# Enabling the Prevention of File System Races by Augmenting Existing Interfaces

Weston A. Adamson, Marius A. Eriksen, David M. Richter
*University of Michigan*
{muzzle, marius, richterd}@citi.umich.edu

## Abstract

*Race conditions in file systems occur when a programmer assumes that file data and metadata are immutable across a series of file system operations, when in fact there is no mechanism enforcing such a constraint. Such race conditions remain a largely-ignored security issue, yet their impact is certainly non-trivial. Security researchers have mitigated many common file system races by modifying existing file system operations and adding new ones. Though effective, these solutions do not provide any way to prevent arbitrary race conditions. That is, they do not provide any generalized mechanism with which a programmer can protect file system resources across multiple operations.*

*We present a general solution to this problem:* `fprotect`*, which is a small extension to the OpenBSD VFS and the POSIX file system interface. Our functional implementation provides atomicity, consistency, and isolation guarantees across multiple file system operations by introducing a transactional facility which restricts access to selected file system objects. We describe the nature of file system race conditions and how our system protects against them, investigate avenues for integrating our extensions into existing applications, and present a performance analysis.*

## 1 Introduction

Race conditions in file system operations are widely acknowledged as a cause of security problems. They were discussed as early as the 1970s [3] and, as a class of security vulnerabilities, are known to cause privilege elevation when successfully exploited.

The execution of many programs depends on metadata and data from the file system; consequently, security-conscious programmers will check file system conditions before acting upon them. The time between the checking of a condition and its use is a *critical section*; that is, a period during which a program's correct execution depends on the guarantee that some condition(s) remain invariant. Race conditions occur when the operating system does not fully enforce consistency in critical sections.

The generality of the POSIX file system interface necessitates the use of many fine-grained operations to perform tasks on file system objects. The interface guarantees exclusive access to the file system objects data and metadata for the duration of a *single* operation, but has no mechanism to enforce exclusive access across *multiple* operations, and thus cannot ensure that a file system object remains invariant throughout a critical section.

Our solution to file system race conditions is to augment the POSIX file system interface to enable programmers to demarkate critical sections of operations; we call this new system `fprotect`. It *guarantees* atomicity, consistency, and isolation in the specified critical sections. This provides the programmer with the ability to carry out a *transaction*, during which we guarantee a coherent and reliable unit of interaction with a number of file system objects. This approach has an intuitive appeal, since the programmer knows which consistency semantics are vital to their application.

Realizing that any interface change is a hard sell, the primary goals of our work are to make the interface as simple as possible while maintaining strong security guarantees. We also aim to provide programmers with a very simple migration path; applications using our extensions are well-behaved when mixed with "legacy" applications that do not use the API, providing the same strong consistency guarantees.

The threat model that this API attempts to mitigate is a lower-privileged adversary achieving privilege escalation by exploiting some condition of the file system in a *critical section*.

File system races that exploit such a *critical section* can typically be categorized as "Time Of Check To Time Of Use" (TOCTTOU) races [3, 11]. An example of a TOCTTOU race condition is when a privileged process calls the `access()` operation (the "check") followed by an `open()` operation (the "use"), which will open a file if `access()` returns successfully. In this case, an adversary could replace the file with a link to another file, perhaps critical to the system, thereby effectively hijacking the privileged `open()` operation to open a file of their choice. This "hijacking" would have to occur between the `access()` and `open()` calls, since the privileged process would be able to notice if it occurred before the `access()`. TOCTTOU races, such as this, occur with a variety of other file system operations as described in more detail in section 2.

largely ignored due to the relative ease of other, perhaps more expedient, attacks; code injection is a prime example. As code injection becomes more difficult with the introduction of systems such as W⊕X [1], StackGuard [7] and PointGuard [8], it is realistic to expect adversaries to change their focus in the near future. Additionally, while file system races inherently require that the attacker already have some amount of access to the machine in question, this requirement is not at all far-fetched enough to disqualify them as dangerous threats. In fact, industry research [16, 14, 5] shows that the overwhelming majority of unauthorized access and intrusions, and in particular the most-costly attacks, involve an "insider" who already has access to the system.

Many of the simpler and most common race conditions have been mitigated by altering and adding to the exploited interfaces. Most of these changes are pertinent only to single operations and, though some yield multi-operation consistency, often only in very limited ways. Policy-driven race condition detection systems have also been proposed [21], but are limited by the body of exploit knowledge available at the time. Policy-driven systems are therefore constantly playing "catch-up" as new race conditions are discovered.

It is worth noting that our approach *facilitates* correct behavior – by allowing the programmer to protect file system objects – instead of attempting to detect and prevent race conditions dynamically. It has been shown in the past that simple changes to established APIs may provide great benefits to correctness in code. Specifically, the introduction of the `strlcpy()` and `strlcat()` interfaces to OpenBSD [15, 1] has provided very visible advantages and has also served as a case study for adoption by nearly every major open source operating system.

