Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD

Robert N M Watson *

rwatson@FreeBSD.org

September 8, 2000

Abstract

Trusted operating systems provide a number of features beyond the standard discretionary access control policies of commercial, off-the-shelf operating systems. These include features such as fine-grained event auditing, least-privilege design, mandatory access control policies, and extensive design documentation. The TrustedBSD project is adding trusted operating system features to FreeBSD, an open source UNIX-like operating system under a liberal license. However, TrustedBSD requires extensive changes to the access control mechanisms in FreeBSD. At this point in the project, we have implemented file system extended attributes for storing security labels on files, revamped internal handling of privilege in the operating systems, and are working on an improved generalized access control system.

1 Introduction

1.1 Trusted Operating Systems

A number of features distinguish trusted operating systems from traditional operating systems: carefully integrated discretionary access control policies, system mandatory access control policies for confidentiality or integrity, fine-grained event auditing, and a least-privilege design. Additionally, trusted operating systems must be extensively documented to describe their access control policy and design, allowing the security features of the operating system to be evaluated. The primary reference for trusted operating systems has traditionally been the Rainbow Series, more specifically the Orange Book [oD85] released by the US Department of Defense. [Pfl96]

Discretionary access control (DAC) refers to the protection specified by the owners for objects they create and manage. In the UNIX context, this generally is limited to permissions on files and IPC objects, such as FIFOs and UNIX Domain Sockets.

Mandatory access control (MAC) refers to system policies about interaction between classes of users and objects, usually to limit the flow of information (confidentiality) or the integrity of information. Traditional policies include the military-style Multi-Level Security policy (MLS), and Biba integrity policies. [BL73] [Bib77]

Auditing of system events refers to the extensive run-time logging of system events, especially those events related to authorization and authentication. These logs can be used to assign responsibility, monitor information flow, as well as to research compromises. More recently, Intrusion Detection Systems (IDS) have also made use of audit logs.

Least-privilege design refers to the principle of least privilege: subjects should have only the minimum privilege required to perform necessary tasks. In the context of applications, this has implications for mandatory access control policies, but in trusted operating systems, it also applies to the ability to violate operating system policy: for example, to override DAC and MAC limits, configure system resources and policy, shut down the system, etc.

* With thanks to Safeport Network Services, NAI Labs at Network Associates, and Berkeley Software Design, Inc. for the contribution of networking, computer hardware, and travel resources for this project.

More recently, the Common Criteria were standardized by ISO to allow for a standard vocabulary and evaluation criteria for trusted systems internationally. Since the release of the Common Criteria, many trusted operating system vendors have begun to target CC protection profiles rather than the Orange Book. [(CC00] [ISSO99a] [ISSO99b]

Documentation also plays in important role in trusted operating systems: to achieve any formal evaluations, extensive and careful documentation of the security design of the software is required. This includes design guides documenting the authorization and authentication policies in great detail, as well as operational guides for trusted facility management.

1.2 The TrustedBSD Project

The TrustedBSD project seeks to introduce these features in the FreeBSD operating system, providing them under a liberal license to encourage adoption of trusted operating system features outside of the traditional, restrictive military environment. Once the code is developed on the FreeBSD platform and in stable condition, there are also tentative plans to consider migrating this support to other platforms, as resources permit. Combining both practical open source, commercial, and research goals, the TrustedBSD project is following both traditional designs for trusted operating systems, as well as improving the security framework in FreeBSD to permit additional research into access control and system policies.

The TrustedBSD project is being implemented in a number of stages, based both on the dependencies between components, the desire to build additional experience in the developer base, and the availability of development resources. In early stages, initial framework and infrastructure improvements are taking place in the base FreeBSD source code, facilitating the introduction of more advanced features. Built on this infrastructure, features described above are added to the system in a first implementation; in a second implementation, abstractions will be improved based on experience gained in the first pass. However, even early in the project, the FreeBSD developer and user communities will have access to system improvements and new features.

- In the first stage, support for security labeling and access control in FreeBSD is improved. This includes the ability to manage and persistently store additional security labels for persistent objects, devolution of the superuser privilege into individually assigned and managed capabilities, and clarifications and consistency improvements in inter-process and file system access control. Additionally, support for discretionary access control is improved through the introduction of fine-grained Access Control Lists (ACLs) on files and directories.

- In the second stage, the kernel and user-land environments are modified to take advantage of improved privilege structuring, allowing reduced privilege to be allocated to tools and daemons that take advantage of privilege.

- In the third stage, initial support for mandatory access control is introduced, building on the improved discretionary access control support, object labeling, and privilege management. Both confidentiality and integrity models will be supported, as well as scalable system partitioning schemes.

- In the fourth stage, auditing of system events will be introduced, both for traditional audit log purposes, and with the intent of introducing a pluggable host IDS interface.

- In the fifth stage, access control facilities will be reworked for improved generalization, with the intent of allowing new security models to be pluggable, permitting easy extension and configuration. Existing access control models and auditing support will be reworked to fit within this framework.

Throughout the project, an emphasis will be placed on cross-platform portability, standards, and documentation.

1.3 Project Status

The TrustedBSD development group has made substantial progress, including participation in the trusted operating system community, interface portability and standardization, and the implementation of features including file system extended attributes, fine-grained privileges through capabilities, access control lists, and improved access control. Initial work has also begun on mandatory access control, and work leveraging past efforts relating to even auditing is in the planning stages. The remainder of this paper is dedicated to describing these new features, their implementation, and portability concerns.

2 Standards and Portability

Trusted operating systems have been active area of research and development since the early 1970s. A large number of trusted operating system products have been produced, and a moderately sized developer community exists, as well as interest in the US Department of Defense, and around the world. In the past, efforts have been made to standardize interfaces to common trusted operating system features, using POSIX [IEE90] as a vehicle for the standards process. Participating in this community is important, as all members of the community have much to gain through common interfaces supporting common applications, which are, contrary to common operating system designer beliefs, the mainstay of real-world computer use. An important aspect to the TrustedBSD project is participating in appropriate forums, and monitoring of other projects to understand their direction, leverage their progress, and assist in their development. As with other areas in operating system development, a healthy combination of competition and cooperation leads to the best results.

2.1 POSIX.1e

The POSIX.1e draft standard [IEE97] from IEEE is the result of a working group within the trusted OS community seeking to standardize API calls for common trusted operating system features. These include auditing, access control lists, fine grained privileges ("capabilities"), information labeling, and mandatory access control. The standardization effort failed for a variety of reasons, but the draft was released by IEEE for public redistribution. The most recent draft version, D17, has been adopted by the TrustedBSD project, as well as others including members of the Linux community. A number of existing commercial operating systems include features closely resembling those in POSIX.1e, including most prominently Trusted IRIX, as well as features in Solaris.

Interest in the POSIX.1e draft standard continues, with discussion of its feature set in a number of forums, including a POSIX.1e mailing list maintained by the author of this paper. While some features are of limited value to many vendors, including the information label specification, other sections of the draft are cogently written and include detailed rationale for design choices, such as the access control list (ACL) and capability mechanisms. Where necessary and desirable, the TrustedBSD project makes use of these features, conforming where possible to the APIs present there. In June, 2000, SGI held a workshop on POSIX.1e capabilities to gather together elements of the Linux community in light of weaknesses discovered in their capabilities implementation. This forum provided a useful vehicle for resolving ambiguities in the text, and understanding the implications for modern UNIX-like operating systems, especially in the context of features unanticipated by the authors.

3 UFS Extended Attributes

The TrustedBSD feature set requires that new meta-data be persistently associated with files and directories including access control lists, capability sets, and mandatory access control labels. To serve this need, extended attributes were added to the Fast File System (FFS) [MBK+96] in FreeBSD as of version 5.0-RELEASE.

Extended attributes permit sets of arbitrary (name, value) pairs to be associated with files and directories in a file system, allowing the file system to support a class of dynamic extensions requiring additional meta-data without changing the on-disk storage format. Support for extended attributes is present in a number of operating systems; in this section, we consider the semantics of extended attributes in various operating systems, the requirements for attributes in TrustedBSD, the implementation that is currently in the FreeBSD 5.0-CURRENT source tree, and its performance implications. As this new feature introduces new API calls, we also consider portability issues and standards efforts in the trusted operating system community.

3.1 Requirements

TrustedBSD functionality places a fairly concise set of requirements for on-disk meta-data storage associated with files and directories. The various features of TrustedBSD associate a fixed set of well-defined labels with file system objects. These labels must be persistently stored across reboots and failures, and the labels must be bound to the same object regardless of renames, linking operations, etc, much in the style of existing file permissions. All labels need not be defined for all files: in the POSIX.1e specification, some files may have normal permission sets rather than complete ACLs. While most TrustedBSD labels have a fixed size, the implementation should support variable size attributes. Labels must be protected from unauthorized inspection and modification, as the integrity and secrecy of these labels can impact system security.

3.2 Semantics

To satisfy these system and application requirements, the semantics for UFS extended attributes are:

- Extended attributes (EAs) on a file system inode are a set of (name, data) pairs. For each inode, an attribute name may be defined or undefined, and if defined, may be associated with zero or more bytes of data. This is similar to environmental variables in common shells.
- This attribute data, unlike the file itself, does not comprise a complete address space accessed through an independent file descriptor: it is not a file fork. Instead, relatively inflexible API calls are provided to set, get, and remove the data associated with a particular named attribute on the file. These operations are atomic for a particular name, meaning that it is not possible for a get operation to return an inconsistent view of a particular attribute: writes and reads of a particular attribute are serializable.
- Protection models may exist protecting particular attributes, on the basis of namespace, or discretionary/mandatory access control policies. This protection mechanism should make it possible for EAs to be used to safely store system attributes such as ACLs, mandatory access control labels, and capabilities. As application writers may want to make use of attributes from user-land, it may be desirable not to preclude this use.
- For the purposes of management, and potentially backup, it is desirable to be able to retrieve a list of attributes defined for a particular inode.

3.3 Interfaces

The extended attribute service is exposed to userland processes through the addition of several new system calls, providing the ability to set, retrieve, and remove extended attributes from files and directories by name. These services are reflected in similar Virtual File System (VFS) vnode operations within the kernel, and a VFS operation used to configure extended attributes in UFS. The interface also supports the retrieval of defined attribute lists for a file via a special case attribute retrieval.

