Practical Domain and Type Enforcement for

UNIX*

Lee Badger
Daniel F. Sterne
David L. Sherman
Kenneth M. Walker
Sheila A. Haghighat

Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, Maryland 21738

Abstract

Type enforcement is a table-oriented mandatory access control mechanism well-suited for confining applications and restricting information flows. Although both flexible and strong, type enforcement alone imposes significant administrative costs and has not been widely adopted. Domain and Type Enforcement (DTE) is an enhanced version of type enforcement designed to provide needed simplicity and compatibility. Two primary techniques distinguish DTE from simple type enforcement: 1) DTE policies are expressed in a high-level language that includes file security attribute associations as well as access control information, and 2) during system execution, DTE file security attributes are maintained using a concise human-readable format in a runtime DTE policy database, thus removing the need for security-specific low-level data formats. Such formats are a major source of incompatibility for security-enhanced systems. A DTE UNIX prototype system has been implemented to evaluate these primary DTE concepts. This paper presents experiences gained and preliminary results indicating that DTE can provide cost effective security increases to UNIX systems while maintaining a high degree of compatibility with existing programs and media.1

1 Introduction

Trusted systems principles and technologies to date have primarily addressed the DoD mandatory confidentiality policy and its interpretations for computer systems [5, 17, 29, 22]. There is growing recognition, however, that systems are needed that can satisfy other kinds of mandatory security requirements, including integrity [2, 6, 7, 12, 19, 3] and prevention of service denial [21, 16]. Although a variety of security frameworks and mechanisms have been developed for expressing mandatory security requirements other than confidentiality, relatively few have been adopted by mainstream operating system vendors. One reason may be that such 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 security: can significant security enhancements be added to mainstream operating systems in a way that is understandable, effective, and unobtrusive?

This paper describes techniques for substantially improving the security of UNIX systems while maintaining a high degree of backward compatibility and avoiding significant increases in administrative overhead. To validate these techniques, we have implemented a prototype UNIX system; this paper presents preliminary results based on experience with the prototype system. Our work is based on an enhanced version of type enforcement [7], a table-oriented mandatory access control mechanism first proposed for the LOCK Trusted Computing Base (TCB) [26, 25]. Although both flexible and strong, type enforcement alone imposes significant administrative costs and has not been widely adopted by operating system vendors. Domain and Type Enforcement (DTE)[1] is an enhanced version of type enforcement designed to provide needed simplicity and compatibility. Two primary techniques distinguish DTE from traditional type enforcement: 1) DTE policies are expressed in a high-level language that includes configuration of security attributes (e.g., type/file associations) as well as other access control information and 2) during system execution, DTE attributes (e.g., file types) are maintained using a concise human-readable format in a runtime DTE policy database, thus preserving media compatibility with systems that are not DTE-aware.

This research was supported by ARPA contract DABT63-92-C-0020.

1 UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.

This research was supported by ARPA contract DABT63-92-C-0020.
rather than theoretical results. In this vein, we have designed our prototype system’s access controls with the following goals in mind:

**strength** They should be strong enough to support significant increases in assurance, including the ability to confine process execution and control information flow; in particular they should be capable of constraining the accesses of privileged system processes (e.g., UNIX processes that execute with “root” privilege) so that it is possible to prevent a single process that has been subverted from subverting an entire operating system.

**flexibility** They should be flexible enough to satisfy a wide range of access control requirements and policies.

**simplicity** They should be relatively easy to understand, configure, and administer.

**compatibility** They should be backward compatible and interoperable with existing systems, programs, and data formats.

**performance** They should be implementable with high performance.

This paper first describes type enforcement and the principle enhancements that constitute DTE. Next we review experience with a UNIX prototype and analyze the feasibility of DTE in various contexts. Finally, we review related work and discuss future directions.

2 Domain and Type Enforcement

Type enforcement is a table-oriented form of access control originally proposed by Boebert and Kain [7] and later refined in [25]. 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). In this scheme, 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 lookup: 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 attempts to access another subject, 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.

Although type enforcement is both flexible and strong, 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.** Mandatory Access Control (MAC) and, to an extent, Discretionary Access Control (DAC), reflect existing organizational security policies (e.g., the DoD Information Security Program Regulation [13]). For MAC, files typically have the security labels that correspond closely 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 from the type enforcement tables must (currently) be custom-engineered for each security-relevant application.