The rest of this paper is organized as follows: in section 2 we present examples of common types of race conditions; in section 3 we describe the design and implementation of our API; in section 4 we examine the performance of our characteristics of our API; in section 6 we describe related work; and in section 7 we summarize and conclude.

## 2 Filesystem Race Conditions

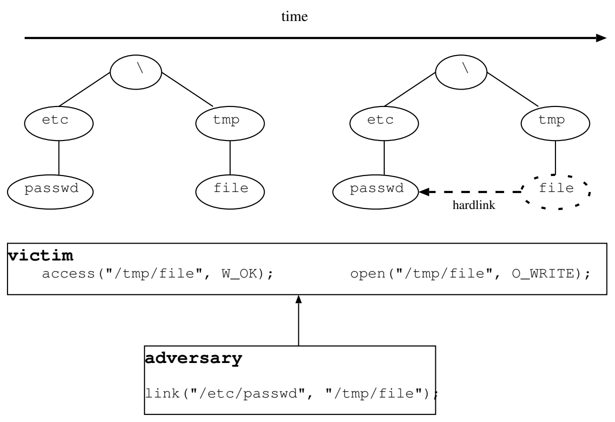
time
\
etc
tmp
passwd
file
\
etc
tmp
passwd
file
hardlink
victim
access("/tmp/file", W_OK);
open("/tmp/file", O_WRITE);
adversary
link("/etc/passwd", "/tmp/file");

Figure 1: A sample TOCTTOU race condition. In the time between calls to `access()` and `open()` by the victim process, an adversary calls `link()` , effectively hijacking the `open()` call.

Race conditions relating to file system object accesses are dangers endemic to the majority of current operating systems. At some level, it seems incongruous that so much effort is spent enforcing process isolation in the address spaces of multiprogrammed environments when that very isolation fails to carry over into the file system used by those processes. In some cases, the application programmer is safe making assumptions about their program executing independently of other elements in the system; in others, great caution is required. Some specialized domains, like database management systems, have been forced to confront race conditions because of the extremely high frequency with which their data are accessed. The insidious nature of race conditions in general-purpose file systems stems from the fact that races are normally quite rare, difficult to exploit, and correspondingly difficult to assess. Despite their usual scarcity, race conditions are nevertheless dangerous potential exploits which arise from unfounded assumptions about the atomicity of grouped file system operations. File system races are essentially exploited by the same pattern: first, access to or the existence of some file system object is checked by the victim; then, an attack is performed via operation interleaving; last, the victim performs some file operation that is unsafe because conditions changed after the check.

## 2.1 File Access

Tsyrklevich [21] describes a trivial case wherein a setuid program uses `access()` to check whether a user has the ability to open a file. If the `access()` call succeeds, a subsequent call to `open()` obtains a file descriptor. The race lies in the fact that both `access()` and `open()` refer to the file in question by its pathname. There is no guarantee that the pathname refers to the same inode by the time the setuid program calls `open()`; that is, after the `access()` check, an attacker could delete the original file and create an identically-named symlink to a protected resource (see figure 1). The `open()` call would then hand the compromised file to the attacker.

An exploit we discovered assumes a threat model wherein the attacker already has some administrative access on the machine in question and seeks to perform privileged operations covertly (e.g., to avoid indictment or to frame another privileged user). In OpenBSD, the `vipw` command is used to update the password file safely. Despite that `vipw` verifies the user's authorization, acquires locks on the files involved, performs consistency checks on the modifications made, and only thereafter generates a new, secure password database, it is nevertheless exploitable. After starting up, `vipw` creates a child process (a shell interpreter) which also creates a child process (normally `vi`), with which the user edits the password file. After creating its child, `vipw` then calls `waitpid()`, which blocks until `vi` has returned.

A vulnerability window exists between the time that the `vi` exits and the time that the parent `vipw` resumes execution – at which point the password database is updated if `vi` exited cleanly. Though the weakness is obscure, it can be easily exploited by a simple shell script that waits for a valid user to start `vipw`, then waits until the (well-known) temporary file changes, at which point the script quickly inserts into it a well-formed password file entry. When `vipw` resumes, it will commit the attacker's entry to the password database, which in our example creates a new user with arbitrary group memberships. Attacks of this sort could easily go entirely unnoticed, considering that an intrusion detection system would find nothing wrong with the unwitting user legitimately updating the password file.

