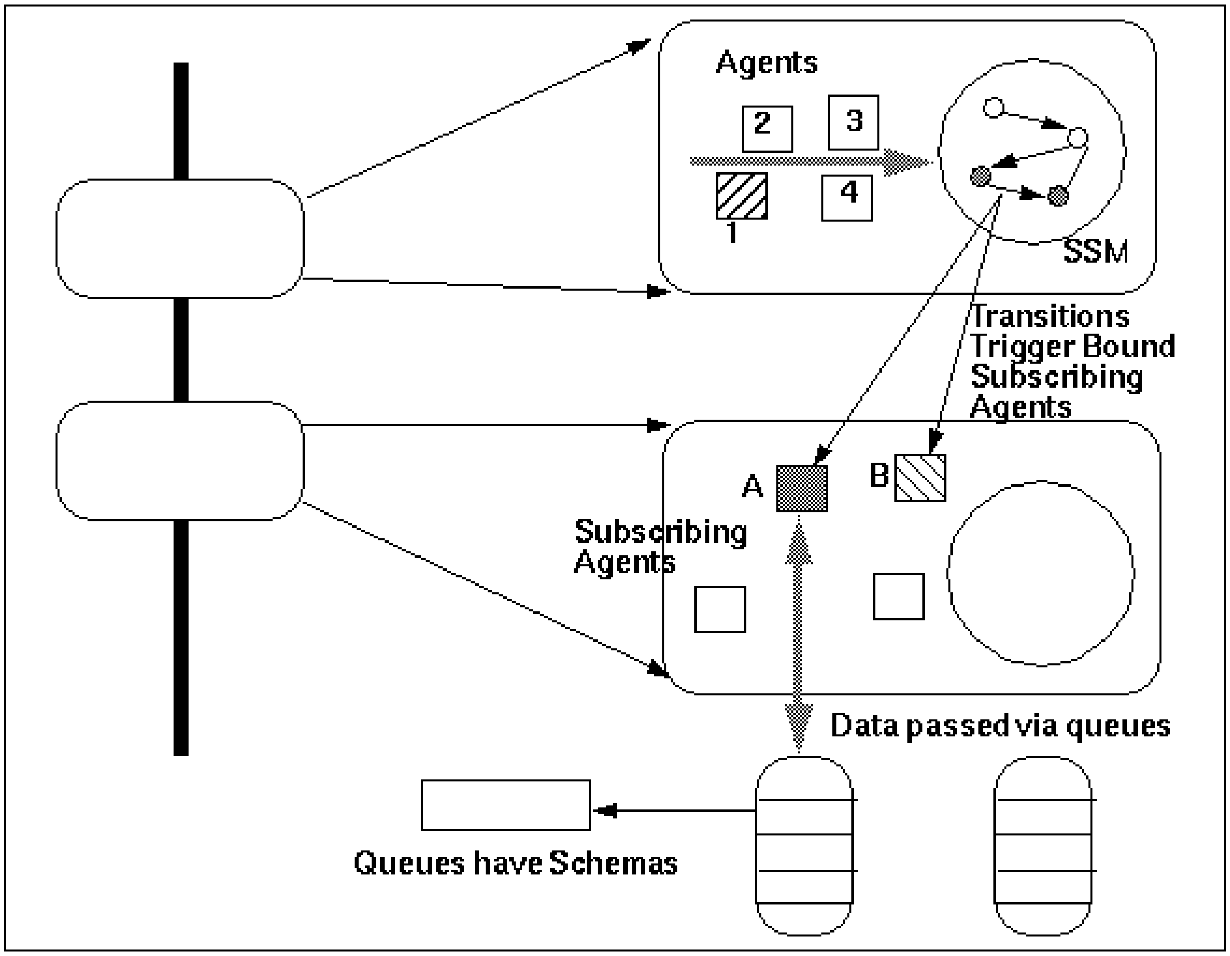

Figure 2. Depot Flow of Control

(f) ultimately the system call to allocate memory is executed; and

(g) as in a threaded language, return is made to Agent 1.

