A Domain and Type Enforcement UNIX Prototype

Lee Badger, Daniel F. Sterne, David L. Sherman,
and Kenneth M. Walker
Trusted Information Systems, Inc.

ABSTRACT: UNIX system security today often relies on correct operation of numerous privileged subsystems and careful attention by expert system administrators. In the context of global and possibly hostile networks, these traditional UNIX weaknesses raise a legitimate question about whether UNIX systems are appropriate platforms for processing and safeguarding important information resources. Domain and Type Enforcement (DTE) is an access control technology for partitioning host operating systems such as UNIX into access control domains. Such partitioning has promise both to enforce organizational security policies that protect special classes of information and to generically strengthen operating systems against penetration attacks. This paper presents the primary DTE concepts, discusses their application to single hosts, IP networks, and NFS, and then describes the design and implementation of a DTE UNIX prototype system.

Portions reprinted, with permission, from the 1995 IEEE Symposium on Security and Privacy, May 1995, pages 66–77. ©1995 IEEE.

1. Introduction

As UNIX systems become a major part of the global information infrastructure, UNIX security mechanisms are coming under increasing pressure to resist attacks by highly motivated individuals, companies, and governments. Currently, UNIX security rests on protection bits, the root user, and the setuid/setgid mechanism. These mechanisms place a great deal of security responsibility on privileged application programs and expert system administration. This has two important consequences. The first is that UNIX systems often exhibit a "weakest link" phenomemon in which compromise of any privileged subsystem (e.g., fingerd, sendmail, rdist) makes an entire host vulnerable. The second is that reliance on numerous privileged applications increases the difficulty of implementing coordinated security policies that provide uniform protection to data and processing resources. These two problems motivate a legitimate concern over whether UNIX systems are appropriate platforms for processing and safeguarding important information resources in global and possibly hostile networks.

UNIX (and other operating systems) can in theory be hardened against threats inherent in such environments by adding an access control layer that restricts privileged processes so that damage resulting from compromise or error is limited. In addition, such access controls could enforce security policies that appropriately protect information and processing resources based on sensitivity, integrity requirements, etc. These benefits, however, have not been realized by mainstream UNIX systems even though a number of access control mechanisms [Bell 1976; Biba 1977; Boebert 1985; Clark 1987; Lipner 1982] have been available for years. One reason may be that security enhancements often impose significant costs resulting from more complex system administration, application incompatibility (or unavailability), and additional user training. This raises a central question for practical UNIX security: can significant enhancements be added in a way that is understandable, effective, and unobtrusive?

This paper presents our experiences with a new form of access control, Domain and Type Enforcement (DTE) [Badger 1995], and a prototype DTE UNIX system. In recognition of the fact that access control techniques have not been easily accepted by operating system vendors (or users), DTE has been formu-

lated specifically to address requirements of greatest concern for both vendors and users, namely: flexibility, simplicity, operating system interoperability, binary application compatibility, and performance.

In the following sections we present the fundamental DTE concepts, discuss DTE application to single hosts, IP networks and NFS, and then discuss design and implementation issues of the DTE UNIX kernel. Sections 2 and 3 present DTE. Sections 4–6 cover design and implementation issues for adding DTE to UNIX systems. In section 7 we discuss related work. We conclude with a discussion of the current status of DTE, unresolved issues, and plans for further development.

2. *Type Enforcement*

DTE is an enhanced form of type enforcement, a table-oriented access control mechanism originally proposed by Boebert and Kain [Boebert 1985] and later refined in the LOCK system [O'Brien 1991]. As with many access control schemes, type enforcement views a system as a collection of active entities (subjects) and a collection of passive entities (objects). For type enforcement, an invariant access control attribute called a *domain* is associated with each subject, and another invariant attribute called a *type* is associated with each information object. A global table, the Domain Definition Table (DDT), represents allowed interactions between domains and types. Each row of the DDT represents a domain, and each column represents a type. Subject-to-object access control decisions are based on table lookups: when a subject attempts to access an object, the domain of the subject selects a row of the DDT and the type of the object selects a column of the DDT. If the requested access mode (e.g., read, write) is not present in the selected cell, the access attempt is denied.

Subject-to-subject access control is based on a second table, the Domain Interaction Table (DIT), which relates domains to domains. When a subject A attempts to access another subject B, the domain of A selects a row of the DIT and the domain of B selects a column of the DIT. If the selected cell does not contain the requested access mode (e.g., signal, create, destroy), the access attempt is denied.

In its original formulations [Boebert 1985; O'Brien 1991], type enforcement was applied to the Secure Ada Target (later renamed LOCK) trusted computing base (TCB). While LOCK supports a UNIX emulation layer, LOCKix [O'Brien 1991], the *subjects* and *objects* in the original formulation are LOCK abstractions, not UNIX abstractions. As a consequence, LOCK type enforcement directly controls UNIX emulations but does not distinguish between individual UNIX processes running on an emulation. The benefit of this approach is the potential for high assurance since UNIX kernel mechanisms need not be trusted to

provide security. Our goals here are slightly different in that we wish to control individual UNIX processes at the level of assurance achievable with UNIX kernel mechanisms.

While various approaches are possible, a natural application of type enforcement to UNIX would remain consistent with the informal UNIX model for discretionary access control (DAC) based on uids, gids, and protection bits. In this approach, UNIX processes are the subjects and UNIX data containers (e.g., files, shared memory segments) are the objects. In general, system events (e.g., open, exec) that involve comparisons between a process's uid and an object's protection bits also involve a DDT lookup and access control check. Creations of processes with different domain attributes also follow the UNIX model; since a nonprivileged UNIX process may only change its process uid through the setuid mechanism, domain transitions are also bound to specified exec events. Finally, since UNIX processes may also access each other via signals, type enforcement for UNIX adds mediation of signal dispatch based on the DIT.

Unless rights to modify a running type enforcement access control configuration are distributed within a system, type enforcement is a *mandatory* mechanism in that the access control rules are centrally determined, processes cannot change object type attributes, and process domain attributes change only according to systemwide rules. Under this interpretation, type enforcement can express and enforce many fine-grained access restrictions and consequently can partition a UNIX system according to the principle of least privilege [Schroeder 1972], thus limiting damage resulting from compromise or error in privileged programs. This addresses a major weakness of traditional UNIX access controls, which are vulnerable to trojan horse programs that, when executed in the context of privileged users, can give away (or save for later use) access to sensitive resources. In combination with a facility for authenticating users for different domains, a type-enforcing UNIX can also support user roles [Landwehr 1984; Mayer 1989; Baldwin 1990; Thomsen 1990] and implement systemwide mandatory access control policies supporting organizational goals (e.g., proposal information is kept separated from public information). Although type enforcement was originally proposed [Boebert 1985] as an integrity mechanism, it is also capable of enforcing other policies, such as the DoD confidentiality policy [Bell 1976], that control the propagation of sensitive information.

Type enforcement is both flexible and strong, but our experience with a type-enforcing system [Sterne 1992] indicates that its practical application requires solutions to three significant problems:

1. *Type enforcement access control configurations may become too complex.* Type enforcement is most effective when used to express many different

access restrictions for many controlled programs. This introduces significant complexity into the type enforcement tables, which must express all allowed information flows and subject transitions. Additionally, assignments of domains to specific processes and assignments of types to individual files are relevant because these bindings contribute to determining the accesses permitted by the configuration. Such assignments are not expressed in the original type enforcement formulation. These assignments can be numerous; for example, our corporate file server typically runs over 300 processes and locally stores over 600,000 files. Such magnitudes significantly increase complexity.

2. *Type enforcement tabular structures do not naturally map to standard system structures.* Figure 1 shows the mismatch between type enforcement tables and system structures. A typical system is composed of two primary *hierarchies*, the object (i.e., file) hierarchy and the subject (i.e., process) hierarchy. There is no obvious correspondence between the type enforcement tables and typical system hierarchies even though the structures of process and file hierarchies are security-relevant; for example, directory hierarchies determine visibility and grouping of files, and process hierarchies reflect relationships between potentially security-relevant programs and influence propagation of process capabilities (e.g., file descriptors). This mismatch hinders application of type enforcement to actual systems.
3. *Most type enforcement policies need to be invented from scratch.* The DoD Mandatory Access Control (MAC) policy for protecting sensitive information and, to an extent, DAC reflect existing organizational security policies (e.g., the DoD Information Security Program Regulation [DoD 1986]). For MAC, files typically have the security labels that correspond roughly to document classification markings, and processes are labeled with the clearances of their users. For DAC, files are owned by users, and processes have user identities and access files on behalf of users. For type enforcement, however, no such tradition exists, and domain and type attributes in the type enforcement tables must (currently) be custom-engineered for each security-relevant application.