3.3.1 VFS

The Virtual File System (VFS) provides an abstracted file system interface for consumers within the kernel. Vnodes represent files and directories, and are the target for most file system operations. Vnode operations are implemented by the file system associated with the vnode.

```
int
VOP_GETEXTATTR(struct vnode *vp,
               const char *name,
               struct uio *uio,
               struct ucred *ucred,
               struct proc *p);
```

The `VOP_GETEXTATTR()` vnode operation retrieves a specific attribute by name from a locked vnode: the caller provides a `struct uio` to specify the destination within kernel or user memory space. If non-`NULL`, struct cred * identifies the credentials authorizing the attribute request; otherwise, the request is assumed to have originated within the kernel. The name argument points to the attribute name to be retrieved; if NULL, a list of defined attributes for the vnode will be returned within the data block as a series of NULL-terminated strings, followed by a zero-length NULL-terminated string. This list may be limited to the list of attributes accessible to the caller. On success, 0 is returned; otherwise, an errno value is returned identifying the failure mode.

```
int
VOP_SETEXTATTR(struct vnode *vp
               const char *name,
               struct uio *uio,
               struct cred *cred,
               struct proc *p);
```

The VOP_SETEXTATTR() vnode operation sets a specific attribute by name for a locked vnode: the caller provides a struct uio to specify the source within kernel or user memory space. If non-NULL, struct cred * identifies the credentials authorizing the attribute request; otherwise, the request is assumed to have originated within the kernel. The name argument points to the attribute name to be set. On success, 0 is returned; otherwise, an errno value is returned identifying the failure mode.

```
int
VFS_EXTATTRCTL(struct mount *m,
               int cmd,
               const char *attrname,
               caddr_t arg,
               struct proc *p);
```

The VFS_EXTATTRCTL() VFS operation is UFS-specific, and is used to configure the extended attribute service on a UFS file system. The cmd argument specifies the management request:

cmd	Description
UFS_EXTATTR_CMD_START	Start extended attributes on the file system
UFS_EXTATTR_CMD_STOP	Stop extended attributes on the file system
UFS_EXTATTR_CMD_ENABLE	Enable a specific named attribute
UFS_EXTATTR_CMD_DISABLE	Disable a specific named attribute

In each case, m identifies the file system on which the operation is to be performed; attrname identifies the attribute, if any, impacted by the call. The arg argument points to call-specific information; in the case of UFS_EXTATTR_CMD_ENABLE and UFS_EXTATTR_CMD_DISABLE it points to add additional pathname identifying the extended attribute backing file to use. This call is UFS-specific, and non-portable.

3.3.2 System Calls

System calls allow userland processes to make requests for kernel services; for the extended attribute implementation, all of the userland system calls map directly into a corresponding vnode or file system VFS operation, with almost identical semantics.

```
int
extattr_get_file(const char *path,
                 const char *attrname,
                 struct iovec *iovp,
                 unsigned iovcnt);
```

The extattr_get_file() system call maps into VOP_GETEXTATTR(), accepting a pathname path, attribute name attrname, and struct iovec *iovp to scatter-gather read attribute contents into userland buffers. If the attribute name is NULL, reads a list of attributes that may be accessible to the caller.

```
int
extattr_set_file(const char *path,
                 const char *attrname,
                 struct iovec *iovp,
                 unsigned iovcnt);
```

The `extattr_set_file()` system call maps into `VOP_SETEXTATTR()`, accepting a pathname, `path`, attribute name `attrname`, and `struct iovec *iovp` to scatter-gather write attribute contents from userland buffers.

```
int
extattr_delete_file(const char *path,
                    const char *attrname);
```

The `extattr_delete_file()` system call maps into `VOP_SETEXTATTR()` with a `NULL uio` argument, causing the attribute `attrname` on file `path` to be deleted.

```
int
extattrctl(const char *path,
           int cmd,
           const char *attrname,
           char *arg);
```

The `extattrctl()` system call maps directly to `VFS_EXTATTRCTL()`, and permits an appropriately privileged caller to configure extended attributes on the UFS file system identified by `path`.

3.4 Existing Implementations

Before implementing extended attributes in TrustedBSD, it was necessary to consider implementations on other operating systems, both to understand possible approaches the the problem, and to understand their semantics from the perspective of portable application interfaces. A variety of operating systems implement mechanisms with semantics like those described for extended attributes, binding additional information to files and directories, beyond the normal file address space and static attributes. These implementations are generally split into two categories: extended attributes, and file forks.

Extended attributes generally fit the semantics described here, with minor variations in the details of the API calls, and a fairly broad variation in underlying backing store. EAs are implemented in two forms on the IRIX platform–first, in a form where-in each attribute name is backed to a file in the file system, treated as an indexed array of data by inode. This technique is referred to as "Plan G," and was used in Trusted IRIX to store MAC labels and other security attributes. In more recent versions of IRIX supporting the XFS file system, EAs are built into the meta-data supported by the base file system format, and handled as part of the journalling support of the file system, while maintaining many of the same semantics.

Support for extended attributes in Linux has been developed as a set of patches, based on earlier work implementing Access Control Lists. In the Linux implementation, a direct block pointer in the inode is allocated to pointed the an extended attribute block, which stores a set of variable length attribute records. This block is not currently reference counted, nor is there a way to chain additional blocks, bounding the size of all attributes on an inode to the block size of the file system.

The IBM HPFS file system also provides support for EAs as part of its fnode structure; the first 316 bytes will be stored in the fnode of the file; after this, EAs are accessed in a file system run via a B+ Tree. The total size of all EAs is bounded to 64k. [Dun89]

File forks represent the extreme in association additional names and data with a file system object: a complete additional address space. A number of file systems provide file forks, with varying semantics; some provide additional protection for each forks, others rely on a single protection model for all forks. In many of these environments, the file fork is accessed through an extension to the normal file system namespace, allowing a separate file handle to be associated with access to each fork, and all of the normal file system operations to be performed on the fork.

The Solaris file system does not provide a generic EA service, but does allow for the storage of ACLs on file system objects via a "shadow inode," a second inode linked to the first by an inode pointer in the

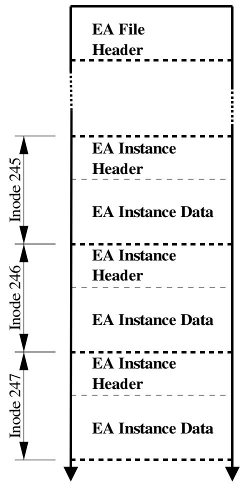

Figure 1: Extended Attribute Backing File Layout

primary inode. This inode is reference counted, and may be referred to by multiple files with the same ACL. While the API resembles more an extended attribute, the underlying mechanism resembles more a file fork, in that it may potentially support far more broad semantics. Both NTFS and HFS+ support complete file fork address spaces, returning file handles with all normal file access methods available.

3.5 Design and Implementation in TrustedBSD

The TrustedBSD extended attribute implementation was developed rapidly, and with a number of specific goals in mind. First, it must cleanly provide a service consistent with the portable interfaces defined in previous sections. Second, it should provide acceptable levels of performance and consistency for trusted operating system environments, while not precluding the development of faster implementations, or those with stronger consistency guarantees. Third, the implementation should be straight forward and quick to implement, so that other new services could leverage the attribute service. Fourth, for the sake of an initial implementation and ease of use, the implementation should not require substantial (if any) changes to the underlying file store format, or low level management tools, such as the file system checker.

These goals provide for an implementation in the short term that meets application and system requirements, while providing a low overhead means to test and use TrustedBSD services without heavy-weight system modifications or reconfiguration. The implementation is also tailored to the requirements of TrustedBSD: a relatively small set of well-known attributes, generally of a fixed maximum size per file or directory.

3.5.1 Storage

The UFS EA service, in a manner similar to that in Trusted IRIX Plan G and to the UFS Quota service, backs extended attribute data into a per-attribute, per-fs backing file. The backing file acts as an array of attribute data blocks, indexed by the inode number of the file each "attribute instance" is associated with. The file is prefixed by a per-attribute header, describing the contents of the file–specifically, the version of the file header, and maximum length of the attribute data allowing offsets into the file to be determined. Each attribute instance is also prefixed by a header storing its current status (defined or undefined), the generation number of the inode it is associated with for sanity checking, as well as its current length (0 or more bytes, up to the file header-defined maximum). This layout is illustrated in Figure 1.

The inode generation number allows the implementation to determine whether the inode has been freed and re-allocated while the attribute was disabled; if the generation number is out of sync, the attribute is treated as undefined. This might occur if the file was lost and deleted by `fsck` during the boot process prior to attributes being enabled. This layout results in the header of a particular inode's attribute instance being located at the following offset of the backing file:

```
/*
 * Find base offset of header in file based on file header
 * size, and data header size + maximum data size, indexed
 * by inode number
 */
base_offset = sizeof(struct ufs_extattr_fileheader) +
    ip->i_number * (sizeof(struct ufs_extattr_header) +
    attribute->uele_fileheader.uef_size);
```

This allows constant-time access to the attribute of each file or directory based on a priori knowledge (inode number) of the target. Depending on the requirements on the attribute, the backing file may start out as a sparse file, or it may be preallocated. Sparse allocation for larger attributes substantially conserves space, with the risk that an attribute write may fail due to a lack of disk space. Preallocation can be used to guarantee that the space is available, at a cost of space that may otherwise be unused. In practice, for small fixed size attributes, such as capabilities and MAC labels, this is not substantial wastage. For capabilities, where only relatively few binaries have a defined 24-byte label, this will amount to only .5of disk space on an average install. This space wastage can be reduced by making use of a sparse backing file, with the understanding that running out of disk storage can result in a failure to write out attributes; fine for static capability storage, but less appropriate for ACLs which may be allocated frequently.

The extended attribute implementation currently provides only weak consistency guarantees: in the default implementation, writes are performed synchronously, to guarantee to the caller that updates are written out before the call returns. As shown later in the section, this design choice can substantially impact performance for frequently modified attributes. The risk of inconsistencies can be reduced by selecting attribute sizes that are smaller than a single file system block, allowing the update to be atomic. Potential remedies to this problem are discussed later in the section.

Backing files are typically stored in the ".attribute" directory off of each file system root, named after the attribute they store. Backing files may also be stored on other file systems.