The DEPOT software is divided into three parts: the integration library (consisting of that code that must be rewritten to re-host DEPOT), DEPOT Core services, and a library of standard extensions defining default DEPOT object behavior. The standard library will be discussed in the next section. This section describes the processing model to be implemented by Core services.

A DEPOT is an extensible processing object with the following properties:

- It is persistent. That is, it contains durable, recoverable state that gives it an identity that spans network sessions and host downtimes. It has a network-visible name.
- It is stationary. That is, it resides on a particular host.
- It is queue-driven. Each depot is associated with its own standard input queue, from which it draws requests, and deposits its responses on its own standard output queue.

Conceptually, a DEPOT may be thought of as a specialized server: its application-level code is basically a loop within which a request is dequeued, processed, and enqueued. "Add-on" behavior is via dynamically added extensions. The goal is to completely decouple the application logic from the environment: it refers only to programmer- or system-defined names (e.g., *stdin*, *stdout*, *stderr*).

One may contrast a DEPOT with a more conventional object-oriented framework, which consists of a set of defined object classes that can be specialized to work in a given target environment.

"Specialization" means, ultimately, generating a brand-new object with some behaviors different from the old. This may work well for new applications, but it is not a useful paradigm for the integration of legacy servers written in languages that know nothing of inheritance. Rather, non-application specific behaviors are added to DEPOT servers in the form of dynamically linked, physically replaceable extensions. Because legacies can't possibly invoke new extensions by name, invocation of the extension must be based upon occurrence of some event that the operating system can detect: e.g., a system call, program interrupt, or detection of a fault as the program is running. We therefore provision the DEPOT with a run-time model of application behavior, in the form of a non-deterministic state machine that represents the possible sequences of detectable events available for potential interception by extensions. These transitions are named. When a new extension is installed, the installing authority specifies by name the set of events that are to trigger the extension. For new applications, of course, events can be defined and triggered explicitly if desired (a facility analogous to the notion of a "software interrupt". This description provides an overview (with much detail suppressed) of what we call "event-based programming" (EBP). For new applications, it is merely cute: but it is a key feature for supporting the encapsulation of legacy servers.

For new applications, we encourage the programmer to think in terms of the following structure: writing the "application proper" as a bare-bone "request processor" that contains nothing but highly-re-usable computational logic. All integration with the large-scale environment is handled by drawing from whatever libraries of extensions already exist. Since these are themselves written in terms of object classes, specialization of library extension objects will presumably be an important means of generating new extensions.

As the number of extensions associated with a given DEPOT is likely to be large, prevention of interference and extension management becomes an issue. A modularization facility is provided in the DEPOT model to help manage extensions.

Programmers define dynamically installed service sets (DISSs). These may be modified at runtime, by the dynamic addition of extensions. The extensions within a DISS may be grouped into partially ordered sets. The partial ordering is intended to provide a means for programmers to identify subsets of DISSs that could be allocated to different protection rings on platforms (such as Iron Horse) where application-level rings are available. A DISS corresponds in much respect to what programmers using multiple-inheritance object-oriented languages call *mix-in* classes (the notion is that a basic object class can be enhanced by inheriting from an enumerated set of mix-in classes). Here, however, the mix-in classes are added dynamically as needed, rather than being statically specified as part of the design of a given DEPOT. It is intended that on platforms supporting multiple local protection domains, DISSs will be executed in different domains to provide address-space isolation. Multiple DISSs may, however, be bound together into a new DISS where the designer wishes to avoid unwanted context switches.

5.2 Standard Extension Library

A framework without content is not very useful. While providing the application code is the programmer's responsibility, DEPOT will provide a library of standard extension modules that collectively provide a minimal, but useful distributed TP processing environment.