## 2.2 Directory Exploits

A post to the bug-fileutils mailing list [18] describes how an error in the GNU file utility `rm` could be used to delete arbitrary files and directories. The recursive, depth-first file deletion behavior of, e.g., `rm -rf` has a race that occurs after the utility has descended into a given subdirectory (e.g. `/tmp/foo/bar`) and has deleted the directory contents. If an attacker is able to perform an operation like `mv /tmp/foo/bar /tmp` just before `rm` calls `chdir("..")` to back out of the now-empty `/tmp/foo/bar` directory, `rm` will end up in `/` instead of `/tmp/foo`. This causes `rm`, unwittingly, to attempt to `unlink()` the contents of the root directory. This exploit would be disastrous if a privileged account performed the `rm -rf` operation. Though this particular bug has been fixed, it went undiscovered in the wild for years and countless utilities with recursive directory-traversal functionality are potentially vulnerable to the same style of attack.

Bishop [4] relates a different exploit where a privileged process intends to create a directory and change its owner to a less-privileged user. A race is present after the completion of `mkdir()` and before the intended `chown()`: assuming an attacker has access to the new directory, she can delete it and create an identically-named symlink to a critical file. The privileged process will then use `chown()` to grant the critical file to the attacker.

## 2.3 Temporary Files

Recent work [9] has been devoted exclusively to combating race conditions centered around the creation of temporary files. A simple example has the victim test for a file's existence (using `stat()` or `access()`) and create it when it is not found. An attacker could exploit the time between the test and the file creation by creating a symlink to another file; when the file creation occurs, the symlink target is overwritten. A twist on this approach instead uses the symlink to create a file whose existence is dangerous, e.g. `$HOME/.rhosts`. Another variant exploits a victim program that checks a dummy file to see if the attacker has access privileges; after the test, the attacker symlinks the dummy file's name to a critical file that is then overwritten by the victim.

## 2.4 Setuid Scripts

A recent paper [21] explains a problem that early Unix implementations had with safely executing setuid shell scripts. If the first two bytes of an executable are "`#!`", the kernel sees the file as a script and will read the remainder of that line as a path to the script's interpreter. In some systems, a race can occur after the first line is read and before the interpreter is launched. During this time an attacker has a chance to interpose a symlink to an alternate script, which the interpreter would then run with root privileges.

```
struct fprot_obj {
        char  path[MAXPATHLEN];
        int   flag;
};
int fprotect(struct fprot_obj *objs, u_int nobjs, int flags, int timeout);
int funprotect(struct fprot_obj *objs, u_int nobjs);

Flags:
FPROT_PARENT   locks the parent directory of the specified object
FPROT_PROXY    pass the lock by proxy to child process(es)
FPROT_HIDE     mark file system object ``invisible'' to other processes
```

Figure 2: The `fprotect` API. A set of objects grouped in an `fprotect()` call must be grouped in the same set passed to the corresponding `funprotect()` call.

# 3 Design and Implementation

## 3.1 Interface Design

In order to allow userland programs to demark *critical regions* of file system activity, we added two system calls to OpenBSD-CURRENT: `fprotect()` and `funprotect()` (Figure 2 shows the API[1]). These system calls manipulate the *lockset*, a list of file system objects ("objects") which are in a critical region, of a process. Whenever the lockset is nonempty, the process is in a critical region ("fprotect region"). `fprotect()` adds to, and `funprotect()` subtracts from, the lockset. A group of objects added by `fprotect()` must be removed in the same group by `funprotect()`.

Whenever an object is in the lockset, the process is provided with a set of guarantees for that object:

1. *Exclusive access.* No other process may manipulate the object while it is in the lockset, unless child processes are explicitly granted proxy-access.
2. *Atomicity and Consistency.* All changes to the object are either applied to the object successfully, or not applied at all.
3. *Isolation.* Any changes made to objects are not visible to other processes until the object is removed from the lockset.

With `fprotect`, objects are referenced by their *pathnames*. These pathnames represent the objects as resolved by those pathnames at the time of the `fprotect()` call. Associated with each pathname is a list of protections applied to that object. A flag is passed into `fprotect()` along with this list of paths indicating hints and flags that `fprotect()` can use to configure and optimize its operations.

The interface also specifies a timeout indicating the maximum amount of time that should be allowed for `fprotect()` to return (this value may set to infinity). `fprotect()` may block, since it might need to wait for exclusive access to a file already exclusively held by another process.

`funprotect()` has arguments similar to `fprotect()`, only with an inverse effect: pathnames passed to `funprotect()` will be removed from the caller's lockset. If `funprotect()` is called without a list of object descriptors, it will remove every object in the process's *lockset* – the list of all objects currently protected on the behalf of that process.

Note that our API allows for manipulation of the *lockset*; that is, the *lockset* is simply a set of references to files that are currently under `fprotect`'s watch. This means that `fprotect()` and `funprotect()` calls may be nested. This method allows programmers to specify critical regions over separate file system objects which may be interleaved, giving a programmer as much flexibility as possible while still keeping the interface simple.

When a process exits (normally or abnormally), its lockset is emptied. Newly-created child processes do not inherit their parent's lockset; however, `fprotect()` accepts a flag that will grant the caller's child processes proxy access to protected objects. In the current implementation, this means that child processes are not allowed to mutate their parent's lockset and that the parent process is responsible for freeing the locks.