3. Domain and Type Enforcement (DTE)

Domain and Type Enforcement (DTE) is an enhanced version of type enforcement designed to provide the benefits of type enforcement at reduced cost and complexity. Three primary techniques distinguish DTE from simple type enforcement:

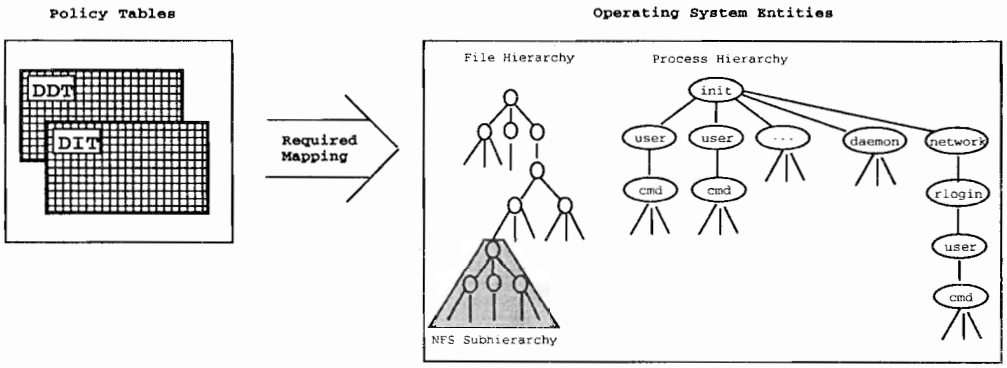

Figure 1. Mismatch Between Policy Concepts and System Structures.

1. DTE policies are specified in a high-level language suitable for expressing reusable access control configurations. A DTE specification includes security attribute associations such as type/file associations as well as other access control information. Inclusion of type associations allows a DTE policy to be comprehensive: all the access control information is gathered in one place. The language provides a high-level view of information traditionally enumerated in type enforcement tables and includes facilities for superimposing security attribute bindings and domain transitions on applications that are not aware of DTE.

2. During system execution, DTE file security attributes are maintained "implicitly" in a form that capitalizes on intrinsic object hierarchies (e.g., directories of files) to concisely represent security attributes. When significant portions of system file name spaces are homogeneously typed, this strategy simplifies security configuration and removes the need to physically store a type label with every file. This permits DTE policies to be easily applied to existing media with full backward compatibility with existing disk and file system formats.

3. DTE provides direct support for interoperating with hosts that are not aware of DTE. The primary technique for this is a language-driven mechanism that associates a DTE domain with each non-DTE host. DTE systems view processes running on a non-DTE system as running in the domain associated with the host. This allows DTE systems to mediate communications with non-DTE hosts according to local DTE policies.

3.1. DTE Language Support

DTE Language (DTEL) is a simple high-level symbolic language for expressing reusable DTE configurations in a human—rather than machine—friendly form. The general scheme of DTEL is to express information traditionally held in DDT and DIT tables with as much simplifying structure as possible. We anticipate that some systems will require attributes that are closely related; DTEL therefore supports such inherent (and simplifying) structure by providing macro facilities that allow security attributes to be defined using shared components. To document and clarify specifications, DTEL supports standard `C` commenting conventions. Currently, DTEL provides four primary statements for expressing a DTE configuration for a single host: the `type` statement, the `domain` statement, the `initial_domain` statement, and the `assign` statement.[1] The purpose of this section is not to fully document DTEL but to demonstrate through a small example that a meaningful DTEL policy can be expressed completely in a form simple and concise enough to be administered at reasonable cost. Our metric for "reasonable cost" is that policy administration should be no more difficult than routine UNIX administration tasks such as configuring remote file systems or adding user accounts. To validate that our example policy is not trivial, we have run it on our prototype DTE system and found it to provide useful protection. We now introduce the primary DTEL statements in the context of a commercial policy designed to provide protection and separation for enterprise data types and user authorizations in an engineering organization.

A DTEL `type` statement declares one or more types to be part of a DTE configuration; other DTEL statements may refer only to types declared with the `type` statement. Each object is associated with exactly one of the types provided by the type statements. For example, the following `type` statement declares one type for ordinary UNIX files, programs, etc., and three types describing enterprise data:

```
type    unix_t,     /* normal UNIX files */
        specs_t,    /* engineering specs */
        budget_t,   /* budget projections */
        rates_t;    /* labor rates */
```

1. We have also formulated but not completely implemented DTEL facilities to control mount operations, but we restrict our attention here to implemented features with which we have actual experience. The `mount` statement would add several lines to the example presented in this section.

A DTEL `domain` statement consists of four components:

entry points Programs, identified by path name, that are bound to the domain and must be invoked in order to enter the domain. This mechanism allows access rights of a domain to be explicitly bound to programs that are believed to employ them appropriately.

object access rights Permissible modes of access, when executing in the domain, to objects of specified types, e.g., the normal UNIX modes `rwx`. UNIX overloads the `x` mode for directory traversal; DTEL distinguishes between execute and traverse access using a new mode, `d`, that applies only to directories.

default output type Optionally, a domain may designate that new objects are created with a specified type, to which the domain has write access, when the creating events (e.g., file opens) do not provide explicit type parameters. This mechanism allows existing programs running in domains that have multiple types writeable to create new objects without ambiguity concerning the types of the objects.

subject access rights Permissible modes of access, when executing in the domain, to subjects in other specified domains. In addition to providing individual subject access rights for UNIX signals (sigkill, sigpause, etc.), DTEL provides two access rights for creating new subjects. If a domain A has *exec* access rights to another domain B, a subject in A may create a subject in B by executing one of B's entry-point programs and requesting that the program run in B. To assign custom-tailored domains to existing programs that are not aware of DTE, it is important to be able to cause domain transitions at points where one program starts another, for example, where a system process starts a network daemon. In UNIX, all such program loads are performed by the `exec()` family of system calls. If a domain A has `auto` access rights to another domain B, a subject in A automatically creates a subject in B when it does a normal `exec()` system call of an entry point program of B. This mechanism allows domain transitions to be superimposed on many existing system daemons and subsystems without modifying their executables.

For example, the following defines three domains for regulating access to engineering specifications, budget projections, and labor rates, which are labeled using the types declared above:

```
#define DEFAULT (/bin/sh), (/bin/csh), (rxd->unix_t)
```

```
domain engineer_d     = DEFAULT, (rwd->specs_t);

domain project_d      = DEFAULT, (rwd->budget_t), (rd->rates_t);

domain accounting_d   = DEFAULT, (rd->budget_t), (rwd->rates_t);
```

Each domain is a list of tuples where each tuple contains either a UNIX path or a collection of access modes to (designated by ->) a collection of type or domain names. The `domain` statement collapses the DDT and DIT into one representation. These sample domains regulate observation and modification of typed data to support three kinds of user authorizations for an engineering organization: (1) engineers manipulate engineering specifications only, (2) project leaders observe labor rates and create budget projections, and (3) accountants observe budget projections and set labor rates. Although these user-oriented domains have command shells for entry point programs, it is also possible to further refine these domains by installing other programs as the entry points, for example, spreadsheet or database programs, that are specific to user responsibilities.

In addition to user-oriented domains, a complete DTEL specification must associate domains with all operating system processes and must also provide a mechanism for user-oriented domains to be initiated. To accomplish this, it is necessary for a DTEL policy to reflect the structure of the UNIX process hierarchy because domains are process attributes inherited by default from process to process. Typically, after kernel initialization, a UNIX system starts an initial process that runs the `/etc/init` program. This process is then responsible for creating all other UNIX processes, including, indirectly, the login process that starts user sessions. DTEL specifies the domains of all processes by setting the domain of the first process and then controlling domain-changing operations using the *exec* and *auto* domain access modes. The DTEL `initial_domain` statement specifies the domain of the first process. Child processes inherit the domains of their parents and optionally transition to other domains during `exec()` operations constrained by their *exec* and *auto* access rights.

For example, the following defines two system-oriented domains supporting the user-oriented domains given above by providing a mechanism to initiate them and by running the rest of the system in domains that have no access to the user-oriented data types:[2]

2. Except indirectly through device special files. In a less simple example, programs, such as fsck, that need such access, would run in special domains that grant access, and the rest of the system programs would be controlled.

```
domain system_d = (/etc/init), (rwxd->unix_t), (auto->login_d);

domain login_d  = (/bin/login), (rwxd->unix_t),
                  (exec->engineer_d,project_d, accounting_d);

initial_domain = system_d;
```

The `initial_domain` statement causes the initial process to run in the `system_d` domain, which has access only to `unix_t` data. This domain is inherited by all system processes except for the login process. When a process in `system_d` runs the login program, the *auto* access mode from `system_d` to `login_d` causes the login program to run in the `login_d` domain, which has the ability to create processes in the three user-oriented domains. To minimize privilege, only the login program can initiate the user-oriented domains. In this scenario, the login program is DTE-aware and properly authenticates and checks the authorization of each user before starting a process in the user's domain.

The fourth DTEL statement is the `assign` statement, which is used to associate exactly one type with each file on a system. Assign statements support "implicit typing," a technique for associating types with files based on directory hierarchies by stating general rules and then listing exceptions. Figure 2 displays the concept. In that figure, all files below the root directory, by default, have the type `unix_t`. In three subdirectories, however, `unix_t` is "overridden" by the `specs_t`, `budget_t`, and `rates_t` types. In each subdirectory, all files by default have the type of the subdirectory. Using this technique, it is easy to associate a small number of types with a large number of files as long as type associations tend to group according to existing directory hierarchies. In our experience, directory hierarchies tend to organize files by purpose, origin, sensitivity, etc., in short, the same criteria by which type labels would often be assigned. Although types may naturally reflect directory hierarchies, there are clearly exceptions to this rule, and assign statements can also express exceptions for individual files as overrides to the default type associations.