3.5.2 Initialization

Extended attributes are intended to be enabled from mount-time for a file system. The `VFS_EXTATTRCTL()` VFS call and `extattrctl()` system call provide an interface for privileged users to map a particular attribute backing file to an attribute name. As each attribute is stored in ".attribute/attributename" off of the file system root, it is possibly to easily automate the loading of extended attributes on a per-file system basis.

One down-side to this approach to attribute storage is that privilege is required to create new attribute names in the system; while this is appropriate in the TrustedBSD environment, it may be less so in environments where application programs expect to be able to make use of arbitrary attributes for file meta-data. For example, in SGI's XFS file system, applications can create new attributes in the application namespace without intervention by an administrator, and frequently do so, storing information such as file icons for file managers in these attributes.

3.5.3 Protection Model

The protection model selected was one similar to the XFS and Linux models: two distinct EA namespaces are provided, "application" and "system". System attributes are identified by a "$" symbol prefixing the attribute name; only users with appropriate privilege are permitted to directly modify or read attributes in this namespace. All other attribute names are interpreted as existing in the application namespace, and are subject to the discretionary and mandatory protections on the target file or directory, permitting the owner of the file system object to protect the attribute data as needed.

When applications need to modify the contents of a system extended attribute, such as the access ACL for a file, they are expected to make use of existing system calls for those file attributes. This permits the file system to hide whether or not the implementation of these labeling features is provided by the file system itself, or through the EA service. This is distinct from the Linux implementation, where the underlying implementation for application services retrieving ACLs on files makes use of the system attribute directly. As such, mapping of labels into extended attributes occurs in the supporting userland libraries in Linux, as well as in the kernel.

3.5.4 Performance

The majority of the performance impact of adding trusted operating system features to an off-the-shelf UNIX-like operating system lies in the management of persistent labels. As such, understanding the performance implications of the extended attribute implementation helps explain any reduced performance as a result of other features. In this section, we analyze the performance impact of the extended attributes implementation, as well as the cost of varying consistency requirements. In a later section, will examine the impact of the EA implementation in the context of a specific consumer, the capability label.

For the purposes of performance measurement, we consider a set of micro-benchmarks measuring the latency of extended attribute access on an system under low load. As there are two components to each attribute, an attribute instance header storing attribute instance meta-data, and the attribute data itself, we perform two sets of measurements: first, measurements on an extended attribute with a data size of zero bytes, and second, on an extended attribute with a data size of twenty-four bytes, the size of a capability structure (described later in this paper). An additional variable in the reading of the extended attribute is whether or not it was defined–we refer to the situation in which the caller requests an EA and it is defined as a "read hit"; when the EA is not defined and an error is returned, it is defined as a "read miss". An additional variable for extended attribute reading is whether or not the attribute was actually defined, and therefore might require a read of attribute data as well as the attribute header. We are also interested in comparing the costs of cached access to extended attribute data vs. cached access to traditional attribute data (such as inode time stamps and file mode).

To this end, the following matrix of tests was performed. Each test is repeated ten times, with the first iteration preceded by a flushing of the file system cache. Each iteration consists of the operation on a single file, preceded by an untimed `stat()` system call to assure that the name and inode are in the cache.

Name	Description
stat	Latency to `stat` a file
utimes	Latency to modify the timestamps on a file
openclose	Latency to `open` and `close` a file
read-miss	Latency of a "read miss" on an attribute
read-hit	Latency of a "read hit" on an attribute
write-sync	Latency of a synchronous write on an attribute

These performance tests were performed on an E-Machine 366i2, based on an Intel Celeron 366 chip, with 64mb of RAM, running FreeBSD 5.0-CURRENT with POSIX.1e capability patches. The timing of tests was measured using the `clock_gettime()` function; in this environment `clock_getres()` returns a clock resolution of 0.000000838 seconds. In all cases, results were several orders of magnitude more coarse than the clock resolution. No extraneous processes ran during the tests, and a reboot was performed between each test to restore system state and flush all file system caches.

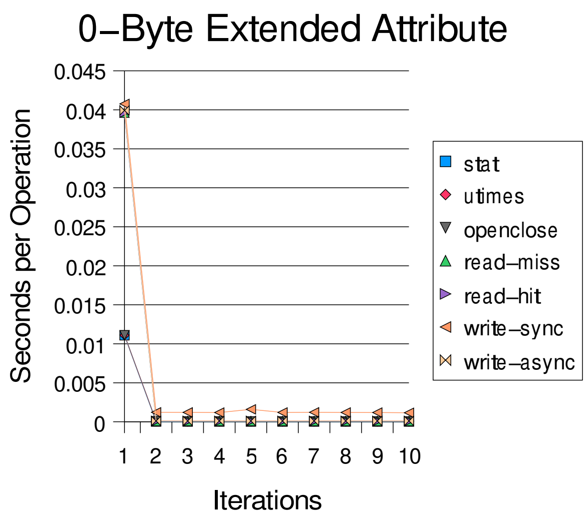

Figure 2: 0-byte Extended Attribute, with synchronous writes

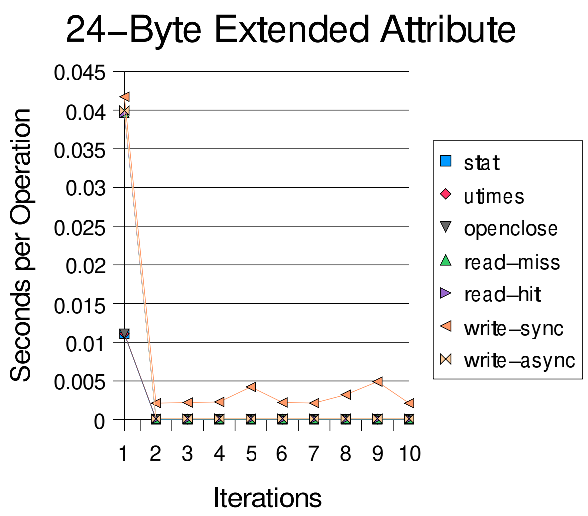

Figure 3: 24-byte Extended Attribute, with synchronous writes

From Figure 2 and Figure 3, it is clear that the initial latency associated with loading the extended attribute from disk is the predominant factor in the pre-cache first iteration, at approximately 40 milliseconds, four times as slow as the baseline `stat()`, `utimes()`, and `openclose()` figures, all around 11 milliseconds. Reads and writes will both involve first reading in any supporting meta-data for the file, including indirect pointer blocks, and the data blocks themselves: writes will in general be modifications of an existing block, or an allocation of a new block, so a write cannot un-block until appropriate data and meta-data are available to determine how the write will occur, and to allow it to be buffered. At this resolution, later iterations see all extended attributes operations approach that baseline except for the synchronous extended attribute write. In the 0-byte extended attribute, the performance of the write is fairly consistent, whereas with the 24-byte attribute, it is far less predictable.

The slow write performance past the initial iteration, once data is in the cache, is due to the `IO_SYNC` flags specified as an argument to the `VOP_WRITE()` calls in the extended attribute implementation– one invocation of `VOP_WRITE()` when only the header is written out, and two when actual data is written. As the security labels are sensitive to file system consistency, synchronous writing guarantees that the label change is written to disk before the attribute write returns. This is a realistic safety policy with unfortunate performance side effects; however, there has been extensive research conducted on maintaining file system meta-data consistency while avoiding the cost of fully synchronous operations. In FreeBSD, the Soft Updates technique [MG99] is used to order and write-combine cached meta-data updates to always maintain a consistent and recoverable file system on disk, resulting in equal or better performance relative to asynchronous operation. It is easy to imagine that such techniques could also be applied to extended attribute meta-data, allowing us to compare the performance of extended attribute writes using asynchronous writes. The write-async graph line, therefore, represents the same write operations with the `IO_SYNC` flag removed. At this resolution, the asynchronous write case appears to perform at about the same level as the extended attribute reads. Therefore, to better understand the performance properties once the data is in the cache, we consider the same data at a different resolution, leaving out the synchronous file write case, and the initial load from the disk into cache.

Name	Description
write-async	Latency of asynchronous write on an attribute

In the next two graphs, Figure 4 and Figure 5, the same results are presented, with improved resolution. When cached, the performance of all of the tests is fairly consistent: at approximately 26 microseconds per access, the `stat()` vnode call represents a safe approximation of the minimum cost for VFS traversal and locking to access the inode statistics. At 50 microseconds, `openclose()` reflects the same cost: two locking vnode operations. In a best-case scenario, an extended attribute implementation would not be able to out-perform these cases, as the EA implementation will see increased cost over retrieving data in the inode, and scheduling an inode write, while still paying the same vnode operation costs. The `utimes()` vnode operation, at around 87 microseconds per invocation, represents the minimum time to perform a meta-data update on a UFS inode, scheduling a write or a softupdate. Both the 0-byte and 24-byte extended attribute read miss requests perform better than the `utimes` operation, as does the read hit operation at 78 microseconds for the 0-byte attribute and 81 microseconds for the 24-byte attribute. If the cost of name lookup and VFS operations is subtracted from the un-cached case, the actual cost of the EA disk operations is around 28.5 milliseconds for all operations. The asynchronous extended attribute write hovers at 112 microseconds for the 0-byte attribute, and 118 microseconds for the 24-byte attribute. However, both of these compare quite well with the 1.24 millisecond synchronous write for the 0-byte attribute, and the 2.8 millisecond synchronous write for the 24-byte attribute. Given that a normal inode attribute update, `utimes()` takes 90 microseconds to complete in cache, the cached behavior of extended attribute writes when asynchronous closely approximates that of regular attributes; assuming that the asynchronous write operation has to go through the same procedures as `utimes()`, this places the cached overhead of an EA write at 25 microseconds, around the cost of a single `stat()` operation. These results suggest that this extended attributes implementation, when caching is effective, can approximate the performance of native vnode statistics operations. In situations where worst-case disk accesses must be made without caching, a slow-down can be experienced; however, when overlapping elements of the EA and non-EA tests are removed, this cost is not substantial; in the context of most TrustedBSD EA applications, the name and vnode operations on the target vnode will already be cached, meaning that in these applications the smaller cost will be paid most often. To understand these costs in the context of a real-world workload, benchmarks later in this paper consider the cost of a 24-byte EA capability load at exec()-time.