The following list (not necessarily complete) identifies some of the standard modules:

- Queue Flow Manager. As described so far, work "magically" appears on input queues and disappears. The Queue Flow Manager is the DISS that performs this magic. Each request is framed with routing control information (e.g., where to return the response). The QFM is triggered by system calls for input and output and uses frame information to forward output and error responses, to induce scheduling events when the input queue is empty, to store responses that can't be immediately forwarded, and to apply a scheduling policy periodically.
- DEPOT object and management (enable, disable, lock, unlock, shutdown, etc.). The semantics of object management will conform to ISO/IEC 10164-2.
- DEPOT object security. This will enforce traditional discretionary access controls constraining who may successfully modify the depot, examine its state, modify its set of extensions, issue management commands, etc. The overall protection policy is also enforced.
- Default instrumentation, allowing DEPOT and queue contents to be examined and modified for debugging.
- Transaction processing, permitting a request to be processed as part of a local or distributed transaction. This facility includes deadlock detection and recovery and failure recovery capabilities. The transaction management architecture will conform to an existing standard.

The position of the library DISSs in the module hierarchy relative to the application code (which is in most respects simply another DISS) has not yet be determined. The notion is that when a new DEPOT is created, it will be furnished with a default behavior (e.g., as an instrumented initial configuration) which can then be shaped into the desired server by dynamically replacing the default application DISS and library DISSs as needed.

Each module, when installed, potentially extends the externally visible API of the DEPOT. That is, not only does it respond to internally induced events but can (if so designed) be sent messages and emit responses on its own input and output queues. It has its own state-machine event description (that, in effect, extends the state machine description of the application by substituting for a transition a subordinate state machine). It is quite possible for a coherent set instantiations installed on various depots to communicate with each other in order to perform collective management and control tasks – we intend to implement much, if not all, of the transaction manager this way.

This leads to an alternative view of the system. So far, we have provided a “DEPOT-centric” description, focusing on a single DEPOT and its local environment. However, if we examine a “modular slice” of a set of cooperating DEPOTs (focusing, for example, on all of the “security enforcement” modules) we can view the system as a layered set of cooperating agents, (DISSs) attached to host DEPOTs, that together exhibit some desired collective behavior. This is likely to be the view adopted by subsystem designers when working on a new class DISSs.

5.3 DEPOT on other platforms

We have described DEPOT as hosted upon a customized platform, Iron Horse. To meet the needs of the OLTP community, it is important, however, that DEPOT be portable to a wide variety of platforms. The DEPOT integration library component maps DEPOT infrastructure abstractions to the host operating system and, if necessary, implements missing functions (e.g., transactional logging).

6 Comparison with Other Efforts

The reader is reminded of our litmus test for evaluating the overall success of a security architecture:

It must be possible to protect, with a strong degree of confidence, critical enterprise data and the servers manipulating that data while continuing to offer less-protected service to clients executing on less trusted, or untrusted platforms.

A system approach failing to meet the first criterion cannot be called *secure* in any justifiable sense, while a system approach failing to meet the second criterion cannot be called *operationally useful* in any justifiable sense.

6.1 Scout

Scout is an advanced communications-oriented operating system [27]. Many of its key abstractions (paths, modules, caches) seem to map directly to DEPOT key abstractions (flows, DEPOTs, queues). Are these two independently developed versions of the same abstract framework? We find that they are not.

Scout focuses upon a very different problem domain. The common thread among its expected applications is that they use embedded computers with relatively fixed programs meeting real-time performance requirements. Scout makes radical use of these common characteristics for many of its optimizations: optimal processing paths and configurations are identified in advance and a customized configuration is then compiled. Scout generally emphasizes optimizations at the lower layers of the network stack although the techniques are applicable at higher layers as well.

In contrast, DEPOT focuses on applications that must retooled rapidly while large sections of code continue to execute. Dynamic linking of extensions is a natural solution to this problem. DEPOTs TP-orientation tends to focus on the higher layers of the network stack although available throughout.

Scout and DEPOT are rather complementary. The notion of using Scout techniques to optimize a cooperating DISS slice is attractive: one could imagine the optimized instances and Scout infrastructure being distributed and installed locally via DEPOT. We plan to investigate such ideas.