An assign statement associates a type with a path P and is optionally recursive; recursive statements (indicated by `-r`) apply to all paths having P as a prefix. For statements having paths such that one is a prefix of another, the statement having the longest path P overrides statements having shorter paths for all files reached through P. DTEL type associations are tranquil in that the type of an object does not change over the object's lifetime. As a consequence, maintenance of attribute associations at runtime may force (automatic) rebindings of attributes to

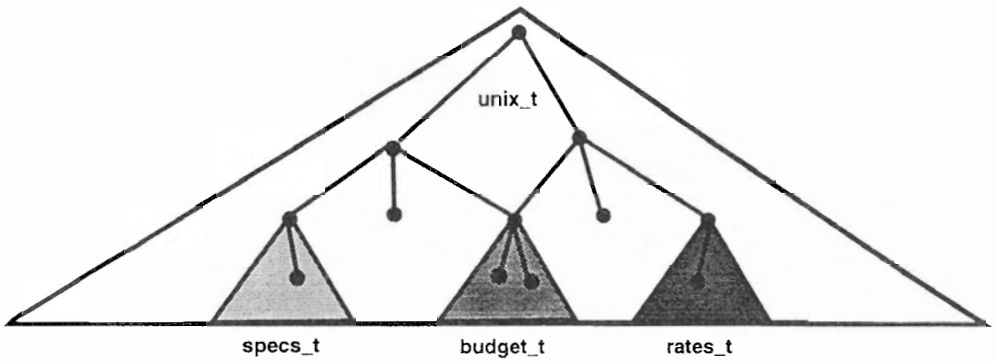

Figure 2. Implicit Types.

hierarchical structures. For example, when a file is renamed, its assign statement, if any, is changed to reflect the file's new location. Constraints can be placed on type assignments. DTEL provides a feature to force type assignments to be static (indicated by `-s`) at runtime, which locks specification-time type assignments for hierarchical portions of the object name space and denies any attempt at runtime to create objects of other types in those areas.

One consequence of binding attributes by location is that files that can be reached through multiple (hard link) paths may appear to have multiple types. Symbolic links are not an issue because they merely name hard link paths represented by DTEL assign statements. To prevent multiple hardlinks from introducing ambiguity, DTEL will employ a tool at specification time that discovers whether multiple assign statements name the same file. For each such file, the tool will prompt the security administrator to decide which among the possible types the file should have and will then add additional assign statements to ensure that all assign statements for the file give the same type. Once initialized, a DTE system maintains type bindings unambiguously even in the presence of multiple links.

For example, the following assign statements provide areas for the domains and types displayed in figure 2:

```
assign -r -s unix_t     /; /* default type */
assign -r -s specs_t    /projects/specs;
assign -r -s budget_t   /projects/budget;
assign -r -s rates_t    /projects/rates;
```

In order to allow UNIX system processes to continue to function, all system processes except login run in a domain that gives access to the standard UNIX objects accessible from the root directory (`/`) that has type `unix_t`; this assures

compatibility for basic system functions. The DTEL processor requires that / is given a type using an `assign` statement. User processes run in one of the three user-oriented domains having appropriate access to the three subdirectories for `specs_t`, `budget_t`, and `rates_t`.

The four basic DTEL statements are sufficient to express complete access control policies for processes, files, and most volatile system abstractions such as shared memory, semaphores, and message queues. Figure 3 shows the completed sample policy, which provides three user-oriented domains and all mechanisms required to support them on a typical DTE UNIX system. This sample policy can be incrementally refined to add additional user domains, distinguish between console and network user sessions, simultaneously support additional organizational policies, and harden UNIX itself by running UNIX system components in more tightly constrained domains using the *auto* access mode. Through such extensions, DTE policies can be configured to fit individual site requirements. Because UNIX system process interactions are relatively standard, however, we believe that such policies can also be standardized and portable (or configurable via macros) between UNIX systems.

3.2. Runtime Implicit Attribute Management

Although DTEL specifies type associations implicitly, DTEL does not mandate how the attributes are actually maintained at runtime. Traditional trusted systems (such as Multics[Organick 1975, National Computer Security Center 1986], Trusted XENIX [National Computer Security Center 1991], TMach [Branstad 1989], etc.) store one MAC label or DAC ACL for each file, usually with the file's on-disk representation. This is an option for DTE systems also, but we rejected it for two reasons:

- The type labels would be distributed across all object media and therefore could not be easily reviewed or analyzed without physically scanning all objects.
- More importantly, keeping type labels explicitly with files would require changes to low-level file and file system formats, causing DTE systems to be incompatible with existing systems from which they import network file systems.

```
/*
 *       DTEL Commercial Policy.
 */

type     unix_t,          /* normal UNIX files, programs, etc. */
         specs_t,         /* engineering specifications */
         budget_t,        /* budget projections */
         rates_t;         /* labor rates */

#define  DEFAULT          (/bin/sh), (/bin/csh), (rxd->unix_t)  /* macro */

domain   engineer_d       = DEFAULT, (rwd->specs_t);
domain   project_d        = DEFAULT, (rwd->budget_t), (rd->rates_t);
domain   accounting_d     = DEFAULT, (rd->budget_t), (rwd->rates_t);
domain   system_d         = (/etc/init), (rwxd->unix_t), (auto->login_d);
domain   login_d          = (/bin/login), (rwxd->unix_t), (exec-> engineer_d,
                                                                  project_d,
                                                                  accounting_d);

initial_domain system_d;        /* system starts in this domain */

assign   -r -s            unix_t              /;      /* default for all files */
assign   -r -s            specs_t             /projects/specs;
assign   -r -s            budget_t            /projects/budget;
assign   -r -s            rates_t             /projects/rates;
```

Figure 3. Example DTEL Policy.

Instead, our approach maintains type associations in a UNIX kernel-resident runtime policy database that is established at system boot time.

Each UNIX file object access starts with a resolution mechanism that converts a pathname or an NFS file handle to an internal object handle that is then used for subsequent object manipulations (read, write, etc.). During this resolution mechanism, a DTE UNIX kernel consults the runtime policy database to determine file type attributes that are then used for DTE mediation. The runtime policy database is tightly integrated into the name resolution mechanism and ensures that all file objects have type attributes. Because the attributes are maintained implicitly instead of being enumerated exhaustively, most configurations can be easily held in kernel buffers; storage for the runtime policy database therefore typically requires no additional I/O and imposes a negligible performance overhead for security attribute maintenance.

Two primary classes of system calls may cause changes to the runtime policy database: file creations and file rename events. A file creation updates the runtime

policy database if the type of the new file (constrained by the creating process's domain) is different from the type inherited from the new file's location. Such a file creation updates the runtime policy database by adding a new `assign` statement to represent the exception. The frequency with which file creations force policy database updates is dependent on the security configuration; for some policies (like the example in section 3.1), there are no runtime policy database updates because the file hierarchy is locked down using the `-s` assign directive. If many exceptions designate the same type, they may be coalesced into a single recursive `assign` statement, thus preserving the compactness of the runtime policy database.

A rename event can be modeled as a link operation establishing the new path name to an object followed by an unlink operation removing the original path name. To maintain tranquility of type bindings, the runtime policy database intercepts rename events and adds an `assign` statement if necessary for the new path name to preserve the file's existing type. As with file creations, rename events only need to update the runtime policy if the type associated with the new path name is different from the type associated with the old path name.

4. *DTE Networking*

Since UNIX systems are usually networked, DTE systems must work naturally while communicating both with other DTE systems and with non-DTE systems. In particular, multiple DTE systems must provide mechanisms allowing coordinated protection of information among themselves, and DTE systems must protect themselves from non-DTE systems. In addition to requiring homogeneous or compatible DTE policies, coordinated protection requires that policy information regarding types of communicated objects and domains of communicating processes be transmitted reliably through the network subsystem. To accomplish this, DTE adds two attributes to network communications carrying user data: 1) the type of the data written by the sending process and 2) the domain of the process that sent the data, the *source domain*. A receiving process can view the data's type, which the receiver must know to adequately protect the data, or possibly to protect itself from the data. Additionally, a receiver can view the sender's domain; a DTE server that receives a request can therefore use the client's domain to decide whether to perform the requested function [Sherman 1995].

To maintain compatibility with existing network protocols and applications,

DTE attributes are carried as IP options,[3] with no change to packet contents. DTE mediates communications over standard datagram and stream-oriented services. In each case, DTE imposes access control mediation both at send time and receive time: to successfully send data of type `t`, a process's domain must permit write access to `t`, and to successfully receive data of type `t`, a process's domain must permit read access to `t`. For datagram protocols such as UDP, a single type labels the contents of an entire packet. For stream protocols such as TCP, different portions of a stream may have different types of data; a sequence of contiguous bytes having the same type is a *substream.*

These design choices give a high priority to compatibility and interoperability. Our datagram approach is not unusual, and homogeneously typed datagrams work well for existing applications since they are unaware of DTE and therefore only generate one type of data. Our stream approach, however, is less typical. A simpler approach would bind a security attribute to a stream socket and therefore to all data communicated on it. Typical UNIX service interactions, however, make this approach problematic. An important example is `inetd`, which receives socket connections for services it spawns: `inetd` must be able to connect to a socket and then hand the descriptor to a child process that may run in a different domain. The use of substreams removes the need for `inetd` to run in an all-powerful domain. Programs like telnet and rlogin provide other examples: if a user runs a program that produces output of multiple types, a single connection can carry the output back to the client in multiple substreams, but statically typed connections would force dynamic creation of new TCP connections to send the data. While multiple connections could be used to transmit multiple types of data, this would change application-layer protocols (like `rcmd`) and prevent DTE network applications from interoperating with their non-DTE peers.

