The following paper was originally published in the
Proceedings of the Fifth USENIX UNIX Security Symposium
Salt Lake City, Utah, June 1995.

A Domain and Type Enforcement UNIX Prototype

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

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

A Domain and Type Enforcement UNIX* Prototype

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

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 reviews the primary DTE concepts, discusses their application to IP networks and NFS, and then describes the design and implementation of a DTE UNIX prototype system.

1 Introduction

As UNIX systems become a major part of the National 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, which 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, lpd, 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. This benefit, however, has not been realized by mainstream UNIX systems even though a number of access control mechanisms [4, 2, 6, 9, 8, 18] 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

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

form of access control, Domain and Type Enforcement (DTE) [1] 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 formulated specifically to address requirements of greatest concern for both vendors and users, namely: flexibility, simplicity, operating system interoperability, binary application compatibility, and performance. This paper reviews DTE, [1] discusses how DTE can be applied to IP networks and NFS and then discusses design and implementation issues of the DTE UNIX kernel. Finally this paper reviews related work and discusses our plans for further development of DTE over the next few years.

2 DTE

DTE is an enhanced form of type enforcement, a table-oriented access control mechanism originally proposed by Boebert and Kain [9] and later refined in the LOCK system [21]. 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 type enforcement for UNIX, an access control attribute called a *domain* is associated with each subject (process), and another attribute called a *type* is associated with each object (file, message, shared memory segment, etc.). A global table, the Domain Definition Table (DDT), represents allowed access modes between domains and types (e.g., read, write, execute), and another table, the Domain Interaction Table (DIT), represents allowed access modes between domains (e.g., signal, create, destroy). As a system runs, access attempts are mediated using table lookups: access attempts for modes not authorized in the tables are denied.

Although type enforcement is very flexible, the access control tables can quickly become too complex, and type enforcement is difficult to use in practice. Additionally, the presence of type attributes on files appears to require a new and incompatible file system format. To address these issues, DTE enhances type enforcement in two ways:

1. DTE policies are specified in DTE Language (DTEL), a high-level language suitable for expressing reusable access control configurations that are compatible with current applications and system configurations.

2. During system execution, DTE file security attributes are not stored one-to-one with files on disk, but are instead maintained *implicitly* in a form that capitalizes on the directory hierarchy to compactly represent portions of a file hierarchy that have identical attributes. Using *implicit typing*, DTE can therefore be applied to existing files with no change to file system formats.

DTE is a configurable, kernel-level access control mechanism. At each system boot, a DTE UNIX system processes a DTEL specification and establishes access controls during UNIX kernel initialization. All processes, including root processes, are subject to DTE controls. DTEL currently provides four[2] primary statements for expressing a DTE configuration:

type Declares one or more object types to be available to other parts of a DTEL specification.

domain Expressed as a list of tuples, defines a restricted execution environment composed of three parts: 1) "entry point" programs, identified by pathname, that a process must execute in order to enter the domain (e.g., `(/bin/login)`), 2) access rights to types of objects (e.g., `(rwx->foo_t)`), and 3) access rights to subjects in other domains (e.g., `(sigkill->user_d)`). A DTEL domain controls a process's access to files, a process's access via signals to processes running in other domains, and a process's ability to create processes in other domains by executing their entry point programs. For backward binary compatibility, the domain statement also provides an access designator to force domain transitions on older programs that are not aware of DTE: if a domain A has *auto* access rights to another domain B, a subject in A automatically creates a subject in B when it executes, via exec(), an entry point program of B.

initial_domain Selects the domain of the first process.

assign Associates a type with one or more files. An assign statement may be recursive, in which case it applies to a directory and everything below, and one assign statement may override another; for instance, an assign statement for `/tmp/foo` may override a recursive assign statement for `/tmp`.

[1] DTE is described in more detail in [1].

[2] For brevity we omit peripheral DTEL statements and features and also restrict our attention here to implemented features with which we have actual experience.