Domain and Type Enforcement (DTE) is an enhanced version of type enforcement designed to address these issues. Two primary techniques distinguish DTE from simple type enforcement:

1. **DTE policies are specified in a high-level language suitable for expressing reusable access con-
trol configurations. A DTE specification includes security attribute associations such as type/file associations as well as other access control information. 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. Implicit typing simplifies security configuration establishment 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 Language Support

DTE Language (DTEL) is a high-level symbolic language for expressing reusable DTE configurations in a human—rather than machine—friendly form. DTEL security attributes such as domain definitions express fundamental constraints on subject creation and object accesses; consequently a DTEL specification must be effective from an early stage in a system’s initialization. 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 configura-

tion: the type statement, the domain statement, the initial domain statement, and the assign statement. 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. 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 */

A DTEL domain statement defines three components:

entry points Programs, identified by path name, that are bound to the domain and must be in-

---

2 We have also formulated but not completely implemented a DTEL mount statement that controls 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.
voked 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.

**Subject access rights** Permissible modes of access, when executing in the domain, to subjects in other specified domains. In addition to providing 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 DTEL, 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 functionalities.

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:

```c
#define DEF (/bin/sh), (/bin/csh), \n (rwx->unix.t)
domain engineer.d = DEF, (rwx->specs.t);
domain project.d = DEF, (rwx->budget.t), (rd->rates.t);
domain accounting.d = DEF, (rd->budget.t), (rd->rates.t);
```