In order for a DTE system to properly control network applications, all communications must carry type and source domain attributes. At the same time, however, DTE applications must interoperate with applications running on non-DTE systems that do not provide DTE attributes. To provide interoperability without weakening DTE, DTE hosts associate a domain with every foreign non-DTE host and mediate all network traffic with that host so that the effect of the mediation is as though the host were actually running DTE and the process sending (or receiving) from that host were running in the associated domain. The DTEL `inet_assign` statement can associate a single domain with the "universe" of

3. For experimental purposes, we currently assume that network packets are not stolen or modified. We plan to take advantage of known and emerging cryptographic techniques and protocols for communications authentication [Kohl 1993], integrity, and confidentiality [Ioannidis 1994, National Bureau of Standards 1977] as appropriate.

foreign non-DTE hosts, associate a different domain with each class A, B, or C network, and finally associate specific domains with individual non-DTE hosts that, for various reasons (such as quality of administration), are more or less trustworthy than other hosts on their LAN.

For example, the following `inet_assign` statements associate the `foreign_d` domain with all hosts not on a specific class C network. The domain `tis_d` is associated with all hosts on a specific network, and the domain `dte_dev_d` is associated with an individual host used for DTE development. Similar to the approach taken with assign statements, `inet_assign` statements for individual hosts override `inet_assign` statements for networks, which override the required `inet_assign` statement for the "universe."

```
inet_assign foreign_d     0.0.0.0;          /* unknown hosts */
inet_assign tis_d         10.11.12.0;       /* class C LAN */
inet_assign dte_dev_d     10.11.12.13;      /* individual host */
```

This technique has performed well in our corporate LAN, allowing us to appropriately "trust" specified non-DTE hosts. Although we are using source-address "authentication" for compatibility at present, our plans include moving to stronger authentication, such as is envisioned for IP6, as the overall network infrastructure evolves.

Although our experience with DTE networking is still somewhat limited, we have been able to run existing UNIX applications such as rsh, rlogin, telnet, ping, sup, and mount in suitable DTE domains and we have encountered no "show stoppers." We have discovered, however, that although TCP/IP hosts should drop IP options they don't recognize, that doesn't always happen and in particular, SunOS 4.1.1 on Sun 3 systems crashes when presented with an unrecognized option. As a result, we have added features to our systems that prevent the sending of DTE attributes to hosts that are not known to run DTE. We are now formulating the requirements of a DTE protocol that would maintain timely information on the DTE status of a machine as well as provide DTE policy coordination functions that ensure that different machines "mean" the same thing by DTE attributes they exchange.

5. DTE NFS

The ubiquitous use of NFS highlights the need for DTE to both support NFS on DTE systems and also to interoperate with non-DTE systems that use NFS. An

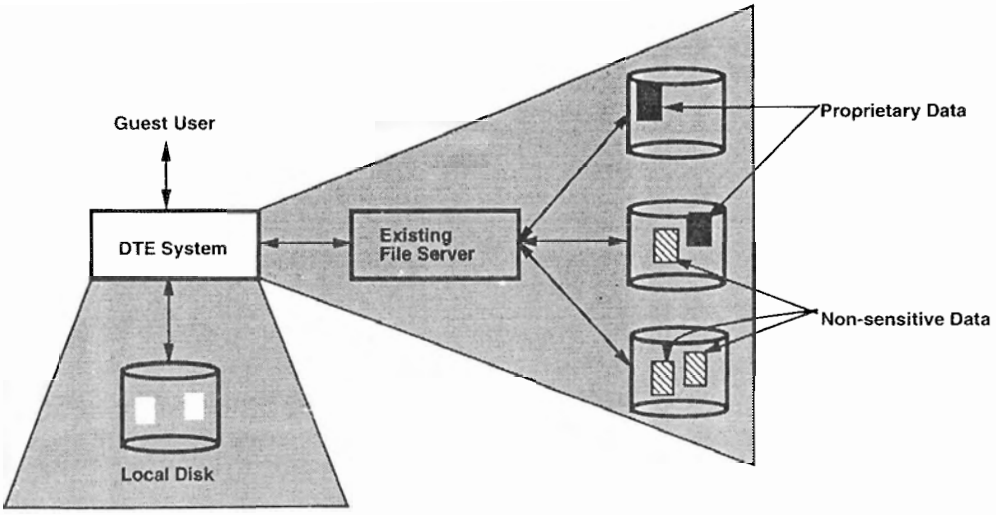

Figure 4. DTE NFS Clients.

integration of DTE and NFS for DTE-aware clients and servers is relatively simple and involves sending and receiving DTE attributes between DTE systems that then use the attributes for mediation in the same way they use locally stored DTE attributes. To make DTE useful in the short term, however, interoperability with non-DTE NFS clients and non-DTE NFS servers may be even more important.

A significant benefit of implicit typing in this regard is that DTE client workstations locally associate types with all files, even files provided over NFS by file servers that are not DTE-aware. This ability has allowed us to use DTE workstations to make selected portions of our corporate file server available to selected groups of users with a minimum of administrative effort. As electronic commerce increases the need for cooperation between organizations, we expect this scenario to become more common. Figure 4 displays the concept. A guest user has an account only on a DTE system. This system mounts from an existing file server and applies the type "proprietary data" to some files on the imported file system and the type "nonsensitive data" to the others. All guest user processes running on the DTE system are restricted according to the local DTE policy to access only the nonsensitive data.

The DTEL `inet_assign` statement allows a DTE system to refuse communication with selected non-DTE hosts and to prevent important types of data from being exported to non-DTE hosts (regardless of which communication service is used). If communication with a non-DTE NFS server is allowed, the client-side DTE/NFS subsystem associates types with imported files based on their path-

names. A premise of our work is that access controls must be flexible: it is up to the system administrator of a DTE system to determine whether a non-DTE host should be trusted to properly maintain data of various types. Although all the data received at the IP layer will be typed according to the DTE domain associated with the non-DTE file server, the DTE/NFS subsystem on the client system resides in the DTE UNIX kernel and is trusted to override the default communications type with correct file types as specified in the system's DTEL specification.

Initially, we added DTE only to the NFS client side, as described above. We are currently testing a DTE/NFS server that can serve clients on both DTE and non-DTE systems. When the client is on a DTE system, NFS requests are labeled by the client system with the source domain of the requesting process or, under some circumstances, a reserved domain that represents the client system's NFS subsystem. When a reserved NFS subsystem domain is used, remote file access is restricted to files accessible by the NFS subsystem domain, and the client DTE system further mediates the requesting process's access to locally cached representations of files based on the requesting process's domain and the types of the files. The DTE/NFS server uses the source domain as a client credential to consult the system's DTEL specification and determine whether the request is authorized at the server. In addition, each IP packet that carries the contents of a file accessed via DTE/NFS is labeled with the type associated with that file. A potential benefit of this approach is that both source domain and type attributes are readily visible to routers and network firewalls and could allow future versions of such devices to consult them when making filtering and routing decisions. An additional benefit is that the NFS protocol need not be modified. Although NFS client requests sent by non-DTE systems lack source domain attributes, the DTE/NFS server's IP subsystem attaches them (in accordance with the DTE system's DTEL specification) before passing the requests to the DTE/NFS subsystem for mediation. From the non-DTE client's point of view, the DTE/NFS server behaves like a non-DTE server, except that access may be denied for some requests where, in the absence of DTE, the request would have been granted.

The NFS protocol is designed so that NFS server systems may crash, reboot, and resume NFS service without requiring clients to perform new lookup operations on files that were open at the time of the crash. Each NFS request contains an NFS file handle that identifies the file by file number, which allows a typical UNIX system to access the file directly without performing a pathname translation. Unlike the permission bits and owner identifiers associated with a file, however, the implicit DTE attributes are not stored within inodes but in a separate attribute database organized by pathname instead of file number. If a newly rebooted DTE/NFS file server could not locate security attribute information for an NFS request, it would have to refuse the request, resulting in a stale file

handle at the client application. To prevent this, the DTE/NFS prototype reconstructs pathnames based on inode numbers by maintaining a cache of parent inode numbers for nondirectory files accessed via NFS, thereby permitting it to find file attributes in the DTE attribute database.

On our DTE/NFS prototype, the NFS daemon, like all other processes, runs in its own domain and is constrained in accordance with the system's DTEL specification. On most systems, this domain will likely be configured to give the daemon the ability to access and export many types of information. Nevertheless, it is not necessary to make *all* types accessible to it. If highly sensitive or critical types of information are stored on a system, it may be desirable to prevent them from being exported. Standard NFS provides features for limiting the exporting of files, but these features are coarse-grained, dealing only with whole file systems. By making certain types of files inaccessible to the NFS daemon, DTE provides a strong additional mechanism that can be employed by administrators to prevent individual files on arbitrary file systems from being exported.

Our experience with DTE/NFS servers is still very limited; however, our initial results are encouraging: NFS clients on DTE or non-DTE systems can be granted fine-grained restricted access to NFS-exported file hierarchies without change to applications or to non-DTE system configurations. The DTE prototype system's security attribute management strategy requires implementation of a new system cache and secondary storage to store the cache across system reboots. The cache, however, requires little human administration and only a small amount of additional I/O that only occurs in the context of I/O already required by NFS.