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

Figure 1: Example DTEL Policy

An important goal for DTE is to superimpose useful security policies on existing UNIX configurations while using implicit typing to maintain backward compatibility with existing data formats and applications. Figure 1 shows a DTEL specification of a commercial policy designed to provide data protection and user authorizations in an engineering organization. To validate that our example specification is not trivial, we have run it on our prototype DTE system and found it to provide useful protection. This specification provides three types of protected user data, one type of system data, three user domains, and two supporting system domains. The user domains correspond to job descriptions, such as engineer or accountant, and the system domains provide operating system support. Additionally, this specification assigns type attributes to all files.

A DTE system running the specification of figure 1 starts the first process in the `system_d` domain, which is then inherited for all other system processes except the login program. The specification uses the `auto` mechanism to run login in the `login_d` domain even though the existing getty program does not request the domain transition. The `login_d` domain has the authority to create the user domains (`engineer_d`, `project_d`, and `accounting_d`), based on user authentications. Each user login session is confined by one of the user domains controlling access to protected data, which resides in three directories under `/projects`. Though simple, this sample specification 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 its root daemons in tightly constrained domains.

3 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. 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 always 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 always 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.

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 addition to maintaining compatibility with UNIX network abstractions and application-level protocols, it is also necessary to define how DTE systems interoperate with non-DTE systems. 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. Using DTEL, a DTE system can associate a single domain with the "universe" of foreign non-DTE hosts, associate a different domain to each class A, B, or C network, and finally associate specific domains to individual non-DTE hosts that, for various reasons (such as quality of administration), are more or less trustworthy than their LAN. 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 SunOS 4.1.1 on Sun 3 systems, in particular, 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 be currently running 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 negotiation functions that ensure that different machines "mean" the same thing by DTE attributes they ex-

[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 [15], integrity, and confidentiality [10, 11] as appropriate.

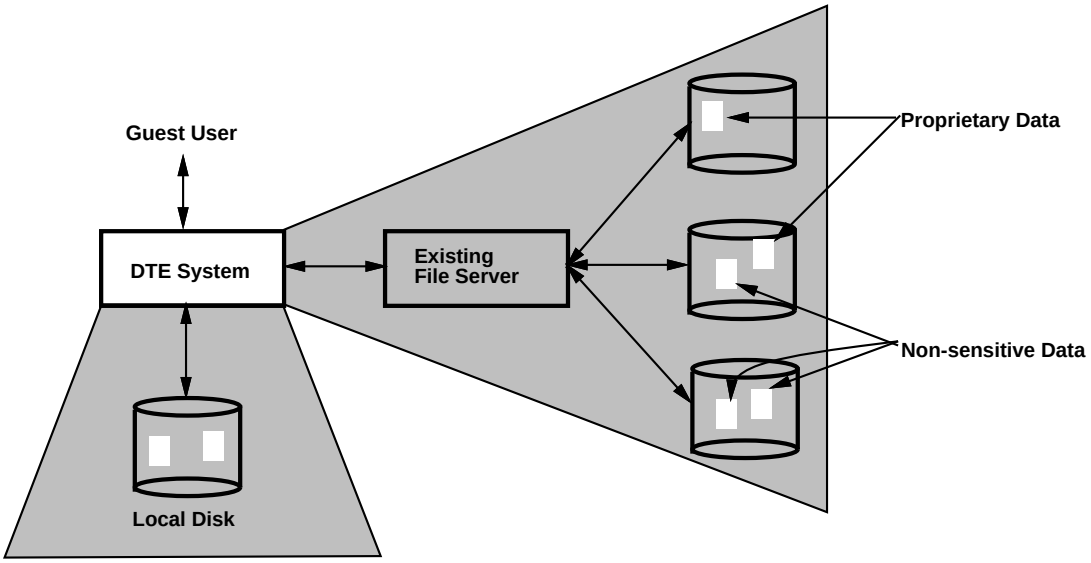

Figure 2: DTE NFS Clients

change. Although we only have experience to date with UDP and TCP, our techniques appear to apply to raw IP, and potentially also to multicast protocols such as ISIS [5] and PSYNC [22].