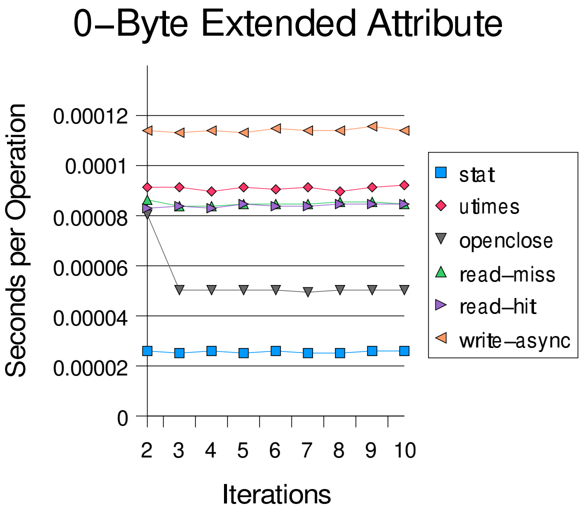

Figure 4: 0-byte Extended Attribute, without synchronous writes

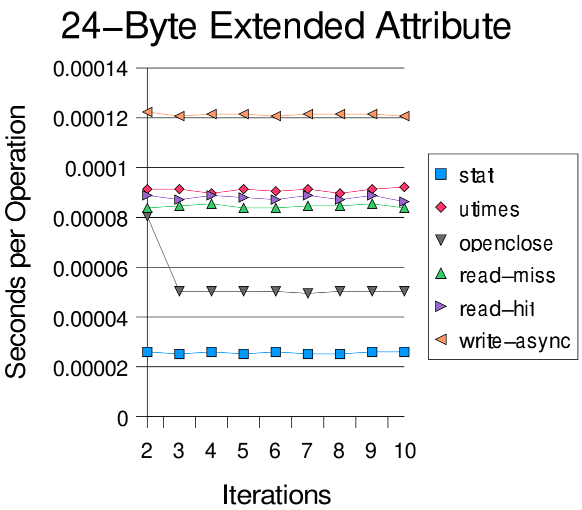

Figure 5: 24-byte Extended Attribute, without synchronous writes

These performance tests did not attempt to measure the performance costs of concurrent access on extended attributes. Given that the implementation involves coarse-granularity locking on the extended attribute file, there is a strongly likelihood that frequently requested, un-cached extended attributes would result in contention on the attribute's lock. However, as vnode locks are required to be held on a vnode before a `VOP_WRITE()` or `VOP_READ()` operation can take place, this contention may be unavoidable by virtue of using a single backing file.

3.5.5 Potential Improvements

This implementation makes no attempt to reduce storage space overhead through reference counting and singular storage of redundant data. For example, ACLs over a set of files are often identical, either by virtue of the ACL inheritance property on newly created directory entries, or as a result of management requirements for the files. Some implementations, such as the older Linux ACL implementation, and the shipped Solaris ACL implementation, reference count ACL data blocks, allowing them to be shared copy-on-write among inodes. A number of schemes exist that could make this feasible in the current implementation, including probabilistic techniques for detecting redundant data, and introducing a layer of indirection.

Performance could be improved in a number of ways. The double `VOP_WRITE()` call in synchronous operation substantially impacts performance; combining the two writes into a single invocation would improve performance with no loss in functionality. Performance could also be improved by reducing reliance on the file system cache to store in-memory extended attributes. Caching of attributes would benefit from being tied to in-memory inode and vnode data, attempting to match the in-memory storage of attributes with their respective inode/vnode data. Similarly, false sharing could be reduced, and both locational and temporal locality could be leveraged.

Concurrent performance would also be improved via improvements to the locking model: currently, a single lock exists per-attribute, per-file system. I.e., a single lock is present for an `$acl_access` attribute instance on the /usr file system, disallowing concurrent access to multiple instances of the attribute at a particular moment, in effect serializing reads and writes to that attribute over all files in the file system, rather than over a single file. By pushing down lock granularity, preferably relying on vnode locks to protect against concurrent access to attributes on a particular vnode, performance for frequently accessed and hence highly contentious attributes (such as ACLs) might be improved.

The backing file mechanism allowed for fast implementation, and acceptable performance. However, it imposes both performance and management limits on the use of extended attributes: storage space is wasted, and not located optimally on the disk for simultaneous reading of associated inodes, and the initialization model for new EAs is too restrictive if they are to be used by applications, rather than allocated by administrators. These problems, as well as consistency concerns relating to system failure and crash recovery, suggest tighter integration with the file store, and with the soft updates consistency mechanisms.

3.5.6 Portability

Substantial effort has been made to come to agreement on common extended attribute service semantics across the IRIX, Linux, and FreeBSD platforms, via the POSIX.1e forum, as well as in-person meetings and private communication. In general, the semantics are agreed upon: the atomic replacement, environmental variable-like model seems to be widely accepted as the appropriate semantic for EA access and modification. Similarly, the protection model relating to application and system namespaces seems well-accepted. The IRIX EA implementation provides a namespace argument in the API calls in the form of a flag, whereas both FreeBSD and Linux rely on an in-band "$" symbol in the attribute name to indicate a system attribute. Actual API calls are still under negotiation, as are command line utilities, but it is expected that if not identical, the calls will be very similar. In the mean time, interfaces on Linux and FreeBSD should be considered experimental.

4 Process Capabilities

UNIX systems impose a system safety policy, isolating user processes based on credentials, limiting access to files based on file permissions, and protecting system resources from inappropriate access or modification.

Typically, the root user is permitted to selectively violate these policies based on holding an effective uid of 0: root can kill any process, read or write any file in the file system, and configure system resources such as network interfaces and hardware devices. This concentration of privilege in the uid 0 account represents a substantial risk by violating the principle of least privilege: that subjects be given only the privileges they require to complete a job. With the uid 0 policy, even simple applications such as ping, which requires only the ability to bind a raw socket, are given complete ability to violate any system policy. Bugs in such applications often leak privileges that they do not even require to function.

4.1 POSIX.1e Capabilities Model

The POSIX.1e draft specification from IEEE describes an alternative framework for the management of special privilege in a UNIX-like environment. Rather than relying on a simple check of a processes effective uid to determine privilege, each process is associated with a set of capabilities. Possessing a capability permits the process to violate a specific part of the system protection policy when required, allowing a least privilege policy to be implemented. In the superuser model, privileges could become elevated during the execution of setuid and setgid programs, gaining additional credentials; a setuid-root program allows the process to attain high levels of privilege. In the capabilities model, it is similarly possible to bind privileges to files, but in a fine-grained manner. This permits executables to gain specific privileges required for their functioning, but unlike the setuid model, does not wholesale grant unnecessary privileges.

Each capability is associated with several flags, determining the availability and inheritance of that capability. The `CAP_PERMITTED` flag indicates whether or not the process is permitted to make use of the capability. The `CAP_EFFECTIVE` flag determines whether or not the capability is currently enabled for the process, allowing the process to selectively enable the capability only when required, limiting the scope for damage. The `CAP_INHERITABLE` flag allows a process to determine whether or not binaries executed by the process will inherit the capability, allowing a privileged process to bound the privilege of another program it executes.

When a capability is bound to a file, these three flags are also present, indicating what rights a process may gain through executing the binary, whether or not they are effective by default, and determining in what situations capabilities will be inherited from the parent, and what the inheritance flags of the process should be set to by default when the program is executing.

In the POSIX.1e model, capabilities for a process are determined using the following formulas at exec()-time:

$$pI' = pI$$

$$pP' = (fP \& X) | (fI \& pI)$$

$$pE' = (fE \& pP')$$

pI, pI'	Process inheritable flags prior to and following `exec()`
pP, pP'	Process permitted flags prior to and following `exec()`
pE, pE'	Process effective flags prior to and following `exec()`
X	Undefined global bounding set

The choice of capabilities in TrustedBSD is largely derived from portability concerns, although generally speaking, the set of capabilities in existing capability implementations matches the requirements for TrustedBSD. It should be observed that in common UNIX implementations, a number of the capabilities provide for overlapping or equivalent functionality. For example, the POSIX.1e `CAP_FOWNER`, `CAP_SETFCAP`, and `CAP_SETUID` flags can be leveraged to provide identical privilege. However, the selection of capabilities reflects the type of privilege required, and that these privileges may not be equivalent in all implementations. For example, in environments where loadable kernel modules have safety checks and properties, such as the SPIN/Modula-3 [Ber95] environment, the ability to modify the kernel at run-time does not connote all privilege in the system. That said, these ambiguous capabilities should be managed with care.

4.2 Capabilities Bound Extension

POSIX.1e draft 17 defines the basic `CAP_EFFECTIVE`, `CAP_INHERITABLE`, and `CAP_PERMITTED` flags. In the TrustedBSD implementation, a per-process inherited privilege bound is also available. This may be used to provide services in the style of the `jail()` system call, permanently limiting the capabilities acquired by any future child process of the current process. This feature is not described in POSIX.1e, but has been actively discussed on the POSIX.1e mailing list, and in other related forums.

Inherited capability bounds are implemented as an additional `u_int64_t` reflecting a mask of capabilities permitted for a process, and may be set using the non-portable `cap_set_proc_mask()` system call. This bound may be set only by processes with the `CAP_SETPCAP` capability effective for the process; even with `CAP_SETPCAP`, the new bounding set must be a superset of the old bounding set, permitting some forms of nested bounding. This additional bit-mask takes effect during the inheritance procedure following a call to `exec()`, where the bound limits the capabilities acquirable for the process by executing the binary. If the process setting up the bound appropriate configures its own capabilities, there is no path by which the capability set can be expanded beyond those permitted by the bound. The capability bound, pB is inserted into the inheritance equation to replace the undefined X, representing the starting bound prior to the invocation of `exec()`.

$$X = pB$$

In the Linux implementation, a slightly different substitution is used, providing support for a per file-system bound on capabilities gained, a property of the mount options. As with the `nosuid` mount flag, this allows the file system-wide disabling of capability features. Such a feature is under consideration for the TrustedBSD implementation, also:

$$X = (fB \& pB)$$

One implication of inherited capability bounding is that that programs will be less likely to correctly predict the privileges they will end up running with on the basis of being correctly installed in the file system. This raises the question of whether or not a binary should be executed if its resulting effective capability set is not that predicted by the base rules or described specifically by its permitted set: i.e., that the program requires a capability that was not permitted within the jail. Two failure modes are easy to imagine: first, that the application is responsible for determining if it gained all capabilities it required, which is possible through the `cap_get_proc()` system call. Second, the invocation of `exec()` might be blocked with `EPERM`. In the current implementation, the application is responsible for making the determination of whether or not it has sufficient privilege to perform the operation; this lets an application with multiple privileges maintain some functionality, at the cost of being aware that some privileges it expects might not be present. In a `jail()`, for example, this might allow an administrative tool only to inspect settings, rather than modify them, returning a permission error as appropriate.