6. DTE UNIX Prototype

To gain experience with DTE concepts, we have implemented a prototype DTE UNIX system based on OSF/1 MK4.0. Although our system is based on a Mach microkernel, the DTE features are located in relatively high layers of the UNIX server's architecture, require no knowledge of microkernel interfaces, and are therefore reasonably portable to kernelized UNIX systems. We have also recently ported the DTE prototype to run on TMach Version 0.2 [Branstad 1989], a high-assurance trusted computing base designed to satisfy DoD security requirements as specified in the Trusted Computer System Evaluation Criteria [National Computer Security Center 1985]. Even though TMach employs a TMach-specific file system format, the integration required almost no change to the DTE implementation because the integration points between the UNIX server and TMach are generally at low layers in the UNIX architecture, whereas DTE is mostly implemented in the upper layers of the UNIX "kernel."

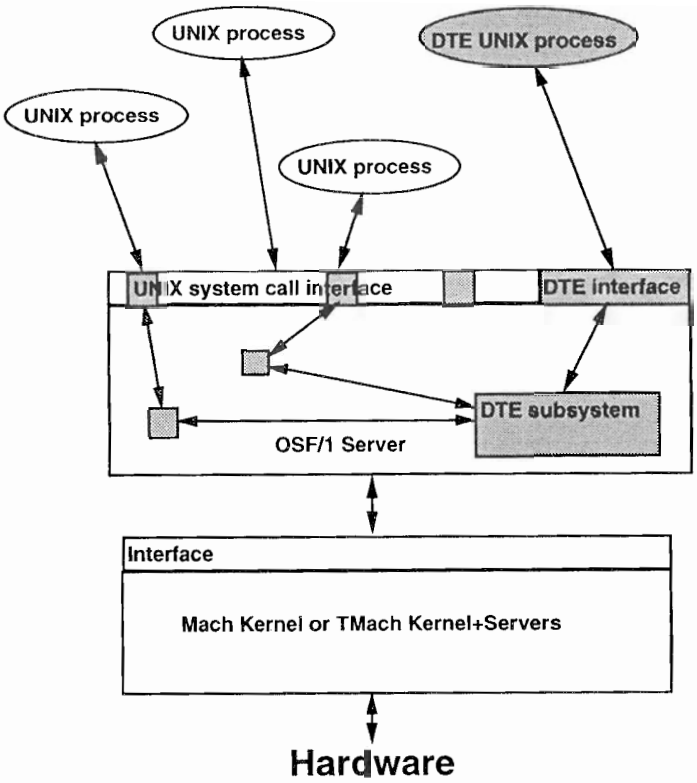

Figure 5. DTE System Architecture.

Figure 5 shows the prototype's architecture. To enhance portability, the majority of the DTE implementation is located in an isolated subsystem consisting of 7,300 lines of commented C code and 3,600 lines of commented lex and yacc code. Other UNIX kernel subsystems call into the DTE subsystem to request security services. This part of the integration consists of another 7,200 lines of code, bringing the total DTE integration to approximately 17,000 lines of kernel-resident code. The DTE prototype's kernel provides 20 new system calls for DTE-aware applications to use for retrieving security attributes for display to the user and for implementing security-relevant functions.

In addition to kernel changes, we have implemented a DTE version of the login program that authenticates users for specific roles [Landwehr 1984; Baldwin 1990; Thomsen 1990] and then confines user sessions to specific domains using domain transitions authorized by the DTEL specification. To allow users to view DTE attributes for processes and files, we have implemented DTE-aware versions of ls and ps, and we have also added DTE flags and options to a number of UNIX utilities (mkdir, ln, cron, at, cp, rcp, rsh) that allow users to create ob-

jects of specific types or to request jobs to run in specific domains. Additionally, we have implemented a DTE-aware version of GNU Emacs 19.22 that displays type attributes of file buffers and allows users to simultaneously view and manipulate labeled information in multiple windows.

As the prototype boots, it reads its DTEL specification and confines all processes, regardless of UNIX root privileges, to specified domains. DTE is active before single-user mode has been reached. According to its DTEL specification, the prototype labels files, network packets, and processes; determines domain interactions; and mediates process access requests. We have tested a number of policies using the prototype, including a policy to partition the components of a simulated command and control system, a policy to strengthen UNIX by confining UNIX root processes in 27 separate domains, and an enterprise data protection policy (similar to that of figure 3). Additionally, we use DTE client workstations to permit but safely limit access by "guest" users who are authorized to see some but not all TIS sensitive data.

In addition to providing enhanced security through additional mediation of UNIX abstractions, the DTE prototype's design and implementation have given a high priority to maintaining operating system interoperability and binary application compatibility. Three aspects of the DTE prototype are central to achieving these goals: (1) preserving existing data formats by employing implicit security attributes, (2) ensuring that implicit attributes are recoverable in the presence of system shutdowns and power failures, and (3) adding DTE networking support without change to existing protocols.

6.1. Mediation

The *subjects* of the DTE prototype are processes. The *objects* are files of any UNIX type (normal, directory, symbolic link, character and block special, FIFO, UNIX socket), IP datagrams, TCP substreams, and IPC mechanisms such as shared memory segments, semaphores, and message queues.[4] In the case of TCP substreams, a single substream may be implemented by a sequence of IP datagrams; in this case, the system ensures that the DTE security attributes of the substream and its constituent objects are identical.

For each access decision, the prototype looks up the domain of the requesting subject in kernel memory and searches an implicit attribute database for the requested access mode to the type or domain of the object or subject to be accessed.

4. The DTE prototype does not currently mediate shared memory segments, semaphores, or message queues, but mediation of these abstractions would be straightforward.

If a default output type is not specified for a domain using DTEL, the DTE prototype requires operations creating new objects to specify the types of the new objects explicitly using DTE system calls and denies requests that do not; this keeps type associations unambiguous. The prototype's mediation falls in seven primary categories:

file mediation Most access decisions are driven by open requests; after an initial open, no further access decisions are required for individual read or write operations. The DTE prototype supports the traditional `rwx` UNIX access modes except that it does not overload the `x` bit for directory traversal. Instead, it provides a new mode, `d`, that grants directory traversal. This allows `rd` access to be easily given to hierarchies of files without granting execute permission. The prototype prevents modification or relocation of entry point programs; this maintains validity of entry point paths described in DTEL domain statements. Additionally, the DTE prototype implements the `-s` strict DTEL option and denies any attempt to create an object of a given type within a hierarchy strictly bound to another type.

message mediation Each message or substream sent is an object creation; the default output type of the sending domain, or an explicitly provided type, determines the type of the message or substream. A process is prohibited from creating messages of types for which the process does not have `w` access. At receive time, the domain of the receiving process must have `r` access to the type of the message or substream; access denial at receive time results in discarded messages or inaccessible substream data that must be flushed or handed off to another domain that can consume it. Under some conditions, error messages are returned to the sender.

process mediation The DTE prototype mediates all signals as separate access rights. Additionally, the prototype implements the "exec" and "auto" DTEL access rights, which authorize subjects in specified domains to create subjects in other specified domains, and prevents domain changes except through execution of entry point programs.

descriptor revalidation During a domain transition, open file descriptors passed through the exec operation are remediated in the context of the new domain.

mount mediation The prototype does not currently mediate mount events. When implemented, the mount mediation will restrict which device special files can be mounted at which portions of the hierarchical file space.

device mediation The prototype currently mediates devices based on the types of device special files. This strategy will be expanded to cover creation of new device special files by adding mediation to the mknod system call that forces all device special files for a device to be of the same type and that restricts mknod system calls based on domain access rights to the types of devices.

special system calls A number of UNIX kernel extensions allow user processes to set kernel paging areas or load new modules into running kernels. The DTE prototype does not currently mediate these system calls. The DTE domain transition mechanism provides a basis, however, for isolation of special access rights for performing these operations in particular domains that minimize the risk of abuse.

6.2. Implicit Attributes

For entities that must be recreated at each system boot (such as process structures or IP datagrams), the DTE prototype attaches security attributes explicitly to each object. Compatibility and performance can be maintained with this strategy because modifications need not affect secondary memory data formats or require additional I/O.

Files, however, present a more difficult case both because security attributes must be maintained on disk to survive system reboots and because files are usually numerous. To address these issues, the prototype associates security attributes with files "implicitly" based on their locations within directory hierarchies. For portability, most of the prototype's functions for file security attributes are implemented at the Virtual File System (VFS) layer and build associations between vnodes [McKusick 1995] and security attributes. Since all currently accessed files are represented by vnodes, all files in use have associated security attributes. When the prototype boots, it creates in kernel memory a tree of *map nodes* that describe how security attributes are bound to the hierarchical file name space. Although our current prototype simply keeps this tree entirely in memory, it can in principle be paged to disk as necessary.