[1] We refer to our system simply as `fprotect`.

## 3.2 Locking System

In order to maintain internal consistency, the OpenBSD VFS layer enforces exclusive file access for the duration of a *single* file system operation ("operation"). This is done by holding a vnode lock, the *primary vnode lock*, on the underlying object being mutated by that particular operation. This lock is acquired through the underlying file system and guarantees exclusive access to that object. We extended this mechanism with the ability to maintain these locks across *several* file system operations while still adhering to the internal consistency semantics in the VFS.

Recall that, under `fprotect`, every process maintains a lockset. The process also maintains a locked instance of each object in its lockset. By VFS semantics, an object is mutated only when this lock is held. Thus, by this simple scheme, a process has exclusive access to all objects in its lockset, even across multiple operations.

However, a naïve implementation relying on this scheme of holding primary locks during regions protected with `fprotect` has two show-stopping issues. We circumvent these difficulties by extending the VFS locking scheme with *lock polling* and with the introduction of a *secondary vnode lock*.

**Lock Polling** A significant problem with a simple locking scheme is that if a file $f$'s lock is already held by process $p_1$ when process $p_2$ tries to `fprotect()` $f$, $p_2$ must sleep while waiting for the lock to be released. This gives rise to a problem when several file names are passed to an `fprotect()` call and all of their locks need to be acquired as a group. For instance, assume `fprotect()` needs to acquire $x$ locks as a group and $y < x$ of them may be acquired immediately; even though the first $y$ locks can be acquired without waiting, the next lock requires the calling process to sleep. If other locks become available during this sleep, the process has no way to acquire them as they become available. This can lead to unreasonably long periods of time during which `fprotect()` holds some lock(s) while waiting for another to become available. Note that all of the time a lock is held within an `fprotect()` call is "wasted time"; that is, the calling process cannot utilize the lock until `fprotect()` returns control. Worse, this scheme is inherently deadlock-prone.

To mitigate this problem, we implemented *lock polling* in OpenBSD. Lock polling allows the process to register interest in many locks, then sleep on a single wait channel which is activated when any of them become available. Effectively, this allows the process to wait for *many* locks at once, never sleeping while waiting for just a single lock to become available. This solves our first issue and also accommodates `fprotect()` timeouts, since whenever a process sleeps in `fprotect()` it does so on a single wait channel controlled by fprotect, and thus can be timed-out.

Our lock polling works as follows: when using `fprotect()` on a group of files, each descriptor's pathname is resolved to a vnode by `namei()`, after which `fprotect()` attempts to lock the vnode with a non-blocking call. If that call fails, a flag is set on the unavailable vnode, the calling process is added to a wait channel analogous to those used by `poll(2)`, and the calling process `sleep()`s. When the vnode is eventually unlocked by `funprotect()`, all processes waiting on the channel will be awakened, at which point they resume attempting the lock acquisition.

**Secondary Vnode Lock** The VFS routines involved in resolving pathnames to vnodes have two characteristics which complicate `fprotect`'s role: the routines are invoked very frequently and can only resolve pathnames without blocking when each component of the pathname is unlocked. Moreover, these routines are also used by `fprotect`. Performing a lookup operation on a locked file system object would often block for unreasonably long periods of time. This problem would likely cascade by triggering the deadlock prevention code (see section 3.6) and causing many operations to fail when otherwise they would not.

In order to avoid these problems, we implemented a *secondary vnode lock* associated with each vnode. While our system effectively extends the protection granted by the primary vnode lock from the length of a single VFS operation to the length of the entire fprotected region, a shorter-duration lock is still needed; the secondary vnode lock provides this. Its duration is a single VFS operation, and in this case we use it to grant temporary, read-only access to a vnode for the express purpose of performing unimpeded pathname resolution. More specifically, when an object is in a lockset, the VFS lookup routines acquire the secondary lock during the (short-lived) vnode operations. If the calling process owns the lockset, any subsequent mutating operations it performs will be allowed, since the caller also holds the primary lock. However, if the calling process is not the owner, the secondary lock will allow the pathname resolution to proceed quickly, but any other file system operations will be disallowed because the primary lock is not held.