4 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 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 [1] 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 2 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 "non sensitive 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 non-sensitive data.

DTE network features allow 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 pathnames. 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, all NFS requests are labeled by the client system with the source domain of the re-

questing process. The DTE/NFS server then uses the source domain as a client credential to consult the system's DTEL specification and determine whether the request is authorized. 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 name 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 non-directory 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 highly 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 and are available only to a system administrator. By making certain types of files inaccessible to the NFS daemon, DTE provides a strong additional mechanism that can be employed by administers 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 requires only a small amount of additional I/O that only occurs in the context of I/O already required by NFS.

5 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 [7], a high-assurance trusted computing base designed to satisfy DoD security requirements as specified in the Trusted Computer System Evaluation Criteria [20]. 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 3 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

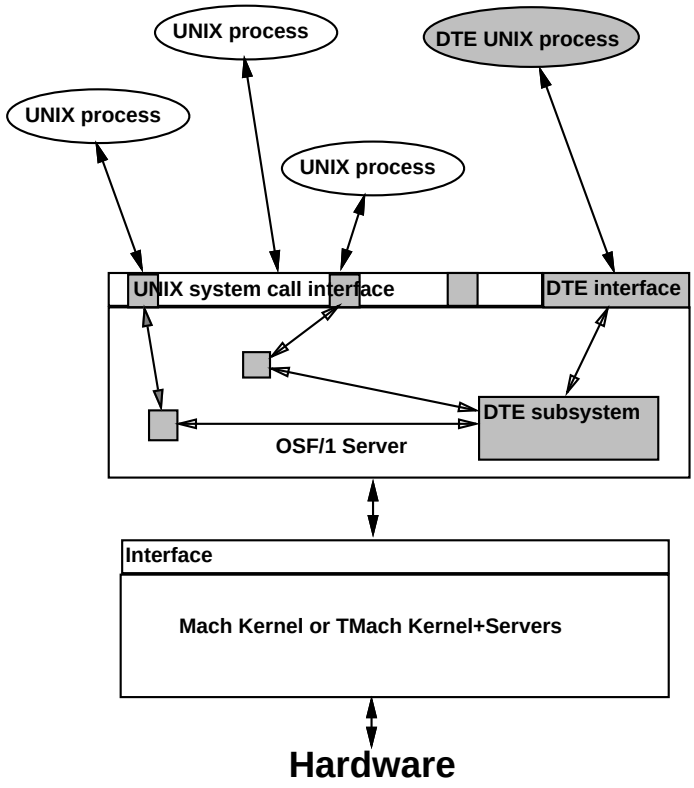

Figure 3: DTE System Architecture

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 [17, 3, 26] 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 a number of UNIX utilities such as ls and ps, and we have implemented a DTE-aware version of 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 1). 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.

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.

5.1 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 [19] 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 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.