6.2 Liquid Software

The Liquid Software effort is focused on facilitating a new paradigm for network computing based upon the notion of highly mobile non-interpreted code [22]. Integral to the concept is the development of a "gigabit compiler" that generates platform dependent code from incoming platform independent, intermediate code at reception speeds. Although the compiler is a major focus, significant effort is also being devoted to system-level issues (such as security and safety) and how such a capability might be used. The target initial platform is Scout, discussed above.

DEPOT represents a less general version of "mobile code": DISS modules would either be interpreted, or compiled before being loaded and linked to the application. Installing the "same" DISS on different platforms would require a separate compilation for each platform and the correct version sent to each target. A Liquid Software compiler on each target host offers an instant solution to this problem and would be immediately useful.

Roundhouse has something to offer to the Liquid Software effort as well: a plausible solution to some (not all) of the security issues they have identified: viz., externally enforced protection domains on selected platforms, and a sound approach for dealing with heterogeneously protected instances of cooperating program instances.

6.3 Cherubim

Cherubim, characterized by its researchers as "a mobile, agent-based security architecture"[11][44], is a project examined in some detail here because the problem domain being addressed is essentially the same as ours: OLTP and C^4I systems. Moreover, the perceptions expressed concerning the need for malleable, application-oriented security policies in this domain correspond to ours. Listed research topics are:

Core Security Services. This may be characterized as a portable crypto-system implementation that provides a Java interface so that portable programs written in Java can gain access to cryptographically secure communications. A complete set of cryptographic services – encryption, decryption, digital signature, and signature verification – is supported. It seems fair to characterize the service as an object-oriented API that, through specialization, permits access to underlying popular crypto-systems.

This effort does not compete with our research, which does not currently involve actual use or implementation of a crypto system. Our protection model assumes (but does not detail) the availability of cryptographic services. Our intent is simply to select and use existing cryptographic technology as the DEPOT effort proceeds. Cherubim Core Security Services may be an attractive candidate for use at that time.

Active Capability Based Authorizations. Traditional discretionary access controls (DAC) are often based upon the idea of an Access Control Matrix: where a Boolean value indicates whether or not a given user may access a given object. *Cherubim* is to replaces the Boolean value with an executable script.

This responds to a genuine need. However, the *Cherubim* team seems to be describing a system in which application-level security is to be designed, implemented, and dynamically modified by programmers. (Programmers like scripts, ordinary people don't.) This corresponds to current OLTP practice. However, many managers responsible for information security view current practice as a significant threat: they would rather place control of application security in the hands of operations staff and system administrators. ***The DEPOT framework, by radically de-coupling security design from application design, permits this***. The programming and design of the security DISS extensions can be performed by a smaller and more trusted group of programmers, implementing controls to be used by operational personnel. A system of cooperating DISS instances permits "single point of management" operation of controls. Finally, operational personnel have complete control over *which* set of available security DISS extensions are coupled to the application servers.

Reflective Security Architecture. This effort is focused on the area of increasing the "programmability" of security features into applications. Modest in scope, the effort is moving in the proper direction of trying to extricate security-critical from application-level code. Efforts include extensions to the Interface Definition Language (IDL) needed to simply specify services, customizable binding methods, and extensions of existing visualization software to data flows (mistakenly called information flows).

While useful, these efforts are not comparable in scope to the truly dynamic binding of policy to application offered by DEPOT. In order to use the binding constructs offered under this topic, the application must still be disabled, re-linked (at some level of abstraction), and restarted. It is progress to bind policy to application at static link time, instead of compilation time. It is even more progress to bind it at and during run-time, as does DEPOT, permitting dynamic policy reconfigurations on the fly. This capability is essential to support the needs of continuously operational applications, such as C^4I.

Active Security Policy. This is an exploration of the range and usability of policies expressible using the Active Capability Based security model. Although we disagree in principle with when such policies should be bound to the application, DEPOT could easily serve as an alternative framework for hosting the actual policies being explored.