It is important to stress that the presence and usage of the secondary vnode lock does not violate our lockset semantics, since the lock is employed only during one VFS operation and does not confer any mutability

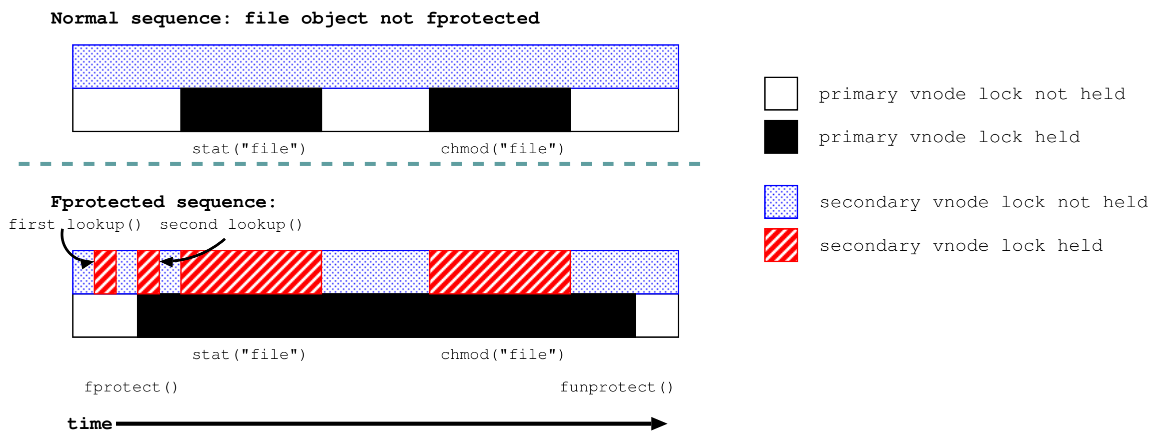

Figure 3: A demonstration of the two-tiered locking mechanism. In the *normal sequence*, the primary lock is acquired then released by both the `stat()` and `chmod()` calls. In the *fprotected sequence* the primary lock is acquired in `fprotect()`, held for the duration of `stat()` and `chmod()`, and released in `funprotect()`. The secondary lock (fprotected region only) is held for the involved VFS operations: the two `lookup()` operations called in `fprotect()` (as explained in section 3.3), as well as the `stat()` and `chmod()` operatations.

rights to the holder. This approach succeeds in allowing processes to safely perform arbitrary pathname resolution while avoiding the possibility of blocking indefinitely.

## 3.3 Name Resolution Consistency

One of the implementation challenges with `fprotect` was to avoid race conditions when resolving pathnames into their corresponding vnodes. In OpenBSD's VFS, name lookups are done through the `lookup()` routine. In its common use, `lookup()` returns the vnode with its primary vnode lock held. This is done to ensure that no external operation can change the pathname corresponding to a resolved vnode before the calling function has a chance to lock it.

Unfortunately, the `fprotect()` system call could not employ this method, because it may be attempting to lock multiple objects, as described in section 3.2. If `fprotect()` did use this method, lock polling would be impossible, as simply resolving which vnodes were referenced in the call would (individually) wait for each primary vnode lock.

Instead, `fprotect()` calls `lookup()` with a special flag signifying that the vnode should be returned with its primary vnode lock released. This allows `fprotect()` to reference all of the locks on which it will wait when polling. This method, while necessary, introduces a vulnerability window during which a vnode's path could be changed during the lock polling.

This potential inconsistency is mitigated by calling `lookup()` again, with the same path, once the original vnode's primary lock is held. If this second `lookup()` fails or returns a reference to a different vnode, the primary lock of the "stale" vnode is released, and `fprotect()` retries the lock acquisition process from the first `lookup()`.

## 3.4 Copy-On-Write for Rollback

In order to provide atomicity – that is, guarantee that a transaction is either fully completed or effectively never happened – we implemented a rollback mechanism. Our rollback mechanism provides the means with which to undo any change made by a process to a particular vnode.

Due to the lack of a unified buffer cache in OpenBSD, we are forced to forego a more elegant implementation: a unified buffer cache would provide us with a consistent interface to an in-memory cache of vnode contents. If a vnode is marked for copy-on-write (CoW), we could simply install a new mapping for that process that is backed by the original mapping. A WRITE operation causes any affected pages to be copied from the original mapping and new, anonymously-backed pages are mapped in to their places. A unified buffer cache is expected to appear in OpenBSD 3.6 [22]; in other words, in about a year.

We implement CoW as a (partial) vnode layer that is stacked above the file system. To provide CoW for data, we provide alternate versions of the READ and WRITE vnode operations. During a transaction, a WRITE operation causes the buffer and control structures needed to perform the operation to be copied into the kernel and added to a linked list associated with the vnode. We call this a vnode's *write journal*.

A READ operation first checks if there is any overlap in the requested (offset, length) with any entries in the journal and, if there is, copies the overlap into the user buffer. It then fills the remaining holes with the underlying file system's READ operation. This is done in a manner such that the last entry in the journal – which is to say, the last writer – wins. We maintain an ordered, non-overlapping list of ranges covered by the associated journal. Entries in this list refer to the journal entry which it represents. For any overlapping ranges in the journal, only the last writer is referred to by the corresponding range in the ordered list. This list allows us to perform reads more efficiently, since it is ordered and maintains last writers: We can simply walk through the list to find the candidate journal entry for any reads with (offset, length) ranges covering journalled data. Also, the maintenance cost of a write is minimal, since it simply inserts the range it covers into the ordered list, splitting up or removing existing entries which are supersets or subsets, respectively, of that write.