Each domain is a list of tuples where each tuple either contains a UNIX path or a collection of access modes (designated by "->") a collection of type or domain names. The `domain` statement collapses the DDT and DIT into one representation. These domains regulate observation and modification to support three kinds of user authorizations supporting 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 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:

```c
domain system.d = (etc/init), (rwx->unix.t), (auto->login.d);
domain login.d = (bin/login), (rwx->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 case, note that because of the circular flow of information between the project leader and the account, this policy cannot be represented with a lattice-based mandatory access control mechanism without using trusted subjects.

---

3 Note that, because of the circular flow of information between the project leader and the account, this policy cannot be represented with a lattice-based mandatory access control mechanism without using trusted subjects.
scenario, the login program is DTEL-aware and properly authenticates and checks the authorisation 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 every 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 uni.x.t. In three subdirectories, however, uni.x.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 "s") 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 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\(^4\) may appear to have multiple types. To prevent this, 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 DTEL 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

\(^4\)Symbolic links are not an issue because they merely name hard link paths represented by DTEL assign statements.
shared memory, semaphores, and message queues. Figure 3 shows the completed example policy, which provides three user-oriented domains and all mechanisms required to support them on a typical DTE UNIX system. This example policy can be incrementally refined to add additional user domains, distinguish between console and network user sessions, simultaneously support additional organisational 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, such policies can also be standardized and portable (or configurable via macros) between UNIX systems.

4 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 [8], Trusted XENIX [9], TMach [10], 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 analysed 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.

Instead, our approach maintains type associations in a UNIX kernel-resident runtime policy database that is established at system boot time. All UNIX file object accesses start with a name resolution mechanism that converts pathnames to object handles that are 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 many policies (like the example in section 3), 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 inherited from the new path name is different from the type inherited from the old path name.

To provide recoverability in the face of power interruptions, the runtime policy database writes modifications out to permanent storage. In the case of a file creation, an ordinary UNIX system performs two disk operations, one to write a new inode and one to add a new directory entry to point to it. If the file creation causes a change to the runtime policy database, a DTE system performs a third disk operation to record the new assign statement in a DTE log file. Similarly, an ordinary UNIX rename event requires several disk operations (the OSF/1 UNIX system we use requires three synchronous disk operations) and, if the rename event modifies the runtime policy database, a DTE system performs only one additional disk operation. To preserve performance, a DTE UNIX system performs I/O only in the context of existing UNIX I/O; no memory-speed operation has been reduced to disk-speed. To facilitate administration of evolving attribute associations, all DTE disk operations write assign statements in ASCII using DTEL syntax. Periodically, the entire set of attribute associations are sorted and written to alternating "snapshot" files to provide the system manager with a current view of type associations. In the absence of administrative action to reconfigure DTE, these snapshot files, augmented by the DTE log files, are used for recovery at each reboot. This strategy facilitates system administration while maintaining interoperability with existing systems because implicit types can be associated with files without regard to their number, names, locations, or storage formats.

5 Experience With a UNIX Prototype

To validate our techniques, we have implemented a prototype DTE UNIX system. This system implements most of the concepts of DTEL, including implicit attribute management, type and domain defni-
/ *
 * DTEL Commercial Policy.
 */

type unix.t, /* normal UNIX files, programs, etc. */
specs.t, /* engineering specifications */
budget.t, /* budget projections */
rates.t; /* labor rates */

#define DEF (/bin/sh), (/bin/csh), (rxd->unix.t) /* macro */

domain engineer.d = DEF, (pwd->specs.t);
domain project.d = DEF, (pwd->budget.t), (rd->rates.t);
domain accounting.d = DEF, (rd->budget.t), (pwd->rates.t);
domain system.d = (/etc/init), (rxd->unix.t), (auto->login.d);
domain login.d = (/bin/login), (pwd->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

of functions, and context checking (e.g., domains don’t reference nonexistent types, etc.). The prototype consists of a DTE subsystem, including the DTEL compiler and support routines for access control, and an integration of the subsystem into an OSF/1 MK 4.0 UNIX server. Because DTE requires no changes to low level formats, DTE implementations should be relatively portable between UNIX kernels; to test this, we have also ported the prototype to TMach Version 0.2. The DTE modifications consist of roughly 17,000 lines of commented C, lex, and yacc code, of which 3,600 lines comprise the DTEL processor.

As the prototype boots, it reads its DTEL specifications, confines all processes in specified domains, associates type labels with all files, and mediates accesses based on these attributes. Additionally, the prototype labels and mediates communication over UNIX pipes and sockets. The prototype controls UNIX-domain sockets internally and controls Internet-domain sockets by typing IP datagrams, UDP datagrams, and TCP streams. A detailed discussion [27, 23] of that work is beyond the scope of this paper and will be separately published. We have not yet added mediation for UNIX shared memory segments, message queues, or semaphores; those extensions are straightforward and will be added later.