4.3 Mapping Privileges into Capabilities

POSIX.1e defines a base set of capabilities, each associated with the violation of a specific system policy associated with a POSIX call. In essence, the POSIX.1e capabilities allow the term "appropriate privilege" throughout the POSIX specifications to be replaced with a specific capability. For example, the `CAP_CHOWN` capability permits a process to violate the restriction that users may not change the ownership on files– this privilege is restricted to the superuser in BSD environments. Some of the POSIX.1e capabilities are associated with other POSIX.1e features: mandatory access control, auditing, and capability management itself.

The Linux POSIX.1e implementation defines a set of additional capabilities, associated with sets of calls not defined in POSIX: calls related to the sockets interface, as well as calls loading kernel modules, configuring aspects of the kernel, and relating to System V IPC calls; similarly, each capability enables the process to violate system policies relating to specific system calls.

The TrustedBSD capabilities implementation makes use of most of the POSIX.1e capability definitions, with slight modifications where additional entry points and related functions exist to those described in

POSIX. Also, TrustedBSD makes use of a number of Linux POSIX.1e extensions, some with identical semantics, and others with similar semantics.

The following table lists the capabilities supported in TrustedBSD, where the definition of the capability came from, and whether or not the TrustedBSD implementation of that capability has substantial differences from the implementation in other operating systems, which developers using capabilities should be aware of.

Capability Name	Portability	Description
CAP_CHOWN	POSIX.1e	Override superuser restriction on chown()
CAP_DAC_EXECUTE	POSIX.1e	Override DAC/MAC restrictions on executing a file.
CAP_DAC_WRITE	POSIX.1e	Override DAC/MAC restrictions on writing to a file.
CAP_DAC_READ_SEARCH	POSIX.1e	Override DAC/MAC restrictions on reading a file/directory.
CAP_FOWNER	POSIX.1e	Override checks requiring the caller to be the file or directory owner.
CAP_FSETID	POSIX.1e	Override requirement that only the file owner set the setuid bit, that the owner must be a member of the file group set the setgid bid, and that the setuid/setgid bits be removed when the file is modified.
CAP_KILL	POSIX.1e	Override restrictions on delivering signals to processes.
CAP_LINK_DIR	POSIX.1e	Override directory linking restriction (not implemented in TrustedBSD.)
CAP_SETFCAP	POSIX.1e	Override restriction that a process cannot set capability state of a file
CAP_SETGID	POSIX.1e	Allow process to change process gids (real, saved, effective, ...)
CAP_SETUID	POSIX.1e	Allow process to change process uids (real, saved, effective, ...)
CAP_MAC_DOWNGRADE	POSIX.1e	Allow process to downgrade MAC label of a file
CAP_MAC_READ	POSIX.1e	Allow process to override MAC restrictions on reading a file
CAP_MAC_RELABEL_SUBJ	POSIX.1e	Override restriction that a process may not modify its own MAC label
CAP_MAC_UPGRADE	POSIX.1e	Allow process to upgrade MAC label of a file
CAP_MAC_WRITE	POSIX.1e	Allow process to override MAC restrictions on writing a file
CAP_AUDIT_CONTROL	POSIX.1e	Allow process to modify audit control parameters
CAP_AUDIT_WRITE	POSIX.1e	Allow process to write data to system audit trail

Capability Name	Portability	Description
CAP_SETPCAP	Linux, TrustedBSD	Override restrictions on how a process sets its capabilities
CAP_LINUX_IMMUTABLE	Linux	Allow modification of S_IMMUTABLE and S_APPEND file attributes
CAP_SYS_SETFFLAG	TrustedBSD	Allow modification of system file attributes
CAP_NET_BIND_SERVICE	Linux, TrustedBSD	Allow binding of privileged network ports
CAP_NET_BROADCAST	Linux, TrustedBSD	Allow broadcasting, listening to multicast requiring privilege
CAP_NET_ADMIN	Linux, TrustedBSD	Interface, firewall, socket debugging, routing, multicasting privileges.
CAP_NET_RAW	Linux, TrustedBSD	Access to raw sockets.
CAP_IPC_LOCK	Linux, TrustedBSD	Allow memory locking.
CAP_IPC_OWNER	Linux, TrustedBSD	Allow SysV IPC ownership check overrides.
CAP_SYS_MODULE	Linux, TrustedBSD	Allow kernel module management.
CAP_SYS_RAWIO	Linux, TrustedBSD	Allow direct device access.
CAP_SYS_CHROOT	Linux, TrustedBSD	Permit calls to `chroot()` and `jail()`
CAP_SYS_PTRACE	Linux, TrustedBSD	Allow debugging of any process.
CAP_SYS_PACCT	Linux, TrustedBSD	Allow management of accounting system.
CAP_SYS_ADMIN	Linux, TrustedBSD	Catch-all for other privileges.
CAP_SYS_BOOT	Linux, TrustedBSD	Allow invocation of `boot()`.
CAP_SYS_NICE	Linux, TrustedBSD	Allow modification of scheduling parameters for all processes, use of real time scheduling.
CAP_SYS_RESOURCE	Linux, TrustedBSD	Allow violation of resources restrictions.
CAP_SYS_TIME	Linux, TrustedBSD	Allow modification of system time.
CAP_SYS_TTY_CONFIG	Linux, TrustedBSD	Allow privileged configuration of tty devices.
CAP_MKNOD	Linux, TrustedBSD	Allow device node creation.

4.4 Integrating Privilege Models

With the introduction of POSIX.1e capabilities, there are now two types of special privilege allotted to processes: rights accrued through the `suser()` call, which returns a success if the effective uid is 0, and rights gained through the effective capabilities of the process. There are a number of ways in which these two privilege policies may be composed, resulting in a variety of possible end policies. However, the composition choice must be made carefully, or the results may be hard to predict, resulting in potential weaknesses.

In SGI's Trusted IRIX product, there are several run-time compositions possible: `CAP_SYS_SUPERUSER`, in which either capabilities or superuser privilege may be used to grant a request, `CAP_SYS_NO_SUPERUSER` in which only capabilities may be used, and `CAP_SYS_DISABLED`, in which only superuser is used (capabilities are effectively disabled). This scope of choices allows the system to provide both a traditional UNIX environment with superuser, that environment with the benefits of capabilities so as to reduce the number of processes running with superuser privilege, and a highly secured environment in which the superuser simply does not exist for the purposes of privileged access. To properly effect a system without a superuser, mandatory access control must also be present to protect the integrity of the trusted code base; Trusted IRIX does this through a Biba integrity model.

In the Linux capabilities implementation, an alternative integration was selected: the superuser privilege would be emulated using capabilities. When a process gains uid 0, (at boot, or through the setuid bit on a root-owned binary), it is provided with all capabilities. The process may selectively drop these capabilities, and cannot normally regain them. This emulation model has a number of interesting properties, one of which is the ability to gain some of the advantages of a least-privilege environment without having capabilities associated with files, reflecting limitations in the Linux file system. However, this integration has a number of limitations, some due to the lack of a mandatory integrity policy in the operating system, allowing root-owned processes that are subverted gain to additional privileges by virtue of discretionary access control protection in the file system. There are also a number of implications for applications that assume that uid 0 denotes privilege, yet inherit only a subset of that privilege, a limitation in all capability environments, but particularly telling when the model for capabilities is subtraction of rights, rather than addition. The recent sendmail/kernel interaction bug in Linux was a result of these unexpected consequences [Sec00]. In the SGI capabilities workshop, these limitations were discussed, and there is the possibility that Linux may migrate to the safer SGI model.

In the TrustedBSD capabilities implementation, SGI-like model is chosen: rather than integrating the capabilities and superuser models, rights can be gained from the administrators choice of one or both of the two models. This boolean composition of the two policies has the property that it is far easier to reason about: when modifications were made to the Linux emulation model to attempt to improve its safety properties, it was difficult to determine the impact of the changes.

4.5 Kernel Implementation

The TrustedBSD capability implementation consists of a number of components: kernel modifications, include files, parts of the `libposix1e` userland library, and in the near future, userland tools for managing capabilities on processes and binaries. In this section, the kernel implementation is discussed; the userland library exists solely to provide conversion between the kernel system call API and the published POSIX.1e API, as well as to provide text-form management, utility routines, and memory management.

Within the kernel, access to capability definitions and functions is provided courtesy of `sys/capability.h`, which in turn relies on `sys/types.h`. In practice, many kernel files will include `sys/proc.h`, which includes the capability include file due to dependencies. `sys/capability.h` defines a capability set via `struct cap`, which stores capability flags in three `u_int64_t` bit-masks:

```
struct cap {
        u_int64_t       c_effective;
        u_int64_t       c_permitted;
        u_int64_t       c_inheritable;
};
```

The capability constants are selected to identify unique bits in each mask, permitting up to 64 capabilities to be defined in the currently implementation. The process credential structure, `struct ucred`, has been extended to include a `struct cap` to store per-process capability information. For kernel processes and the init process, the capability masks are initialized from constants defined in the source code. Kernel processes are provided with all capabilities so as to authorize their requests as required, including the inheritable bits, such that processes spawned from existing kernel processes also gain these privileges. The init process is also started with all capabilities, but with inheritance turned off by default. This allows init to manage the system, and selectively pass capabilities onto children processes as appropriate. Other than for the purposes of passing on privileges to children, init actually requires only the `CAP_BOOT` and `CAP_KILL` capabilities, allowing it to reboot the machine, as well as kill off other processes to do so.

Processes with capabilities set are protected against interference by other, inappropriately privileged, processes, in a manner similar to current protection of setuid and setgid processes. Init is also afforded special protection by virtue of having a process ID of 1.

All other userland processes will gain their capabilities as a result of a `fork()` call in their parent process, resulting in the child receiving a precise duplicate of the parent capabilities. After the process fork, the parent's and child's capability sets, as with the credential set as a whole, are managed independently.