Due to the plethora of metadata operations and the fact that most of these only refer to small pieces of data, we employ a simpler approach. For any metadata-mutating operation, the underlying file system is queried for the metadata and copies it to a buffer associated with the vnode, unless a copy for this metadata already exists. A *metadata journal* is maintained separately. We then perform the operation on the cached copy and append to the metadata journal an entry describing the operation and its parameters. An introspective metadata operation will use the CoW copy of the metadata if it exists, or simply pass the operation through to the underlying file system if it does not. Our ability to do this comes from the fact that the vnode layer is file system-neutral, so we may operate on common representations of metadata.

When a process calls `funprotect()`, the journalled data and metadata are *committed*. That is, the journal entries are traversed in order and the file system operations they describe are performed.

If a failure occurs while the journal is being committed, the file system may be left in an inconsistent state. In order to overcome this, we would need file system support first to perform a set of operations and then to commit the set as a single unit. Exploration of this issue is left for future work.

Another concern with our journalling scheme is excessive memory usage. Journals are not backed by any storage, and cannot get kicked out by memory pressure. Thus, an avenue for a DoS attack is to perform operations that cause a lot of metadata journal activity, thereby exhausting physical resources on the host computer. We employ a simple scheme of aborting a transaction if it consumes more than a preset hard limit of memory. Future work will explore other methods of avoiding this problem, including rate-limiting and providing backed memory for the journals.

### 3.5 Flags and Hints

Another feature of our system is the `flags` parameter that is passed into `fprotect()`. This allows developers to specify options such as whether the parent directory (or any number of parent directories) of a file system object should be locked by the `fprotect()` call; whether child processes can inherit a lock (or an arbitrary subset of the parent's lockset) after a `fork()`; or whether a non-existent, `fprotect()`ed file that is then created becomes visible to the rest of the file system at the point of creation or at the point that it is `funprotect()`ed. These flags are listed in figure **??**.

In demonstrating how our system prevents the `vipw` exploit (see section 2.1), we modified `vipw` so that it encloses its vulnerability window within a `fprotect` transaction. Since `vipw` `fork()`s off a child process (a shell interpreter) that then `fork()`s off its own child process (`vi`), we passed `fprotect()` a flag that grants any child (or grandchild, etc) process proxy access to the locked file system objects. In this manner, `vi` is able to write its changes to the password file even though `vipw` ultimately holds the lock on it.

### 3.6 Deadlock Prevention

Another challenge was in finding an efficient and effective means by which to avoid and/or prevent deadlock. In many cases, OpenBSD's kernel will transparently handle the scheduling decisions relating to sleeping processes that are waiting for locks. Nevertheless, some amount of deadlock management is necessary; therefore, we have integrated a deadlock-prevention technique borrowed from the database and real-time processing communities [19, 20, 12, 24] in which directed dependency graphs are used to identify deadlock conditions.

A deadlock is the result of a process waiting on another process that is either directly waiting on the first process, or indirectly waiting on the first process through other processes.

The deadlock detection system operates by having each process keep a list of all of the other processes upon which it is waiting. Before sleeping, a process

```
OS:   OpenBSD 3.4-current
CPU:  Intel(R) Celeron(TM) CPU 1300MHz ("GenuineIntel" 686-class) 1.31 GHz
MEM:  242307072 (236628K) 133 MHz SDRAM
DISK: wd0 at pciide0 channel 0 drive 0: <QUANTUM FIREBALLlct15 30>
      wd0: 16-sector PIO, LBA, 28629MB, 16383 cyl, 16 head, 63 sec, 58633344 sectors
      wd0(pciide0:0:0): using PIO mode 4, Ultra-DMA mode 4
```

Figure 4: The performance test configuration

*p* checks that each of the processes upon which it depends, either directly or indirectly, is not in turn waiting on it (*p*). If deadlock is unavoidable, the caller's `fprotect()` invocation returns with an error of type EDEADLK.

This deadlock detection system is only used on file system objects that are in an `fprotect` transaction; thus, it is only called before waiting on locksets in the `fprotect()` call, and in the generic OpenBSD VFS locking routine when a process is attempting to access (and thereby acquire the primary vnode lock for) a file system object that is in a fprotect region of another process.

### 3.7 DoS Prevention

An important role of modern operating systems is providing isolation between processes. The deadlock detection system described in section 3.6 handles a certain class of denial of service situations, but does nothing to prevent a process from holding the primary lock of a filesystem object indefinitely, thereby preventing all other processes from accessing that object.

This problem could be addressed by imposing a maximum timeout on transactions, but choosing the proper maximum timeout is quite difficult. The proper maximum timeout would vary from system to system, as it depends on a variety of hardware and software specific settings that are unique to each configuration. Accurately imposing a maximum timeout on transactions is an avenue of further research.