In addition to UNIX kernel-level modifications, we have also made several applications DTE-aware. Most significantly, we have implemented a DTE version of the login program that authenticates users for specific roles [1, 20, 4, 30] and then confines user sessions to specific environments using domain transitions authorized by the DTE policy. To allow users to view and, within DTE constraints, manipulate DTE attributes, we have implemented DTE-aware versions of the ls, ps, mkdir, and in programs. These programs list directory contents and process states like the standard versions except that they also accept new arguments to display security attributes. To analyze DTE policies prior to use in the UNIX kernel, we have linked the DTE subsystem library into a test harness that checks for syntactic and contextual correctness and prints various reports. We have also implemented a modified version of the Emacs text editor that displays type attributes of file buffers and allows users to simultaneously view and manipulate labeled information in multiple windows.

Figure 4 shows the general structure of the prototype system. An OSF/1 UNIX server runs either on a Mach kernel or on the TMach [10] Trusted Computing Base. UNIX processes call the UNIX system call interface provided by the OSF/1 UNIX server. To directly control individual UNIX processes, we have placed DTE mechanisms in the OSF/1 UNIX server, adding new access control constraints to some existing UNIX system calls and new system calls for DTE-aware processes. We believe that DTE must be implemented at the UNIX system call interface in order to
satisfy this objective. This approach is consistent with commercial grade assurance and, from previous evaluation experience with trusted UNIX systems, we believe such an implementation could achieve TCSEC B1 or possibly higher levels of assurance. DTE controls could also be placed lower, in the Mach (or TMach) layer; however, adding DTE to the lower layer would not increase assurance because DTE controls would still be required in the UNIX layer.

Much of our attention has focused on security issues for the commercial UNIX community; however, we are also investigating how DTE can supplement traditional DoD security policies, and we are currently experimenting with DTE on TMach Version 0.2. The TMach-based DTE prototype provides DTE controls for supporting integrity and role-based policies in addition to TCSEC [24] B3 class confidentiality. Our integration strategy, as shown in figure 4, essentially adds DTE into the standard TMach OSF/1 UNIX server provided with TMach and leaves undisrupted the TMach controls in the system's lower layers. Although the standard TMach OSF/1 UNIX server contains a significant amount of TMach-specific code, such as low-level routines using the TMach Trusted File System, the integration of DTE into that server required no changes to TMach and almost no changes to the DTE implementation. Due to the TMach strategy of running multiple operating system personalities, such as OSF/1, concurrently at different security levels, access control enhancements such as DTE may require coordination between personalities running concurrently at different security levels. Initially, we have used a minimal-coordination strategy to gain experience with composite DTE/MLS policies. A complete description of design considerations for adding DTE to operating system personalities running on TMach is beyond the scope of this paper and will be separately published.

We currently use the DTE UNIX prototype on our development machines and typically use the system to build itself. This practice makes us particularly sensitive to achieving high performance as well as better security. Additional access controls in principle must incur some additional overhead, but our subjective experience is that our DTE prototype system performs as well as the OSF/1 UNIX system on which it is based. We attribute this high level of performance to the fact that DTE mediation rarely requires extra disk I/O and never requires I/O when the base system would not already have been performing I/O. DTE mediation within the UNIX kernel does not require complex algorithms and, as a consequence, DTE
mediation overhead is minimal.

Our experience indicates that DTE could be beneficial for implementing a variety of organizational security policies on UNIX workstations. In particular, we have found DTE to be promising in two significant areas: 1) confinement of root programs and 2) confinement of guest users to our corporate file servers.

5.1 UNIX "root" Confinement

Traditional UNIX places no access control restrictions on the UNIX root user, and UNIX system lore is replete with examples of root programs being tricked into using their privileges to steal or destroy data. On a DTE system, root programs could be bound, in both single-user and multiuser modes, to domains that allow them to operate normally but prevent them from accessing files or processes unrelated to their functions. To support the concept of least privilege, we are currently experimenting with domain definitions that partition UNIX processes into tightly confined domains and run virtually every UNIX daemon in a separate domain specialized for its requirements. We now have an experimental DTE policy that runs the UNIX system processes in 27 different domains based on least privilege. A number of key DTE design decisions have in fact resulted from compatibility requirements identified during this effort. A key question for this effort is whether there are intrinsic dependencies that will prevent effective partitioning for least privilege. Although our results are still preliminary, they are positive and we are continuing to refine DTE policies with the goal of developing standard "hardened UNIX" configurations for our own use and possible later distribution.

5.2 Network Reference Monitor

One consequence of runtime-implicit typing is that file formats are not relevant to DTE. DTE controls can therefore be applied even over files that are imported to a DTE system from a non-DTE file server using a network file system (NFS) protocol. We are currently gaining operational experience using DTE to mediate access to our corporate file servers on behalf of guest users whom we authorize to observe part (but not all) of the UNIX world-readable data stored there. In this application, guest users have accounts only on DTE NFS client workstations. The DTE workstations implement a reference monitor that mediates access to network files distributed around a local network even though the systems providing the files are unchanged and not aware of DTE.