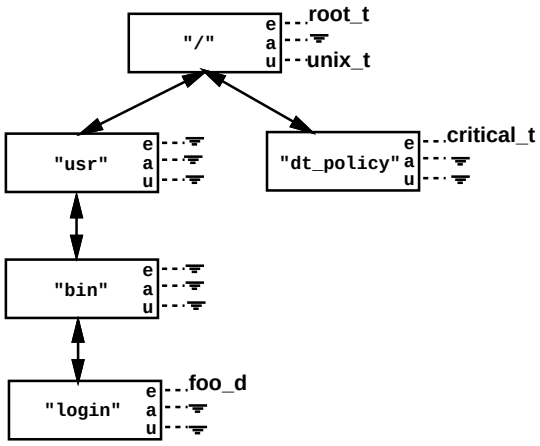

Figure 4: Map Nodes

Each path provided to 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 a "-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 4.

```
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 4, "e" for explicit, "a"

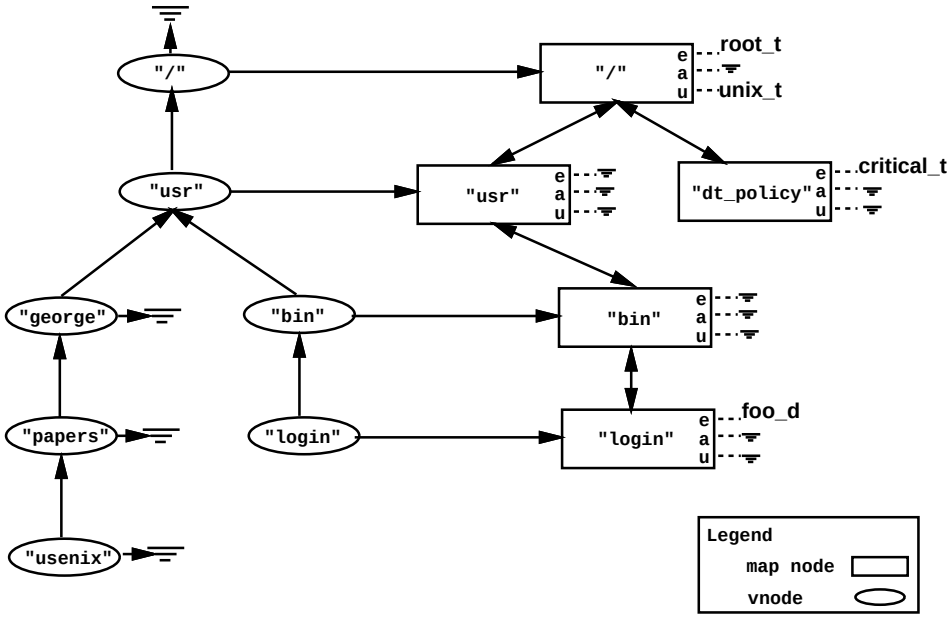

Figure 5: Attribute Associations

for *implicit at*, and "u" for *implicit under*). Figure 4 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 5 shows the vnode associations that result from process access to the 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.

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.

5.2 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.[4] 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 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,

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

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.

5.3 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 network applications such as rcp 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 piggy-backs 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 critera 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 where 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, the on-disk representations do not imply parents 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 any possibility of introducing additional stale file handles at client applications, the cache must be maintained on secondary storage. For intentional DTE/NFS server shutdowns, the cache can be written out only before shutdown. To avoid stale file handles after DTE/NFS server crashes, the cache must be maintained during operation. In this case also, the cache contents can be batch written at timed intervals, resulting in a minimal impact on performance.

6 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 [4, 9, 6, 18, 8] and type-enforcing systems [9, 21, 25, 24, 27]. In general, DTE policies are a proper superset of the DoD lattice model [4] and its integrity variation [6]: DTE can be configured to provide a lattice but can also enforce nonhierarchical security policies such as assured pipelines [9] that drive information through policy-specified pathways of arbitrary connectivity and complexity. DTE can also be configured to provide integrity categories as in [18] and to support the transformation procedures and constrained data items of the Clark/Wilson model [8].

Type enforcement was first proposed in [9] for the Secure Ada Target, a system later renamed LOCK [25]. LOCK provides a Trusted Computing Base (TCB) on top of which a UNIX emulation layer 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 [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 [24], 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. Type enforcement has also been integrated into at least one Internet firewall product, the SCC Sidewinder[5] system [23], 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)[13] provide greater flexibility in UNIX discretionary access controls, and user-mode capabilities[16] 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 [20]. These systems typically provide MLS security but lack the flexibility of DTE. Additionally, tools such as COPS [12] check

[5] Sidewinder is a trademark of Secure Computing Corporation, Inc.

for system miscofigurations 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.

7 Future Directions

We are actively exploring several directions for DTE. The most immediate and important one is the integration of DTE into Internet firewalls. Over the next two years, we will integrate DTE into firewalls in three phases:

DTE Firewalls An integration of DTE into an Internet firewall and selected hosts. This integration will 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 contain any malicious effects. Our primary goal here is to allow more network services to be safely imported into a LAN than is now prudent.

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 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.