Over the life time of the process, the capability set may change for one of two reasons: first, the process may directly manipulate the capability set using the provided `cap_get_proc()` and `cap_set_proc()` system calls; second, the process may invoke `exec()` to execute another program, in which case the inheritance rules described earlier are applied to determine the new capability state of the process.

The `cap_get_proc()` call allows the process to retrieve a copy of its current capability set. The process may then manipulate the copy in memory to its liking, possibly choosing to submit modifications of the set via `cap_set_proc()`. The modifications permitted tightly bound set of capabilities that can result from such a call; the `CAP_PERMITTED` flag set of the replacement capability set must be a subset of the old permitted set; the new `CAP_EFFECTIVE` flag set must be a subset of the new permitted set; the `CAP_INHERITABLE` set must be a subset of the old permitted set.

When a new program is executed by virtue of a call to `exec()` or one of its variant invocations, the new process capability set is calculated based on the formulas defined earlier, combining the prior process capability set and any capabilities retrieved from the program binary. This calculation is performed by `cap_inherit()`, which accepts a reference the the process's credential set, as well as to the vnode of the new program being executed. In TrustedBSD, capability sets will be read from the `$posix1e.cap` extended attribute, if present. As the attribute name begins with a "$" symbol, it is in the system namespace, and may only be set with appropriate privilege. If the attribute is not readable for the binary, either because the EA service is not available for that file system or file, the attribute is not defined for the file, or because the capability read is invalid, the binary will be assumed to have a capability set in which all flags are cleared. In practice, the formulas for calculating capability inheritance dictate that this results in a new process capability set with all capability flags are also cleared.

Capabilities on files may be modified in one of two ways: first, they may be modified directly via their extended attribute. This is generally not recommended, as it is not clear that all file systems will back capabilities into extended attributes, as they may have specific meta-data storage for capability sets. Using EAs to directly manipulate capability sets on files may also not be portable to other platforms, resulting in undesirable and unpredictable outcomes. POSIX.1e also defines two calls for directly manipulating capabilities on files: `cap_get_file()` and `cap_set_file()`. Calls to retrieve capability state require no special privilege, but setting capability state via `cap_set_file()` will require appropriate privilege.

In the FreeBSD kernel, checks for appropriate privilege are generally satisfied by calls to the `suser()` and `suser_xxx()` functions. These calls check that the effective uid of the calling subject (generally a process but sometimes a cached credential structure) is 0. With capabilities in the credential structure, two new calls are introduced to check privilege; in addition to accepting the process and/or credentials, the capability calls accept a capability to check for, which is compared with the credential structure's effective capability flags. Throughout the TrustedBSD kernel, traditional calls to the superuser functions have been replaced with `cap_check()` and `cap_check_xxx()`. To emulate the old superuser behavior, the capability functions can also optionally check the effective uid; the behavior of the check functions are determined by two system MIB entries, managed using `sysctl()`. `kern.security.suser_enabled`, when set to 1, causes an effective uid of 0 to result in a successful return. If `kern.security.capabilities_enabled` is 1, appropriate effective capabilities result in a successful return. Use of independent flags allows both behaviors to coexist in the same system, permitting use of legacy privileged software, as well as capability-enabled software, maintaining functionality while improving security.

4.5.1 Userland Integration

The new privilege authorization model in kernel requires modification of userland applications in a number of ways.

In the general case, applications must be modified to expect failure for any privileged operations they might perform; in many cases, they should also check for failures on calls that do not fail in the traditional UNIX semantics. An example of such a failure might be the inability for a process to change its uid, despite running as root, the bug that caused sendmail to behave improperly on capability-based Linux systems. New types of authorization checks often result in new situations in which said checks can fail.

It is also possible to characterize changes in applications based on the applications interaction with privilege. We categorize privileged applications into three categories, based on use, determining changes in their behavior.

First, applications requiring only a well-defined subset of superuser privilege, and currently based on setuid root binaries, may be changed to using a capability on their binary. For example, the `ping` and `traceroute` utilities currently require root privilege to allow them to bind a raw IP socket, so as to send and receive ICMP messages. By replacing the setuid bit with the `CAP_NET_BIND_SERVICE` capability, the two utilities can function normally with substantially less risk.

Second, the boot sequence requires substantial privilege in order to run: file systems must be checked, kernel modules loaded, and the network configured. Rather than associating capabilities with each of the binaries used during the boot, and relying on the utilities to enforce access policy, the privileges required are inherited from the init process. Init and the /tt /etc/rc script are responsible for determining that only appropriate privilege is passed on to utilities run during the boot.

Third, there are many useful aspects to a system administrator being able to acquire high levels of privilege to work with accounts and configure system services; maintaining the ability to acquire additional privilege is useful. In Trusted IRIX, the `su` command is extended to allow the acquisition of specific privileges, permitting a normal user to acquire, for example, the `CAP_DAC_READ_SEARCH` privilege on demand, allowing the user to read all files on the system for the purposes of a backup or virus scan.

The first type of integration proves relatively straight-forward. Most applications rely on the kernel to perform authorization checks, and fail rapidly if not. For example, the `ping` and `traceroute` commands rapidly return a meaningful error when they are unable to bind the raw socket they require. However, other applications short-circuit the call to the kernel for these services, instead attempting to predict the results of a bind call. For example, OpenSSH and other tools will call `getuid()` to determine the current effective uid, and fail to operate if they discover that it is non-0, even though they would be able to bind the privileged port the require or read the private files that they require to successfully operate. This raises an important distinction: some applications, such as SSH, require privilege to operate correctly, and the uid check should be performed in kernel. Other applications perform a uid check as part of their application policy, preventing inappropriate use. For example, it is appropriate for sendmail to restrict activities based on user trust, despite having the capabilities to perform them as any uid.

Both the boot process and the desire for superuser-like privilege during normal use can be addressed through similar mechanisms: a tool to, based on appropriate privilege, acquire or drop capabilities before executing another command, allowing that command to inherit the capabilities. In the case of the boot process, the utility would filter existing capabilities, whereas as a user utility, the command would read a policy file and set capabilities appropriately. In order for the superuser-like environment to work, applications not intended to run with privilege must be able to acquire that privilege. This can be effected by setting the appropriate privileges true in that binary's capability set, allowing it to acquire those privileges when inherited, but not providing them by default. Other applications without a capability set with relevant bits set positively in their effective set will be unable to inherit privilege in such an environment. This side effect actually makes for good policy, as it prevents applications not certified by the system administrator from acquiring privilege.

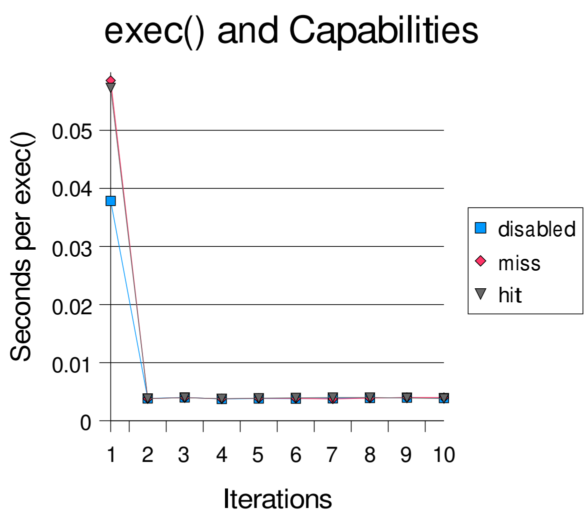

Figure 6: `exec()` and Capabilities

4.6 Performance

Capabilities generally impose little or no overhead during authorization, as the majority of checking consists of replacing an integer comparison in the `struct ucred` of a process with a bit comparison. The cost of loading capabilities from disk at `exec()`, however, is measurable, although generally acceptable. To measure the cost of capability-loading, we examine the time required to execute a minimal program, consisting of an empty `main()` function. Three cases are considered: first, with support for loading and evaluation of capabilities disabled via the `kern.security.capabilities_enabled` flag being set to zero. The, capabilities are enabled, and the read miss and read hit cases are examined, to determine if there is a difference in performance impact for binaries with and without privilege. As with previous performance tests, ten iterations are performed for each case, with a reboot between cases to flush all caches.

Name	Description
disabled	Latency to `exec()` capabilities disabled
cap-miss	Latency to `exec()` with no capability set defined
cap-hit	Latency to `exec()` with a capability set defined

In Figure 6, the performance results are shown. As with the extended attributes, when caching can not be leveraged, there is an observable performance difference between the baseline and capability-enabled conditions. Without capabilities enabled, the `exec()` of a minimal binary requires around 37.8 milliseconds when uncached, and on average 3.9 milliseconds when cached. The capability miss and hit cases are indistinguishable, as the 24-byte capability data is almost always on the same file system block as its corresponding attribute instance header. When not in the cache, the hit and miss cases take approximately 58 milliseconds each to complete, around 20 milliseconds longer than without capability support enabled. When cached, 3.9 milliseconds, essentially identical to the disabled capabilities case. The difference in measured extended attribute read time from the EA micro-benchmark and the capability `exec()` micro-benchmark is likely due to overlapping elements of each test case: the read-miss and read-hit EA test cases include the time for name lookup, VFS operations, and read of the inode and file meta-data from disk. In this capability performance test, the difference in cost was less than that measured in the EA experiment, which may reflect local disk caching conditions during the test, or increased numbers of in-progress disk transactions allow elevator sorting techniques to be more effective.

To understand the performance impact of capabilities on the system as a whole, and under load, we consider the impact of capability misses on buildworld, the build of the entire FreeBSD source tree. The majority, if not all binaries executed as part of the buildworld do not require privilege to execute; as such, all capability costs will be associated with the capability read miss. This is in line with the normal execution profile for performance sensitive applications. In order that the impact of concurrent capability loading might be more visible, the buildworld is executed with the "-3" argument, permitting up to three compiles to take place at once.