Secure Typing System. This theoretical effort focuses on proving security properties using object-oriented theories of types and is of general interest to the community as well as being particularly applicable to Cherubim's other work. The DEPOT effort is not focused on providing application-level policy content beyond that of the default security DISS, and could therefore make only limited use of this work.

In summary, our general criticism of the Cherubim effort is that it relies strongly on the imputed security characteristics of its underlying software base (Java and various network protocols). Java has known problems containing running applets, and Cherubim necessarily inherits them. DEPOT, running in a Java-only environment would be no worse (and no better). However, DEPOT also runs in environments offering hardware-enforced protection, and offers an explicit model of protection showing how heterogeneously hosted instances may safely inter-operate. Cherubim offers no equivalent theoretical abstraction that addresses this difficult problem.

6.4 We compare with competitive alternatives.

Other efforts, [4][29][33][45], seem to fall into two camps:

1. First camp: since the framework will run everywhere, we must solve the protection problem for the worst-case instances – those running on single-state machines. This leads inevitably to the invention of some notion of a protection domain, which must somehow be created by software alone – a known intractable problem. In principle, it is possible to create software-mediated domains either by providing a 100% interpretive environment or (equivalently) passing everything that runs through an unbypassable compiler and verifier (an approach suggested by Liquid Software). Historically, the Department of Defense accreditation community has not been willing to accept completely interpretive solutions for high C^4I applications.

   Even if one assumes success, what one has achieved is the ability to support multiple LPDs on one node: only the starting point for our Protection Model. The problem of how to integrate nodes having protection domains of varying strengths remains. This problem does not seem to be seriously addressed by any competing Java-like effort.

2. Provide security to the granularity of whole nodes (i.e., ignore the potential existence of multiple LPDs in a node and grant or deny access to resources – e.g., a private network – based on node authentication alone. The DVPN project [33] uses this approach. It is common because most existing protocols (e.g., IP) do not provide a means to name or address LPDs within a node: such a capability must be added if a security infrastructure is to successfully cope with heterogeneous platforms. It is not unsound to base network security on node authentication alone, but wasteful and impractical: commanders do not want multiple workstations on one desk.

7 Summary

The *Roundhouse* security architecture incorporates

- A sound, conservative access control model for protection of distributed resources
- Prevents critical resources from migrating to weakly protected domains or hosts
- Prevents critical resources from being contaminated or modified from weakly protected domains or hosts
- Permits strongly protected servers to service not only clients having the same trust characteristics, but less trusted clients as well.
- Completes authentication architectures by specifying protection grades for key distribution and storage functions.
- Provides a framework for dynamically replaceable application-level security that allows the creation of policy-neutral applications that are later dynamically linked to customizable modules defining policies to be enforced.

We believe that this approach is consistent and sound, because it can be mapped to accepted abstract models of ring-like protection systems [38]. In our interpretation of the model for distributed protection, the operations of comparing and/or generating protection labels turns out to include the functions of selecting and using indicated authentication and encryption protocols using protection-related root keys for host identification and authentication. The process of distributing root keys is part of the process for achieving a protected initial state: the binding of a given host public key to a protection class is part of the process of specifying the administrative policy.

Bibliography and References

[1] Department of Defense Directive 5200.1-R -- Information Security Program Regulation, October, 1982.

[2] Report of the Presidential Commission on Critical Infrastructure Protection, October 1997

[3] Anderson, James P., Computer Security Technology Planning Study, 1972, Air Force Electronic Systems Division, ESD-TR-73-51, (Also available as Vol. I, DITCAD-758206. Vol. II, DITCAD-772806), Hanscom AFB, Bedford, MA.

[4] Bhattacharjee, S. Clavert, K.L., Zegura, E. W., An Architecture for Active Networking, Technical Report: GIT-CC-96-20, Georgia Institute of Technolgy, College of Computing, 1996.

[5] Bell, David E. and LaPadula, Leonard, Secure Computer Systems: Mathematical Foundations and Model, 1973, MITRE Corp., M74-244, Bedford, MA.