In order to accomplish these goals, we will soon begin investigating 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.

8 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 recent extensions to DTE to provide greater security for 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 sum, 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 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

[1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, S. A. Haghighat, "Practical Domain and Type Enforcement for UNIX," 1995 IEEE Symposium on Security and Privacy, Oakland CA, May 1995.

[2] L. Badger, "A Model for Specifying Multi-Granularity Integrity Policies," 1989 IEEE Symposium on Security and Privacy, p. 269, Oakland, CA, May 1989.

[3] R.W. Baldwin, "Naming and Grouping Privileges to Simplify Security Management in Large Databases," Proceedings of the 1990 IEEE Symposium on Security and Privacy, p. 116, Oakland, CA, May 1990.

[4] D.E. Bell and L. Lapadula, "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).

[5] K.P. Birman, T. Joseph, K. Kane, F. Schmuck, "The ISIS Programming Manual and User's Guide," Department of Computer Science, Cornell University, June 1988.

[6] K.J. Biba, "Integrity Considerations for Secure Computer Systems," USAF Electronic Systems Division, Bedford, MA, ESD-TR-76-372, 1977.

[7] M. Branstad, H. Tajalli, F. Mayer, D. Dalva, "Access Mediation in a Message Passing Kernel," 1989 IEEE Symposium on Security and Privacy, p. 66, Oakland, CA, May 1989.

[8] 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, Oakland, CA, p. 184, 1987.

[9] W.E. Boebert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD, p. 18, 1985.

[10] J. Ioannidis, M. Blaze, "The Architecture and Implementation of Network-Layer Security Under Unix," Presented at the USENIX Summer 1994 Technical Conference, Boston MA.

[11] NBS, "Data Encryption Standard," Jan. 1977. Federal Information Processing Standards Publication 46.

[12] D. Farmer, "The COPS Security Checker System," Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, p. 165.

[13] G. Fernandez, L. Allen, "Extending the UNIX Protection Model with Access Control Lists," Proceedings of the Summer 1988 USENIX Conference, San Francisco, CA, 1988, p. 119.

[14] T. Fine and S. E. Minear, "Assuring Distributed Trusted Mach," 1993 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, p. 206, 1993.

[15] J. Kohl and C. Neuman, "The Kerberos Network Authentication Service (V5)," RFC 1510, September 1993.

[16] D. Klein, "A Capability Based Protection Mechanism Under Unix," Proceedings of the 1985 Winter USENIX Conference, Dallas, Texas, p. 152.

[17] C.E. Landwehr, C.L. Heitmeyer, and J. McLean, "A Security Model for Military Message Systems," ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 198-222.

[18] S.B. Lipner, "Non-Discretionary Controls for Commercial Applications," Proceedings of the 1982 IEEE Symposium on Security and Privacy, Oakland, CA, p. 2, 1982.

[19] M. K. McKusick, "The Virtual Filesystem Interface in 4.4BSD," USENIX Computing Systems, Vol 8, Winter 1995, p. 3.

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

[21] R. O'Brien and C. Rogers. Developing Applications on LOCK. In *Proc. 14th National Computer Security Conference*, pages 147–156, Washington, DC, October 1991.

[22] L.L. Peterson, N.C. Buchholz, R.D. Schlichting, "Preserving and Using Context Information in Interprocess Communication," ACM Transactions on Computer Systems, 7(3):217-246, Aug. 1989.

[23] Secure Computing Corporation, Sidewinder Press Release, October 10, 1994.

[24] D. Sterne, "A TCB Subset for Integrity and Role-Based Access Control," *Proc. 15th National Computer Security Conference*, pages 680–696, Baltimore, MD, 1992.

[25] 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*, Oakland, CA, p. 167, 1989.

[26] D. J. Thomsen, "Role-based Application Design and Enforcement," In *Proc. of the Fourth IFIP Workshop on Database Security*, Halifax, England, September 1990.

[27] S. Wiseman, "A Secure Capability Computer System," Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, CA, p. 86, 1986.