Figure 5 shows the resulting interactions. In that figure, a guest user is logged into a DTE system. The DTE system allows the guest user DTE-mediated access to local files. Additionally, the DTE system imports files from a non-DTE file server, locally associates DTE types with those files, and mediates the guest user's access to the imported files based on their DTE attributes. As displayed in figure 5, the DTE system associates the type Non_sensitive Data to some imported files and the type Proprietary Data to others. To the guest user, it appears as though the entire network environment provides DTE access controls. We are finding this to be a significant application of DTE since it is often useful to grant limited access to our corporate data.

6 Related Work

DTE is most closely related to mandatory access control techniques [5, 7, 6, 19, 12] and type-enforcing systems [7, 25, 26, 28, 31]. In general, DTE policies are a proper superset of the DoD lattice model [5] and its integrity variation [6]; DTE can be configured to provide a lattice but can also enforce nonhierarchical security policies such as assured pipelines [7] that drive information through policy-specified pathways of arbitrary connectivity and complexity. DTE can also be configured to provide integrity categories as in [19] and to support the transformation procedures and constrained data items of the Clark/Wilson model [12].

Type enforcement was first proposed in [7] for the Secure Ada Target, a system later renamed LOCK [26]. For LOCK, the original Domain Transition Table of [7] was replaced with the Domain Interaction Table (DIT), which provides inter-domain access rights more suitable for processes instead of procedures, as in [7]. LOCK policies as expressed in the DIT do not bind programs to domains, although this effect can be had through careful configuration of LOCK's Domain Definition Table and file-to-type bindings. LOCK provides a UNIX emulation layer, but the type enforcement controls are implemented by the LOCK TCB, not by the untrusted UNIX emulation running on top. As a consequence, LOCK controls do not distinguish among multiple applications running on a single UNIX emulation. This limitation also exists for a Mach-based LOCK derivative [14], which adds type enforcement to the Mach port, task, and virtual memory abstractions but provides no type enforcement within the UNIX emulation layer.

In [28], 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. This variation of type enforcement adds bindings from domains to specific executable programs, which allows the type enforcement policy to limit domains to specific applications. A central architectural feature of the Trusted XENIX-based system is that the type enforcement is a TCB subset and therefore implemented mostly on top of an existing TCB using an outer hardware ring. The TCB subset architecture prohibited any change to low-level disk formats and mandated use of a separate runtime database to manipulate such attributes. This strategy is a precursor of and has inspired the DTE runtime implicit type concept.

7 Future Directions

We are currently exploring a number of enhancements that could make DTEL more useful and increase assurance that DTE policies are correctly specified. One enhancement is to add a small logical assertion language to DTEL, that allows predicates (involving information flow, for example) over a policy to be expressed and automatically checked. To assist policy reusability, DTEL could also be modularized so
that a policy is a collection of policy modules. We believe that DTEL subpolicy modules could significantly simplify specifications as well as provide hooks for supporting interoperability between systems running different but related DTEL specifications. Dynamic changes to a DTEL specification to accommodate new domains and types appear both feasible and useful. Modularity in DTEL may be a suitable basis for such changes as well as for supporting prepackaging and reuse of policy components in DTE systems and applications.

Research on tools that analyze non-DTE system configurations and generate reusable DTEL subpolicy modules holds promise for automating application of DTE policies to many systems. An important class of extensions would tie different DTE policies on separate hosts together in a dynamic and coordinated way, forming distributed networks of DTE systems implementing consistent access controls over communicated data. This composition would require development of protocols and rules for dynamic policy modification authority. An extended version of DTE and DTEL could provide facilities for expressing the relationships between local DTE policies so that information in a cooperating network of DTE systems is consistently handled and so that a set of specified DTE security perimeters is enforced by the participating systems.

A final major area for future development is research into combining and coordinating DTE with Internet firewalls [11] and selected hosts to add depth of defense to the standard firewall perimeter defense. A DTE firewall [15] alone cannot control import or export of sensitive information because such information can be tunneled through firewall defenses using supported protocols (e.g., mail). A DTE firewall in coordination with selected DTE hosts behind the firewall, however, can control information flows and holds promise both to strengthen traditional firewall security perimeters and to enable the safe use of Internet services and applications (such as WWW/Mosaic) behind firewall security perimeters.

8 Conclusions

A central question in practical security is whether significant security enhancements can be added to mainstream operating systems in a way that is understandable, effective, and unobtrusive. This problem differs from that faced by the authors of the TCP-SEC in that security in this context is not necessarily "designed in" as a primary concern but must be useful and cost-effective even when added much later. The premise of this paper is that access controls based on domains and types are powerful enough to provide substantial and practical increases in security, including support for a variety of policies that require containing, protecting, and isolating information. In our view, administrative complexity and incompatibility have been obstacles to the broad acceptance of type enforcement. Our research suggests that a high-level, human-oriented specification language (DTEL) can facilitate natural expression of information flow, containment, and other constraints that are fundamental to a variety of security policies and that support integrity and reliability in complex applications and networked systems. A primary feature of our specification technique is to include attribute associations, in a very compressed form, as a part of policy specifications and to maintain them in this form at runtime. This feature substantially reduces access control complexity and supports application of policies to both existing
media and network-based foreign objects. DTE systems may therefore interoperate and enforce policies within networks of 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.

References