Alternatively, optimistic locking could resolve this issue, as many processes could be in fprotect regions acting on the same object, operating on their own shadow copies of the data and meta-data associated with an object. Then, the only considerations are the memory usage of keeping the associated shadow copies and handling merge semantics. Optimistic locking is discussed in more detail in section 5.

## 4 Performance

The analysis of the `fprotect()` and `funprotect()` system calls were performed on a clean installation of OpenBSD-CURRENT (October 13, 2003), running a generic kernel patched to include the `fprotect` API. This ran on the configuration described in figure 4.

### 4.1 Fprotect System Call

When called, `fprotect()` must ensure that each pathname passed as an argument is not already referenced by the state associated with the calling process. Due to this, the performance of the `fprotect()` system call depends on the number of file system objects currently held in a transaction in the context of the calling process.

After this validation stage the system call continues, acquiring the primary lock for each requested object by polling the locks, as described in section 3.2. During this lock polling procedure, prior to sleeping the deadlock detection routine is called to check the current situation for deadlocks.

It is difficult to analyze the lock polling mechanism, as it is very hard to simulate contention for file system objects accurately – system behavior and file system access patterns vary widely from system to system. Analysis of the deadlock detection system is deferred until section 4.3.

The first time that a process makes an `fprotect()` call, it experiences a significant overhead as data structures for the system are allocated and initialized. With the test configuration (4), the first time a process called `fprotect()`, requesting a lock on one object took 2424 $\mu$s, while the next ten calls (each for one object) took an average of 22.2 $\mu$s. We take this value of 22.2 $\mu$s as the baseline cost of a `fprotect()` call on the test system.

Figure 5 shows the performance of the fprotect system call in relation to the number of active transactions, with one unique file system object passed to each call. The data is taken as an average of ten runs of the

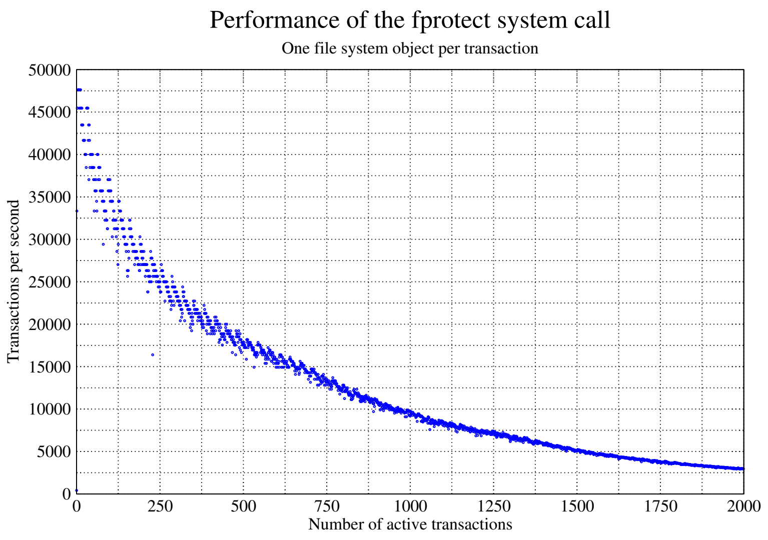

Figure 5: Fprotect() performs optimally when there are less active transactions.

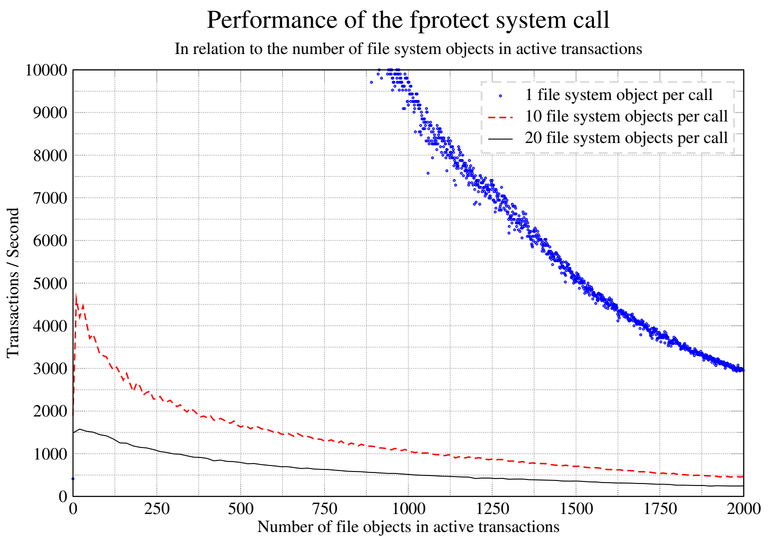

Figure 6: Fprotect() performs optimally when called with relatively few file system object descriptors.

benchmark. This figure shows that the performance of `fprotect()` degrades as more unique files are involved in transactions.