A sequence of map nodes proceeding from the root map node to a leaf map node names an existing path in the hierarchical filesystem name space. Each map node optionally associates one or more security attributes with the path component associated with it. The prototype currently maintains two kinds of security attributes bound to files: type names and domain entry points. To represent attributes implicitly, a map node may also associate security attributes with files

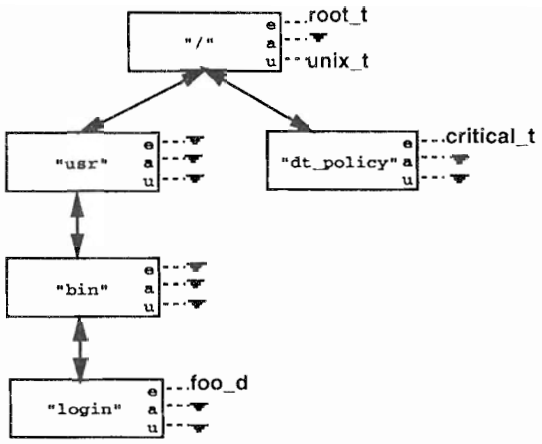

Figure 6. Map Nodes.

whose pathnames merely include the map node as a prefix. Such map nodes represent "implicit" associations. For each security attribute, a map node provides the following options:

implicit at The attribute is bound to this path component. In the absence of higher-priority map nodes that conflict with this map node, the attribute is also bound to all pathnames having this path component as a prefix.

implicit under The attribute is not bound to this path component, but, in the absence of conflicting higher priority map nodes, the attribute is bound to all pathnames having this path component as a prefix.

explicit The attribute is bound to this pathname only.

Informally, the prototype resolves map node conflicts by giving priority to the map node that represents a longer path, interpreting *implicit under* attributes to be "longer" than *implicit at* attributes for the same path and always giving priority to *explicit* attributes.

Each path referenced in a domain or assign statement potentially generates a map node for every component of the path. For example, a path `/a/b/c` given in a DTEL statement generates three map nodes (the root map node is automatically present). Map nodes are shared, however, so if a second DTEL statement specifies `/a/b/c/d`, only one new map node is generated. DTEL provides flags to set the initial options of map nodes: the DTEL assign statement, which associates types with files, takes an `-r` option to designate *implicit at* and a `-u` option to designate *implicit under*. DTEL domain statements automatically generate explicit associations for their entry point attributes. For example, the following DTEL statements generate the map nodes displayed in figure 6.

```
assign root_t          /;
assign -u unix_t       /;
assign critical_t      /dt_policy;
domain foo_d = (/usr/bin/login), ...;
```

That figure shows five map nodes, one for each unique component in the paths `/usr/bin/login` and `/dt_policy`. Each map node records the name of its path component and optionally records attribute associations (in figure 6, `e` for *explicit*, `a` for *implicit at*, and `u` for *implicit under*). Figure 6 shows that the root map node is explicitly of type `root_t` and that all files under the root "inherit" the type `unix_t`. This inherited type is overridden, however, for the file `/dt_policy`, which has an explicit type attribute of `critical_t`. The domain `foo_d` has an entry point program, `/usr/bin/login`, and that file therefore has an explicit domain attribute and it also inherits the type `unix_t`.

Attributes represented by map nodes are related to files by association with standard vnode structures that have been slightly extended to interact with the map node tree. At system initialization, the root vnode is associated with the root map node. Subsequently, all name resolution operations establish bindings so that every vnode is related to a map node. In the case that a map node exists for a file represented by a vnode, a name resolution operation attaches the vnode directly to the map node. If a map node does not exist, the name resolution mechanism attaches the vnode to its parent vnode; since every resolution operation operates from a known absolute or relative path, every new attachment is relative to a known vnode, and all vnodes are eventually connected to the map node tree through a chain of parent vnode pointers. To maintain parent vnode pointers, the DTE prototype references parent vnodes, resulting in a somewhat increased kernel memory requirement for active vnodes. Figure 7 shows the vnode associations that result from process access to two files, `/usr/george/papers/usenix` and `/usr/bin/login`. Because the login program's pathname is fully represented by map nodes, vnodes for the path attach directly. For the path to George's USENIX paper, the first two vnodes of the path connect directly to map nodes, and the rest point to the last map node in the path. Both files have the type `unix_t`, which is provided by the root map node.

By binding attribute values to vnode structures, the DTE prototype ensures that attributes are always available before they are needed even though the attributes may not be stored one-to-one on secondary storage. The DTE prototype retrieves attribute values of files using a simple algorithm that follows vnode parent pointers up until the first map node is reached and then optionally follows map nodes until the "governing" map node is reached.

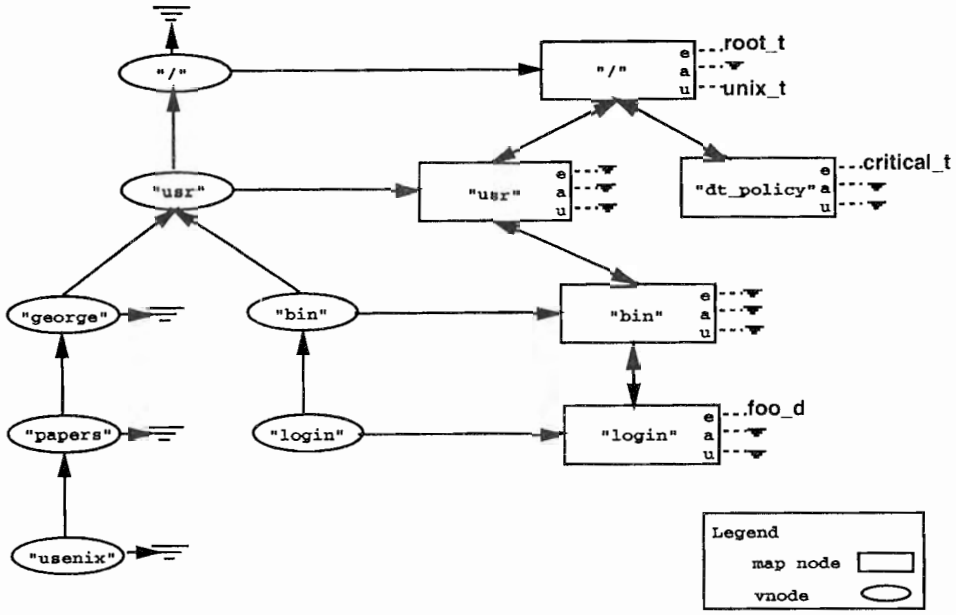

Figure 7. Attribute Associations.

Efficiency is a primary concern for the DTE prototype. The overhead of associating new vnodes with appropriate map nodes during name resolution is negligible, requiring a small and constant number of pointer manipulations. The attribute retrieval operation is a more likely cause of performance degradation, but we believe it is also small. In the DTE prototype, the UNIX kernel function `iaccess()` (and a handful of similar functions) call DTE functions that retrieve file security attributes. Most UNIX access control functions funnel down to the `iaccess()` function, which is called with great frequency since every system call requesting an operation on a pathname must call `iaccess` at least once for every component of the path. In the worst case, each attribute retrieval could require a search to the root map node. Given the modest depth of typical UNIX pathnames and the in-memory status of the map node tree, however, this appears small relative to other overheads of UNIX kernels. At the cost of additional complexity, however, various optimizations could be taken to short-circuit attribute retrieval searches as required.

6.3. Recovery Mechanisms

Although useful security configurations can be constructed that "lock down" the mappings between areas of the hierarchical filesystem name space and security

attributes, resulting in a static tree of map nodes, a more common case in our experience is to allow the map node tree to evolve as files are moved and created to reflect the needs of applications that use files. For example, an application might create a file of type `foo_t` in an area of the name space that inherits `bar_t`; such an event would add a DTEL assign statement, with its map nodes, to the system configuration. Similarly, a `rename()` operation may require that the map node tree be edited so that the rename operation doesn't inadvertently change the type of a file as a side effect. In general, the DTE prototype emulates the semantics of one-to-one attribute storage even though the attributes are not in fact maintained in that manner.

Given the criticality of accurate security attribute associations, dynamism in the map node tree introduces the need to maintain up-to-date associations even in the presence of system reboots or crashes. Writing map nodes to secondary storage poses an obvious risk to performance; the DTE prototype addresses this using a combination of alternate snapshot files and logging. Every thirty seconds, the map nodes are written to disk.[5] Additionally, more timely information is kept in two alternate log files: at system reboot, the most recent snapshot and log file is read to reconstruct the most recent valid state. The batched writes of the policy impose little overhead since no program waits for the writes to complete. In contrast, the log files require synchronous I/O and must be updated as little as possible.

Two basic classes of operations affect the map node tree: create operations and rename operations. In each case, the DTE prototype incurs no additional overhead if the operation does not produce an edit of the map node tree. If the operation creates a new object (e.g., a new empty file at an unused pathname or a rename to an unused pathname), recovery is simple since the attributes can be written first. Maintenance of DTE recovery information in this case requires one synchronous write operation in addition to the two synchronous write operations performed by UNIX to create or rename a file. If an operation overwrites an existing object, however, the use of implicit attributes complicates the recovery strategy; because every file is always associated with attributes inherited from the root directory, neither order of operations:

1. replace a file first and then record the new attribute, or
2. record the new attribute first and then replace the file,

prevents mislabeling if the system crashes between the two operations. To address this, the DTE prototype records this information as a sequence of optimized

5. For large policies, the mechanism could be enhanced to periodically write out only the changed portion.

transactions that makes sparing use of synchronous I/O and, most importantly, that never converts a memory speed operation to disk speed.