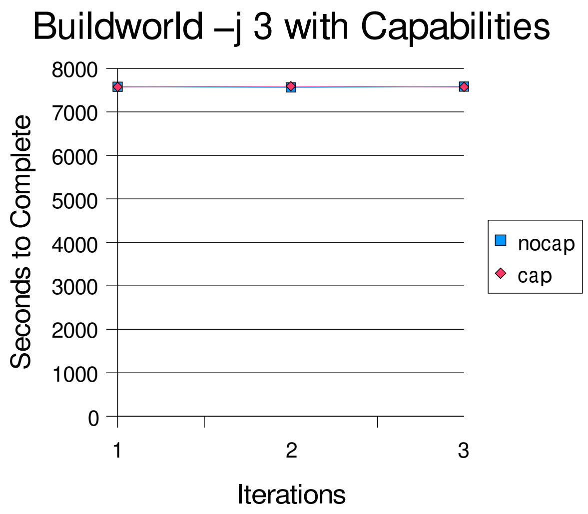

Figure 7: Buildworld -j 3 with Capabilities

Results from three iterations of buildworld, with and without the loading of capabilities from disk, may be found in Figure 7. The buildworld results show no significant difference before and after capability loading is introduced, suggesting that the overall impact on system execution is negligible due to caching, and the low impact of loading small extended attributes for infrequent `exec()` operations.

4.7 Potential Improvements

A number of improvements could be made to the current capability implementation to improve its performance and functionality.

Using extended attributes allows for convenient, rapid development and deployment of capabilities on TrustedBSD. However, this design choice has a number of down sides, both due to the EA implementation in FreeBSD, and the layering of capability functionality above extended attributes. Performance of capabilities is largely limited by the extended attribute implementation: the authorization checks themselves introduce negligible overhead in most cases. An improved extended attribute implementation, taking advantage of disk locality with the binary's inode, would lower the latency in executable startup.

The current layering of capabilities above the VFS layer introduces some other semantics issues: there are two paths to modify capabilities on a binary, with different privilege requirements. Similarly, setuid and setgid binaries enjoy certain advantages in the face of adversity: generally, when a setuid or setgid binary is modified on disk, the file system arranges for those bits to be disabled on the binary, reducing the opportunities for compromise due to a writable binary or race conditions.

Later in this paper, the process of integrating `jail()` and POSIX.1e capabilities is described. This integration of capabilities into base system functionality is still work in progress, but will substantially clean up the existing implementation.

Currently, the backup mechanisms included with the base system understand neither extended attributes, nor capabilities. In a real-world system, the ability to perform backups, as well as replicate file system hierarchies, is required for production use. This requirement could be filled either by modifying backup programs to understand capabilities on binaries, or by modifying them to understand the backing up of extended attributes. For portability reasons, backing up capabilities via the capability interface and generating an exportable format from them makes the most sense; system extended attributes generally have system-provided access mechanisms. Application namespace extended attributes could then be backed up directly.

Some of the strengths of capabilities are lost on the current implementation due to a lack of integrity protection for the Trusted Computing Base (TCB), leaving capability-enabled binaries unprotected against privileged modification. For example, possession of `CAP_DAC_WRITE` is sufficient to write to any root-owned file. This privilege can, in turn, be leveraged to modify binaries that possess other capabilities, gaining access to those capabilities. Protecting the TCB can currently be accomplished to some extent through securelevels, but to be properly implemented requires a system integrity policy, currently slated to be implemented in the next 6 months as part of the TrustedBSD project, but not yet available.

Userland integration is difficult, as it often requires reworking of current policies and implementations never intended for fine-grained privilege and access control. This work is also still in progress, with a number of binaries and the boot process still relying on a superuser. It is an eventual goal to disable `kern.suser_permitted`, relying purely on capability checks to provide privilege in the system. However, until all components of the kernel are updated to make use of `suser()`, a privileged uid 0 will be required.

Currently, the framework for allowing users to acquire privileges for the purposes of management, described in the previous section, is largely unimplemented, requiring mechanisms to store policy mapping capabilities to users. On some trusted operating systems, this is provided via a file, `/etc/capabilities` and the `su` command. It is likely that much the same will be implemented in TrustedBSD, although FreeBSD already has a user capability-assigning file, `/etc/login.conf` which allows classes to be assigned to users, and capabilities to be assigned to the classes.

4.8 Portability Efforts

The POSIX.1e capability functionality has been implemented in a number of operating systems, including Linux, IRIX, Trusted IRIX, and now TrustedBSD. As the POSIX.1e draft was never standardized, and operating systems may require local customizations to the capability sets supported, there are limits to portability. IRIX, for example, implements draft 16 of the capability inheritance rules, which have markedly different properties than those in the TrustedBSD and Linux draft 17 implementations.. The Linux and TrustedBSD implementations intentionally share many properties, including custom capability extensions, and rules. The POSIX.1e and linux-privs mailing lists, and SGI-hosted workshop have proven excellent forums for discussing the semantics of capabilities.

A number of differences remain, even between the largely identical Linux and TrustedBSD implementations, which privileged application developers should be aware of. First, some operating system specific capabilities have marginally different definitions: CAP_LINUX_IMMUTABLE in the Linux implementation is mapped to CAP_SYS_SETFFLAG in TrustedBSD, reflecting the privilege to modify system file flags, rather than purely the Linux immutable and append flags. The Linux technique of emulating uid-0 privilege with capabilities may result in surprises in mixed-capability/setuid environments, especially when capabilities on binaries are not supported. Similarly, applications written with the Linux composition model in mind may find that despite having given up capabilities, they still have privilege when running as uid 0 with `kern.security.suser_enabled` set to true. Such applications should be modified so as not to run with root privileges, and so that they gain appropriate capabilities via their binary, avoiding the subtractive technique employed in Linux using the setuid root mechanism.

Work also continues on a consistent bounding interface, so as to allow both TrustedBSD and Linux to support the same semantics for inherited capability bounding.

5 Abstracted Kernel Access Control Checks and Labels

One important lesson learned in the TrustedBSD work is that the BSD kernel and userland code are designed to enforce a very specific operating system policy. This has been largely intentional: a strong system requires a consistent policy throughout the implementation, and the UNIX security model has satisfied the requirements of many environments quite well. However, adding new access control models raises the need for improved abstractions, as well as more careful and consistent use of existing abstractions. The existing code suffers from a number of limitations when it comes to improving access control abstractions.

5.1 Coding Style Changes

- The existing code often checks credentials for specific privileges directly, rather than making use of even the basic abstractions available now, such as `suser()`. This is frequently the case in the file system, where direct checks on the effective uid are made instead of invoking `suser()`. Frequently, this poor use of the `suser()` abstraction was justified due to `suser()` setting a process accounting flag, `ASU`, indicating that superuser privilege had been invoked. In the capabilities implementation, this behavior has been modified so that a separate API call, `suser_used()` accounts for the use of privilege by a process, allowing `suser()` to be called even when the use of privilege has not yet been determined.

- A number of system APIs rely on divergent behavior: that is, the behavior of the call varies depending on the privileges of the caller. While in some situations this is fine, in other situations the access control model breaks down from, "Is this action permitted?" into, "If this credential is found, act this way; if this other credential is found, act in this other way." This type of code structuring does not lend itself to improved access control abstraction, as divergent behavior in one abstraction may make little sense in another, especially in the presence of compound requests where some aspects of the request may succeed, while others fail. Where possible, splitting the API into individual components, each with a "Succeed if appropriately privileged, otherwise fail," will allow the access checks to be more consistent. Mapping individual requests onto individual access control decisions dramatically improves the extensibility of the code from the perspective of security additions.

- In the networking code, privilege is often checked at one point in the code, and then used elsewhere, several layers deeper in the call stack. For the purposes of auditing the use of privilege, this makes it very difficult to determine if privilege was ever used. With the breaking out of the `ASU` flag functionality from `suser()` into a separate `suser_used()` call, it is now possible to differentiate between a check for appropriate privilege, and accounting for its use. When auditing is implemented, this will make it possible to audit the use of privilege as distinct from the possession of privilege. However, a fair amount of existing code is not structured with this distinction in mind.

- Checks for access to an object are often not ordered based on the desired semantics: typically, it is desirable to work from the least privilege to the greatest privilege, attempting to perform the access, so that auditing of privilege can happen correctly. In the file system code, the check for privileged access frequently happens before the check for discretionary access to the object. In some situations, this is hard to change, as a privileged access may provide different results than an un-privileged one: this violates one of the earlier recommendations, suggesting that diverging behavior in APIs due to credential differences can cause problems for generalized access control.

- The system call and service layering infrastructure in the kernel often passes down a reference to the current process, or current credentials. At some points in the code, access to either the process or the credential is cached for future use, such as in sockets and open file entries, so as to provide the "authorize on open" semantics of UNIX. However, in many situations, only one of the process structure or credentials are passed; as the relevant credential information is currently split over the two (process authorization flags in one, uids in the other), both are often required. Moving all authorization-related material into `struct ucred` would permit only the credential to be passed around for access control, without need for reference to the process, making it easier to generalized access control based on this credential.

5.2 Authorization Infrastructure Changes

In an attempt to improve the structuring of access control code, a number of access checks were modified for capabilities support.

First, as described previously, separation of accounting and access control for privileged actions were implemented, by breaking out the `ASU` flag from the `suser()` access check. This change is maintained in the switch from `suser()` to `cap_check()`.

Second, common access checks were centralized into the same permission checking routing, allowing policies to be understood and modified more easily. In the case of direct interaction between processes, access checks were scattered throughout the kernel, often applying different policies for the same action, depending on the access method chosen. After the modifications, a centralized routine, `p_can()` is invoked with the two processes and an operation specified as an argument. So that privilege may be properly tracked, an optional pointer to an flag is passed, allowing the function to indicate to an interested caller whether or not privilege was required for the call to succeed.

Similar updates were made to permission-related access checks in the file system; access control checks were centralized in a single call, `vaccess()`, which was structured to order attempts to gain access to an object based first on discretionary protections, then based on a privileged override of those protections. In calculating the use of privilege in the revised access control check, the minimum capabilities required to perform the operation are also calculated; when auditing is available, this permits the type of privilege to be audited also, in the form of a capability bit-mask.

These access check improvements generally fall into a pattern: a subject (process) attempts to perform some operation on an object (a file, process, or other kernel resource), which may be protected by discretionary access control (permissions or an access control list), or by mandatory access control (limits on interactions between users and classes of objects). After these checks are performed, a specific capability can be invoked to override specific limitations–in some cases, multiple capabilities may be required to perform the action, as multiple policies may have to be violated for it to succeed. For example, in the file system code, a one capability may be required to override discretionary checks on the file, while another privilege may be required to override mandatory access control limits. Similarly, in the realm of signal delivery and debugging, one capability may be required to override uid-based limits on the request, whereas another may be required to override MAC limits.