[6] Bershad, B.N. and Savage, S. and Pardyak, P. and Sirer, E.G. and Fiuczynski, M. and Becker, D. and Chambers, C. and Eggers, S., Extensibility, Safety and Performance in the SPIN Operating System*, Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Operating Systems Review*, December, Vol. 29, No. 5, pp. 267-284.

[7] Biba, K. J., Integrity Considerations for Secure Computer Systems, 1977, MITRE Corp., ESD-TR-76-372.

[8] Black Forest Group, Top Level Security Issues for a Global System of Interconnected Computers. April 1997.

[9] Bobert, W. E. and Kain, R. Y., Practical Alternative to Hierarchical Integrity Policies, 1985, *Proceesings of the 8th National Computer Security Conference*, October, address Gaithersburg, MD, pp. 18-27.

[10] Brinkley, D. L., and Schell, R. R., Concepts and Terminology for Computer Security, in *Information Security: An Integrated Collection of Essays*, IEEE Computer Society Press, Los Alamitos, CA, ed. Abrams, Jajodia and Podell, 1995, pp. 40-97.

[11] Campbell, R., Qian, T., Liao, W., Liu, Z., Active Capability: A Unified Security Model for Supporting Mobile, Dynamic and Application Specific Delegation, Department of Computer Science, University of IL, February 1996.

[12] Clark, D.D. and Wilson, D. R., A Comparison of Commercial and Military Computer Security Policies, *Proceedings 1987 Symposium on Security and Privacy*, IEEE Computer Society Press, April, Oakland, CA, 1987, pp. 184-194.

[13] Dean, D., Felten, E., and Wallach, D., Java Security: From HotJava to Netscape and Beyond, in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, CA, May, 1996, pp 190 - 200.

[14] Denning, Dorothy E. A Lattice Model of Secure Information Flow, *Comm. A. C. M.*, No. 5, Vol. 19, Ref 3, 1976, pp. 236-343.

[15] ------, Department of Defense Trusted Computer System Evaluation Criteria, National Computer Security Center, DoD 5200.28-STD, December 1985.

[16] ------, Gemini Trusted Network Processor (GTNP), Information Systems Security Products and Service Catalog Supplement, National Security Agency, Report No.CSC-PB-92/001, April 1992, 4-SUP-3a.3.

[17] ------, Final Evaluation Report of HFSI XTS-200, National Computer Security Center, CSC-EPL-92/003 C-Evaluation No. 21-92, 27 May 1992.

[18] Engler, D.R. and Kaashoek, M.F. and O'Toole, J., Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Operating Systems Review*, Vol. 29, No. 5, December 1995, pp. 251-266.

[19] Feldmeier, D., McAuley, and Smith, J., Protocol Boosters, January 1997.

[20] Goldberg, R.P., Architectural Principles for Virtual Computer Systems, Harvard University, Cambridge, MA, Ph. D . Thesis, 1972.

[21] Gosling, J., and McGilton, H., The Java Language Environment: A White Paper. Sun Microsystems, Mountain View, CA, 1995.

[22] Hartman, J., Manber, U., Peterson, L, and Proebsting, T., Liquid Software: A New Paradigm for Networked Systems. Department of Computer Science Technical Report, TR 96-11, University of Arizona, Tuscon, AZ, 1996.

[23] Harrison, M. and Ruzzo, W. and Ullman, J., Protection in Operating Systems, *Comm. A. C. M.*, Vol. 19, No. 8, 1976, pp. 461-471.

[24] Irvine, C. E., Schell, R. R., and Thompson, M. F., Using TNI Concepts for Near-Term Use of High Assurance Database Management Systems, in Proceedings of the 4th RADC Database Security Workshop, April 1991, Little Compton, RI.

[25] ISO/IEC, Information Technology - Open Systems Interconnection - Common Management Information Service Definition, 10164-2.

[26] Lipner, Steven B., Non-Discretionary Controls for Commercial Applications, *Proceedings 1982 Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, 1982, pp. 2-20.