Both the create and rename VFS-layer operations can overwrite an existing file as a side effect. In the case of create, the UNIX VFS layer knows if there is an existing file to overwrite and truncates it for reuse with a new identity. To prevent a crash from relabeling existing file contents, the DTE prototype adds an fsync operation, ensuring that the file is empty, and then writes the new attribute to the log file, resulting in a worst-case scenario of two additional synchronous I/O operations for file creation.

A rename operation `rename(foo, bar)` is essentially:

```
unlink(bar);
link(foo, bar);
unlink(foo);
```

If bar exists, an update to a log file must be made conditional on successful completion of the rename operation or the log file update may relabel the original bar. The log file update cannot be written *after* the rename operation because a system crash could prevent writing of the update. For this operation, the DTE system writes an uncommitted transaction to the log file containing the file number of the file to be moved and, on the next write to the log file, piggy-backs the commit of the previous transaction. During system recovery, the last transaction can be verified through an examination of on-disk file numbers. This strategy holds the recovery I/O burden to at most one synchronous I/O for every rename operation.

In general, the prototype design requires no additional disk access on a per-system call basis. This approach promotes high performance since most DTE-related overhead is in memory operations where data structures can be optimized. For recovery, however, it is necessary to add disk writes during file creates that cause changes in the attribute association database. Depending on a system's configuration, it could be that none, some, or all file creates would cause attribute associations to change.

6.4. Network Implementation

In addition to associating attributes with files and processes and performing access control over those entities, the DTE prototype also inserts DTE attributes into IP datagrams and provides mediation of network messages. A fundamental goal of DTE network mediation is to preserve interoperability with non-DTE systems: this requires using existing IP, UDP, TCP, and NFS services and, as much as possible, preserving application layer protocols such as rsh and rlogin. Although we expect

that it will be useful to add DTE awareness to some additional network applications such as ftp and rdist, we believe that DTE systems must first be useful in networks of non-DTE systems.

Our general scheme is to add DTE attributes in the IP option space; these attributes are tokenized and currently consume 12 bytes of the 40-byte IP option space. DTE networking support at other layers is carried in these attributes at the IP layer. Due to the use of pipes and sockets in UNIX, a UNIX process may cause numerous IP datagrams to be generated and may not be aware of the network consequences of its actions. For the DTE prototype, each message is generated in the context of a process's domain and carries the domain's identity as the message's "source domain." Additionally, each message carries a type attribute; typically, each DTE domain has a default output type that labels messages generated from normal UNIX system calls such as `write()` and `send()`.

For each standard UNIX system call that can generate a message, the DTE kernel retrieves the calling process's domain and default output type from the DTE policy database generated using DTEL. Traditionally, UNIX systems employ a data structure, called an mbuf, that allows buffers of data to be chained together in a manner that facilitates the prepending and stripping of protocol headers in different layers of a UNIX kernel's protocol stacks. The DTE prototype uses a slightly extended form of the typical mbuf structure that provides header space for storing source domain and type identifiers. Standard UNIX system calls that send messages save these attributes in extended mbuf chains; at the bottom of the protocol stack, these attributes are extracted from the chains and encoded as IP options on a per-datagram basis. For received messages, the mechanism works in reverse, extracting received IP options and encoding them in mbuf chains for retrieval by receiving processes.

In addition to support for ordinary UNIX system calls, the DTE prototype provides a number of analogous DTE-specific system calls that allow processes to specify the type of data that they wish to send; DTE access control prevents processes from generating data types unless they have appropriate authorizations as specified in the DTEL specification.

In general, the DTE prototype treats every IP datagram as homogeneously typed; this simplifies access control over datagrams since a process using the raw IP interface, for example, can be allowed or denied access to a datagram based on its domain's access to the datagram's type. This strategy, although simple, does allow several ambiguous situations: for example, if a protocol such as TCP piggybacks control information in packets that also carry user data, should those packets have a protocol-specific type or a user type? Currently, our approach is to label packets with user types when they contain any user data and with protocol-specific types when they contain only protocol data. In the future, a natural extension to

the strategy may include a secondary "subsystem" label for use by protocol subsystems that are trusted to accurately carry user data. To minimize security mechanism, however, we are deferring secondary packet labels until a definite need has been demonstrated. In either case, the use of homogeneously typed datagrams simplifies the implementation of TCP substreams since TCP substreams are always made up of complete IP packets.

UNIX system calls that write data onto a TCP connection enqueue onto a single chain of mbufs associated with a TCP socket; the TCP sliding window processing breaks the data stream into separate IP datagrams based on a variety of criteria to optimize performance and guarantee that receipt of all the data is acknowledged before it is forgotten on the sending side. On the sending side, the DTE prototype implements TCP substreams by breaking the single mbuf chain into multiple chains when necessary to ensure that all the data of each chain has the same type attribute. The TCP sliding window processing has been modified slightly to generate a new datagram at chain boundaries. On the receiving side, this mechanism works in reverse to return substream type information that is then used both to mediate receive operations by processes and to deliver type information for use by DTE-aware processes.

A significant extension to the DTE prototype was required to implement DTE/NFS servers. Essentially, NFS file handles specify inode numbers that have no direct relation to the map nodes that implement implicit attributes for the prototype. A means was therefore required for mapping from inode numbers to map nodes. For directories accessed via NFS, the solution is simple since every directory contains a ". ." entry: using the ". ." entries, it is possible to reconstruct the portion of a pathname required to establish attribute values. The prototype currently carries out this reconstruction at every NFS file handle reception; however, temporarily raising the reference counts of heavily used vnodes probably would increase performance and prevent DTE overhead from being an NFS server bottleneck.

For files, NFS file handles alone do not provide a means of determining parent directories without an exhaustive search of file system inodes. To avoid this, the DTE prototype stores (file-inode-number, parent-directory-inode-number) pairs during NFS lookup operations in a cache. These entries provide a mechanism to reach the first directory that then allows pathnames to be reconstructed as necessary. To prevent the introduction of additional stale file handles at client applications when NFS servers crash or are rebooted, the cache must be maintained on secondary storage.[6] For intentional DTE/NFS server shutdowns, the secondary

6. The current DTE prototype does not write the cache to secondary storage.

memory cache can be written out before shutdown. To reduce the likelihood of stale file handles after DTE/NFS server crashes, the cache contents could be batch written at timed intervals with a minimal impact on performance. At somewhat greater cost, the secondary storage cache could be maintained during NFS lookup operations at the expense of extra I/O operations for memory cache misses.

7. *Related Work*

The work most related to DTE and its UNIX implementation falls into two general classes: access control systems and UNIX security mechanisms.

DTE is most closely related to mandatory access control techniques [Bell 1976; Boebert 1985; Biba 1977; Lipner 1982; Clark 1987] and type-enforcing systems [Boebert 1985; O'Brien 1991; Saydjari 1989; Sterne 1992; Wiseman 1986]. In general, DTE policies are a proper superset of the DoD lattice model [Bell 1976] and its integrity variation [Biba 1977]: DTE can be configured to provide a lattice but can also enforce nonhierarchical security policies such as assured pipelines [Boebert 1985] that drive information through policy-specified pathways of arbitrary connectivity and complexity. DTE can also be configured to provide integrity categories as in [Lipner 1982] and to support the transformation procedures and constrained data items of the Clark/Wilson model [Clark 1987].

Type enforcement was first proposed in [Boebert 1985] for the Secure Ada Target, a system later renamed LOCK [Saydjari 1989]. LOCK provides a Trusted Computing Base (TCB) on top of which a UNIX emulation layer, LOCKix [O'Brien 1991], provides UNIX services. As a consequence, the type enforcement mechanism controls UNIX emulations instead of individual UNIX applications and does not distinguish among multiple applications running on a single UNIX emulation. This limitation also exists for a Mach-based LOCK derivative [Fine 1993], which adds type enforcement to the Mach port, task, and virtual memory abstractions but provides no type enforcement within the UNIX emulation layer.

In [Sterne 1992], type enforcement was added to Trusted XENIX as a TCB subset. This system provides type enforcement at the UNIX system-call interface and can individually control UNIX applications. The TCB subset architecture prohibited change to low-level disk formats and mandated use of a separate runtime database to manipulate such attributes. This strategy is a precursor of the DTE runtime implicit type concept. That system also incorporated the notion of named entry point programs that must be executed to enter a domain. Type enforcement has also been integrated into at least one UNIX-based Internet firewall product,

the SCC Sidewinder system [Secure Computing Corporation 1994], but the authors are not aware of any published technical details.

A number of UNIX security controls and tools have been developed. Access Control Lists (ACLs) [Fernandez 1988] provide greater flexibility in UNIX discretionary access controls, and user-mode capabilities [Klein 1985] also allow finer-grained control over propagation of access rights, but both mechanisms are discretionary in nature and provide little protection against error-prone root programs. A variety of trusted UNIX systems have been implemented and evaluated against the Trusted Computer System Evaluation Criteria [National Computing Security Center 1985]. These systems typically provide MAC but lack the flexibility of DTE. Additionally, tools such as COPS [Farmer 1990] check for system misconfigurations but do not improve on the base UNIX security mechanisms themselves.