As determined earlier, the order and relationship between the the access control mechanism invocations is important, and constitutes a meta-policy in and of itself.

5.3 Improving the Structure of Services

If these abstractions are carried further, there are a number of goals which can help improve the cleanliness and extensibility of the access control code:

- Structure the implementation of services such that the number of independent access control checks is minimized. This permits meta-policies concerning the relationships between different types of checks to be performed centrally. Typically the relationship will be an intersection of rights, or a union of rights (only if all checks succeed, or if any check succeeds). Centralizing this policy will make it easier to understand, and easier to change.

- Attempt to structure implementation of services such that it expresses an operation by a subject on one or more objects. As mandatory access control policies make decisions based on security labels on objects, this makes it easier to introduce such policies. For example, labels might be associated with a variety of objects, including processes, files, and sockets. Labels might also be associated with system resources, such as interfaces, packets matched by specific packet filters, and `sysctl()` MIB entries. Structuring access control in this manner makes it easier to centralize and change access control policies, as well as making it easier to reason about the behavior of the system from a security perspective.

As these changes are made, however, it also important to, in as much as is possible, retain the current apparent semantics of system services. While new failures may be possible when processes make use of a service, the service itself should generally remain unchanged, allowing applications to run in both environments with little modification.

6 Mapping FreeBSD Hacks into TrustedBSD Features

FreeBSD has a number of seemingly ad hoc security features, both inherited from BSD 4.4Lite, and introduced more recently. These include the "securelevel" functionality, intended to protect system integrity in the event of a root compromise, and the `jail()` host partitioning scheme. These features are, in many ways, subsets of more general security models that will be available in TrustedBSD. Not all of the TrustedBSD features required to express these features are currently available, so this section provides a description as to how improvements could be made.

6.1 BSD Securelevels

Securelevels are intended to protect the system in the event of root compromise, distinguishing more clearly the rights of the superuser and the kernel. The securelevel is maintained as a monotonically increasing integer; as the securelevel increases, superuser privileges are gradually restricted, protecting the kernel from manipulation, protecting key files in the file system, and preventing modification of system configuration. In practice, securelevels afford little in the way of protection against qualified attackers, as key configuration files, binaries, and hence directories, must be preserved from manipulation by privileged users. As a classic example of the tradeoff between security and usability, as the restrictions are put into place, the ability to manage the system dramatically drops: a remarkably number of files are touched during the early boot process, including much of `/boot`, `/sbin`, and `/etc`.

Much of the securelevel functionality could be successfully represented using a Biba MAC integrity policy; worked examples of such systems include Trusted IRIX, which labels system objects such that low-integrity processes cannot interfere with them. Integrity protection is also applied to processes themselves, as well as to system devices. Restrictions of available privilege map well into inherited capability bounds, allowing appropriate high-integrity processes to maintain the rights required for system management, will preventing low-integrity processes from gaining inappropriate rights. With proper integrity protection, the privileges provided to the root user by virtue of it owning system files would also be reduced.

6.2 FreeBSD `jail()`

The `jail()` functionality was introduced in FreeBSD 4.0-RELEASE, and is described in [KW00]. The general intent is to limit the scope of privilege acquirable by an appropriately privileged process and all of its descendents, allowing implementation of limited system partitions. This is implemented through the `chroot()` function and modifications to limit the visibility of inappropriate components of the system, such as processes in a different partition. To prevent privileged processes within the partition from escaping, the scope of superuser privileges is limited to only those calls unable to penetrate the jail.

In the current `jail()` implementation, access to superuser privilege for specific requests is specified on a per-`suser()` basis, via the `PRISON_ROOT` flag in `suser_xxx()`. When the flag is present as an argument to `suser_xxx()`, the access check will allow an effective uid to succeed, even if the process is in a partition; otherwise, all `suser()` calls within the jail will fail. With the advent of fine-grained capabilities on processes, it is possible to describe the subset of calls permitted to occur in the jail via a inherited capability bound, rather than per-check flags, providing a cleaner abstraction for jails, as well as more flexibility in the rights permitted in a jail.

With capability bounds, the custom superuser checks can be replaced with normal capability checks; capabilities within the jail must be restricted to prevent abuse of privilege allowing processes to break out of the jail. The following capabilities are generally safe within a jail; others may also be safe subjected to careful control, depending on requirements:

Capability	Comment
CAP_CHOWN	-
CAP_DAC_EXECUTE	-
CAP_DAC_WRITE	-
CAP_DAC_READ_SEARCH	-
CAP_FOWNER	-
CAP_KILL	Subject to mandatory process interaction policy
CAP_SETFCAP	Possibly subject to limits on what capabilities can be set
CAP_SETGID	-
CAP_SETUID	-
CAP_SETPCAP	Subject to process capability bound
CAP_NET_BIND_SERVICE	-
CAP_IPC_LOCK	-
CAP_SYS_PTRACE	Subject to mandatory process interaction policy
CAP_SYS_NICE	Subject to mandatory process interaction policy
CAP_SYS_RESOURCE	Subject to mandatory process interaction policy
CAP_SYS_TTY_CONFIG	-

With the capability implementation, these rights may be further restricted as needed. Once mandatory access control policies are available, special-case protection of processes and system objects may also be replaced with a general-purpose partitioning scheme, as well as integrity and secrecy protection. One interesting and as-yet unaddressed problem concerns use of MAC within the partitioned environments: to what extent will administrators in individual partitions be able to administer mandatory access within their partitions? Traditional MAC schemes have not targeted scalability in terms of the number of compartments or partitions to be protected, as typical consumers of MAC have not required that. However, given that the `jail()` functionality is a popular server tool in FreeBSD, optimizing the mandatory access control policies for a scalable partitioning scheme makes a great deal of sense.

7 Future Directions

Future directions for the TrustedBSD project include improvements on the infrastructure described in this paper, as well as additional feature sets; both are in progress at the time of publication, and will be on-going, with many features targeted for the FreeBSD 5.0 release.

The extended attribute implementation is already sufficient for a developmental prototype, and even release code, but, there are a number of performance and consistency-related improvements that can be made to make this service more palatable for production use. A useful long-range project might be to integrate EA storage into the underlying file store, improving interaction with consistency mechanisms such as soft updates and journalling.

The capabilities integration into the FreeBSD kernel is largely complete; however some remaining conversions from superuser to capability checks remain to be implemented. Integration of capabilities support into the userland environment is still underway: modifying existing applications that require superuser privilege to handle capabilities as an alternative, and introducing a mechanism to manage per-user privileges at login remain to be done, but are important goals. The capabilities implementation should be ready to be committed to the FreeBSD CVS repository for inclusion in FreeBSD 5.0-CURRENT with another month of open testing.

The mandatory access control implementation is the next major undertaking, but may require substantial improvement of the label handling and access control infrastructure. Integrity-based policies will allow improvements of system security and replacement of the experimental securelevel behavior. Combined with secrecy policies and a generalized partitioning scheme, existing security "hacks" in FreeBSD, such as `jail()`, may be improved and replaced, as well as permitting FreeBSD to be used in trusted environments.

Event auditing is an important component to any trusted operating system; a number of different implementation approaches have been attempted when integrating auditing into FreeBSD in the past. The TrustedBSD auditing implementation must leverage these experiences, and emphasize cleanliness of implementation and performance if it is to be useful in common environments.

A long-term implementation goal is the development of a general-purpose kernel access control and label management framework, Poligraph, which would allow pluggable security policies and protection mechanisms. The existing TrustedBSD implementation builds experience and identifies sticking points that will require work to implement such a system, but in a second-generation implementation, such a system would dramatically improve the flexibility of TrustedBSD components, as well as allowing third-party security modules to be developed and integrated easily. An important long-term goal of the project is to develop such a system for TrustedBSD, and if the changes prove acceptable to the FreeBSD developer community, integrate that system into the FreeBSD source base.

8 Conclusion

The TrustedBSD project seeks to introduce trusted operating features into FreeBSD through an aggressive schedule of modifications: improving label management services, access models and abstractions, and file system services. Several of these features are nearing completion, including the file system extended attribute support and capabilities service described in this paper, which substantially improve security without substantial performance degradation. Integration of these improvements into the base FreeBSD operating system is on-going, with these services slated for inclusion in FreeBSD 5.0-RELEASE. Future directions for the project include improved access control abstractions, mandatory access control policies, and fine-grained event auditing.

References

[Ber95] B. Bershad. Extensibility, safety, and performance in the spin operating system, 1995.

[Bib77] K. Biba. Integrity constraints for secure computer systems, 1977.

[BL73] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model, 1973.

[(CC00] NIST Common Criteria Implementation Board (CCIB). Common criteria version 2.1 (ISO IS 15408), 2000.

[Dun89] Roy Duncan. Design goals and implementation of the new high performance file system, 1989.

[IEE90] IEEE. *Information technology–Portable operating system interface (POSIX). Part 1, System application program interface (API) : C language.* Institute of Electrical and Electronics Engineers, inc., December 1990. IEEE Std 1003.1-1990; revision of IEEE Std 1003.1-1988.

[IEE97] IEEE. Draft standard for information technology–portable operating system interface (POSIX)– part 1: System application program interface (API)–amendment ??: Protection, audit and control interfaces: C language, October 1997. PSSG/D17, POSIX.1e.

[ISSO99a] National Security Agency Information Systems Security Organization. Controlled access protection profile version 1.d, October 1999.

[ISSO99b] National Security Agency Information Systems Security Organization. Labeled security protection profile version 1.b, October 1999.

[KW00] Poul-Henning Kemp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings, SANE 2000 Conference.* NLUUG, 2000.

[MBK+96] M. McKusick, K. Bostic, M. Karels, J. Quarterman, and D. Implementation. BSD operating system, 1996.

[MG99] M. McKusick and G. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem, 1999.

[oD85] United States. Dept of Defense. *Department of Defense trusted computer system evaluation criteria.* Dept. of Defense, December 1985. Supersedes CSC-STD-001-83, dtd 15 Aug 83. Library no. S225,711.

[Pfl96] Charles P. Pfleeger. *Security in Computing.* Prentice Hall PTR, second edition, 1996.

[Sec00] SecurityFocus.com. Linux capabilities vulnerability, 2000. http://www.securityfocus.com/bid/1322.