[27] Montz, A. B., Mosberger, D., O'Malley, S. W., Peterson, L., Proebsting, A., and Hartman, J. H., Scout: A Communications-Oriented Operating System, Technical Report No. 94-20, Department of Computer Science, University of Arizona, June 1994.

[28] Myers, Philip, Subversion: The Neglected Aspect of Computer Security, Masters Thesis, Naval Postgraduate School, 1980.

[29] Necula, G. and Lee, P., Safe Kernel Extensions without Runtime Checking, *Proceedings Second Symposium on Operating Systems Design and Implementations*, , Seattle, WA, USENIX Assoc, October 1996, pp. 229-243.

[30] Organick, E. I., The Multics System: An Examination of its Structure, MIT Press, Cambridge, MA, 1972.

[31] Popek, G.J., Correctness in Access Control, Proceedings ACM National Conference, 1973, pp. 236-241,

[32] Popek, G., and Goldberg, R. Formal Requirements for Virtualizable 3rd Generation Architectures, 1974, *Comm. A. C. M.*, Vol. 17, No. 7, 1974, pp. 412—421.

[33] Rodeh, O., Birman, K., and Hayden, M., Dynamic Virtual Private Networks, Technical Report TR-1654, Computer Science Department, Cornell University, Ithaca, NY, 1997.

[34] Saltzer, J. H., and Schroeder, M. D., The Protection of Information in Computer Systems, *Proceedings of the IEEE*, Vol. 63, No. 9, 1975, pp. 1278-1308.

[35] Schaefer, M. and Schell, R. R., Toward an Understanding of Extensible Architectures for Evaluated Trusted Computer System Products, Proceedings 1984 Symposium on Security and Privacy, *IEEE Computer Society Press*, Oakland, 1984,pp. 41-49.

[36] Schell, R., Electronic Commerce Across Open Networks: Doomed to Fail?, Private communication, 1997.

[37] Schroeder, M., Cooperation of Mutually Suspicious Subsystesm in a Computer Utility, Ph.D. dissertation, M.I.T., Cambridge, MA, 1972.

[38] Schroeder, Michael D. and Saltzer, Jerome H., A Hardware Architecture for Implementing Protection Rings, *Comm. A.C.M.*, Vol. 15, No. 3, 1972, pp. 157-170.

[39] Shirley, L. J. and Schell, R. R., Mechanism Sufficiency Validation by Assignment, *Proceedings 1981 Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, 1981, pp. 26-32

[40] Shockley, William R. and Schell, Roger R., TCB Subsets for Incremental Evaluation, *Proceedings Third AIAA Conference on Computer Security*, December 1987, pp. 131-139

[41] Shockley, W. R., Protection of Distributed Applications*, Proceedings 16th National Computer Security Conference*, September, 1993, pp. 364-366.

[42] Smith, J. M., Farber, D. J., Gunter, C.A., Nettles, S.M., Segal, M. E., Sincoskie, W. D., Feldmeier, D. C., and Scott Alexander, D, Switchware: Towards a 21st Century Network Infrastructure, CIS Department, University of Pennsylvania, 1997.

[43] Tennenhouse, D. L., and Wetherall, D. J., Towards an Active Network Architecture, in *Multimedia Computing and Networking* '96, January 1996.

[44] Tock, T., Sturman, D., and Campbell, R., Security, Delegation, and Extensibility, Computer Science Department, University of Illinois, November 1994.

[45] Wahbe, Robert, Lucco, Steven, Anderson, Thomas E., and Graham, Susan L., Efficient Software-Based Fault Isolation, *Proc. Fourteenth ACM Symposium on Operating System Principles*, Ashville, NC, December, 1993, pp. 203-216.

[46] Wetherall, D., J., Guttag, J. V., and Tennenhouse, D. L, ANTS: A Toolkit for Building Dynamically Deploying Network Protocols, to appear in *IEEE OPENARCH'98*, San Francisco, CA, April 1998.

[47] Yemini, Y., and da Silva, S., Towards Programmable Networks, in *IFIP/IEEE International Workshop on Distributed systems: Operations and Management*, L'Aquila, Italy, 1996.