The Trusted Systems Interoperability Group (TSIG) has developed Internet draft standards for NFS and other protocols that support Multi-Level Secure (MLS) networking. These standards communicate significant amounts of information to represent security labels on subjects and objects that may "float" up dynamically and to represent process privileges that may be communicated across networks. For DTE, all of the required security information is contained in the relatively space-efficient type and domain identifiers carried in the IP-layer traffic, avoiding most changes to higher-layer protocols. An IP-layer standard [Kent 1991] currently exists for encoding military sensitivity labels in IP datagrams. While not ideally suited, this standard could potentially be used to transmit DTE attributes.

8. *Open Issues and Plans*

Our experience with the DTE prototype has uncovered or clarified several open issues:

policy complexity Although useful DTEL policies can be simple, some desirable policies appear to be complex. In particular, breaking root programs into as many small domains as possible holds promise for hardening UNIX but also results in a complex policy that could be difficult to evaluate or maintain. One possible strategy for ameliorating complexity is to add a module construct and name scoping features to DTEL. Policy complexity resulting from standard system interactions could then be reduced by decomposing the policy and encapsulating components with

well-defined interfaces. It may also be possible to construct new policies from older, reusable, building blocks.

policy coordination Networks of DTE systems will require techniques to remotely administer and maintain the consistency of DTE policies on numerous machines. Additionally, we expect that some systems will require different policies. Techniques are required to evaluate the safety of network compositions of heterogeneous policies and to safely administer DTE policies remotely.

device management Some devices, such as disks, potentially contain multiple types of data. Other devices, such as bitmapped displays, can mix data of different types. Attaching a single type to each device special file provides some pragmatic protection but does not mediate access to devices based on their actual contents. Additionally, access to some devices, such as ttys, could be further restricted if the devices could be temporarily bound to different types based on the domains of the accessing processes.

file descriptor inheritance Our strategy for remediating file descriptors when they are passed via exec operations to new domains is still evolving. For compatibility with UNIX applications, it might be useful to mediate and minimally restrict the access modes of inherited file descriptors according to the access rights possessed by a new domain, and then to restore them later if they are passed through another domain transition to a more powerful domain. This may, however, allow undesired interactions between domains based on shared resources, such as seek pointers, that are associated with file descriptors.

We are currently working on techniques to address these issues and are also exploring new applications of DTE. The most immediate and important one is the integration of DTE into Internet firewalls. We currently plan to integrate DTE into firewalls in three phases:

DTE Firewalls An integration of DTE into an Internet firewall and selected hosts. The goal of this integration will be to add defense-in-depth to the firewall security perimeter. The DTE firewall will direct traffic from specified external hosts or of specified protocols only to flow to internal DTE hosts that can confine any malicious effects. The primary goals will be to strengthen Internet firewalls by controlling the UNIX root privilege and to allow more network services to be safely imported into a LAN than is prudent without enhanced access controls.

Distributed DTE Firewalls An integration of IP-layer encryption with the DTE firewall. This phase will connect multiple DTE enclaves across the Internet.

Domain and Type Authority Service A DNS-like network service that will distribute digitally signed portions of DTEL policies. Communicating DTE hosts will authenticate to this service and use its DTE policy information as a basis for establishing appropriate inter-host trust relations and also for agreement on how data of specific types should be protected by communicating hosts.

This work will investigate how multiple hosts can exchange DTE information to negotiate network DTE policies, how DTE mechanisms can most effectively use encryption to protect DTE network attributes, how DTEL can be modularized to reduce policy complexity, and how DTE policies can be dynamically and safely extended or modified at runtime.

9. Conclusions

A central question in practical UNIX security is whether significant enhancements can be added in a way that is understandable, effective, and unobtrusive. This is a difficult question because applications and systems have evolved over time and now interact in subtle ways: practical security enhancements must allow existing programs to function properly while preventing unsafe interactions. DTE is an access control mechanism that uses a specification language to add simplicity and uses *implicit typing* to maintain compatibility and interoperability. This paper reports on ideas underlying DTE and facilities it provides for adding greater security to individual hosts, IP-based networking and NFS services, and on design considerations of a DTE UNIX prototype. Our primary results are positive and, although the DTE prototype is a research tool, we have used it internally to provide guest users with safely restricted access to our corporate data.

In summary, DTE has provided a useful research platform for building a hardened, compartmentalized UNIX system. In addition, DTE mechanisms appear suitable for interoperating and enforcing policies within networks that include some existing systems having no DTE controls. This capability is critical because any enhanced protection system must interoperate with existing systems through an extended transition phase as access controls are gradually adopted.

10. Acknowledgements

This work was funded by ARPA contract DABT63-92-C-0020.

We would like to thank Michael Petkac and Karen Oostendorp for their work validating DTE concepts through implementation of a number of DTE-aware network and X window utilities, and also Olafur Gudmundsson for his thoughtful comments on the manuscript.

References

1. L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, Practical Domain and Type Enforcement for UNIX, *1995 IEEE Symposium on Security and Privacy* (May 1995).
2. R. W. Baldwin, Naming and Grouping Privileges to Simplify Security Management in Large Databases, *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, page 116 (May 1990).
3. D. E. Bell and L. La Padula, Secure Computer System: Unified Exposition and Multics Interpretation, Technical Report No. ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, 1976.
4. K. J. Biba, Integrity Considerations for Secure Computer Systems, Technical Report No. ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, 1977.
5. W. E. Boebert and R. Y. Kain, A Practical Alternative to Hierarchical Integrity Policies, *Proceedings of the 8th National Computer Security Conference*, page 18 (1985).
6. M. Branstad, H. Tajalli, F. Mayer, D. Dalva, Access Mediation in a Message Passing Kernel, *1989 IEEE Symposium on Security and Privacy*, page 66 (May 1989).
7. D. D. Clark and D. R. Wilson, A Comparison of Commercial and Military Computer Security Policies, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, page 184 (1987).
8. DoD Information Security Program Regulation, DoD 5200.1-R, Washington, DC, June 1986.
9. D. Farmer, The COPS Security Checker System, *Proceedings of the Summer 1990 USENIX Conference*, page 195, 1990.
10. G. Fernandez and L. Allen, Extending the UNIX Protection Model with Access Control Lists, *Proceedings of the Summer 1988 USENIX Conference*, page 119, 1988.

11. T. Fine and S. E. Minear, Assuring Distributed Trusted Mach, *1993 IEEE Computer Society Symposium on Research in Security and Privacy*, page 206 (1993).
12. J. Ioannidis and M. Blaze, The Architecture and Implementation of Network-Layer Security Under UNIX, USENIX Summer 1994 Technical Conference, (1994).
13. S. Kent, *Security Options for the Internet Protocol*, RFC 1108, November 1991.
14. D. Klein, A Capability Based Protection Mechanism Under UNIX, *Proceedings of the 1985 Winter USENIX Conference*, page 152 (1985).
15. J. Kohl and C. Neuman, *The Kerberos Network Authentication Service (V5)*, RFC 1510, September 1993.
16. C. E. Landwehr, C. L. Heitmeyer, and J. McLean, A Security Model for Military Message Systems, *ACM Transactions on Computer Systems*, 2(3): 198-222, August 1984.
17. S. B. Lipner, Non-Discretionary Controls for Commercial Applications, *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, page 2 (1982).
18. F. L. Mayer, *Security Controls for an Automated Command and Control Information System (ACCIS): Baseline Definition*, Technical Report TISR-201, Trusted Information Systems, Glenwood, MD, May 1989.
19. M. K. McKusick, The Virtual Filesystem Interface in 4.4BSD, *Computing Systems*, 8(1): 3–25, Winter 1995.
20. National Bureau of Standards, *Data Encryption Standard*, Federal Information Processing Standards Publication 46, Jan. 1977.
21. National Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, Dec. 1985.
22. National Computer Security Center, *Final Evaluation Report of Honeywell Multics MR11.0*. National Computer Security Center, Ft Meade, MD, CSC-EPL-85/003, June 1, 1986.
23. National Computer Security Center, *Final Evaluation Report, Trusted Information Systems, Inc. Trusted XENIX*, National Computer Security Center, Ft Meade, MD, January 1991.
24. R. O'Brien and C. Rogers. Developing Applications on LOCK, *Proceedings 14th National Computer Security Conference*, pages 147–156, (October 1991).
25. E. I. Organick, *The Multics System: An Examination of Its Structure*, Cambridge, MA: MIT Press, 1975.
26. O. S. Saydjari, J. M. Beckman, and J. R. Leaman, LOCK Trek: Navigating Uncharted Space, *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, page 167 (1989).
27. M. D. Schroeder, Cooperation of Mutually Suspicious Subsystems, PhD dissertation, M.I.T., 1972.
28. Secure Computing Corporation, Sidewinder Press Release, October 10, 1994.
29. D. L. Sherman, D. F. Sterne, L. Badger, S. L. Murphy, K. M. Walker, S. A. Haghighat, Controlling Network Communication with Domain and Type Enforcement, to appear, *Proceedings 18th National Information Systems Security Conference*, page 211 (Oct. 1995).

30. D. Sterne, A TCB Subset for Integrity and Role-Based Access Control, *Proceedings 15th National Computer Security Conference*, pages 680–696 (1992).
31. D. J. Thomsen, Role-based Application Design and Enforcement, In S. Jajodia and C. E. Landwehr, editors, *Database Security, IV: Status and Prospects*, pp. 151–168, New York: North-Hollard, 1991.
32. S. Wiseman, A Secure Capability Computer System, *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, page 86 (1986).