The performance of `fprotect()` is also negatively impacted as more file system object descriptors are passed as parameters to the system call. Figure 6 shows the performance of the `fprotect()` system call in relation to the number of file objects in active transactions, averaged over ten runs on the testing system. Clearly, `fprotect()` performs better under a heavy load when passed one object than ten or twenty objects per transaction.

As transactions have the best chance of success if they are short in duration, we expect that transactions will usually operate on few file system objects. Therefore, the normal behavior of a `fprotect()` call should be similar to the behavior with transactions on one file.

## 4.2 Funprotect System Call

The performance of the `funprotect()` system call also depends on the number of objects held in a transaction by a process, as it must search for the specified transaction and ensure that objects are unprotected in the same sets as they were protected.

Figure 7 shows that the performance of `funprotect()`, called with one file system object

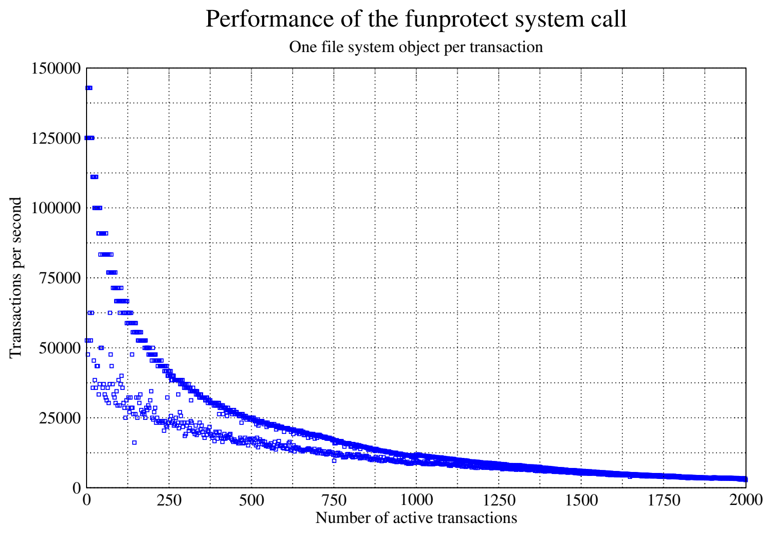

Figure 7: Funprotect() performs optimally when there are less active transactions.

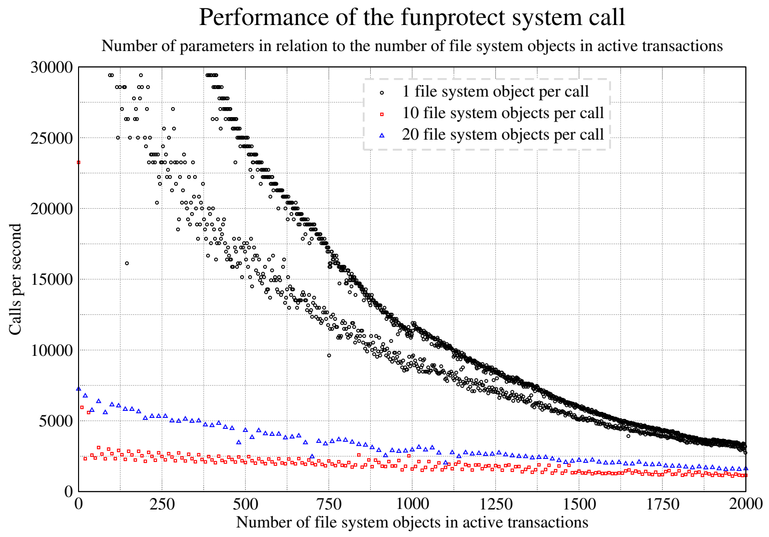

Figure 8: Funprotect() performs optimally when called with relatively few file system object descriptors.

descriptor as a parameter, in relation to the number of active transactions held by the process. These results are averaged over ten runs of the benchmark on the testing system 4.

Figure 8 shows the performance of `funprotect()` in relation to the number of file objects in active transactions, averaged over ten runs on the testing system (4). There are some unexpected results with the `funprotect` performance; For some reason calls referencing twenty objects perform better than calls referencing ten objects. There is also an odd split in the data when `fprotect()` is called with one object per transaction. We are investigating this odd behavior.

Invocations of `funprotect()` perform better when few file system object descriptors are passed as arguments. We expect that the common use of `funprotect()`, like `fprotect()`, to consist of relatively few file system object descriptors as parameters. Therefore, the performance of `funprotect()` with one file per call is representative of the common usage.

## 4.3 Deadlock Detection System

The deadlock detection routine is called in `fprotect()` prior to waiting on a lock, as described in section 4.1. As discussed in section 3.6, the deadlock detection routine makes a set of all processes that the current process waits on, whether directly or indi-

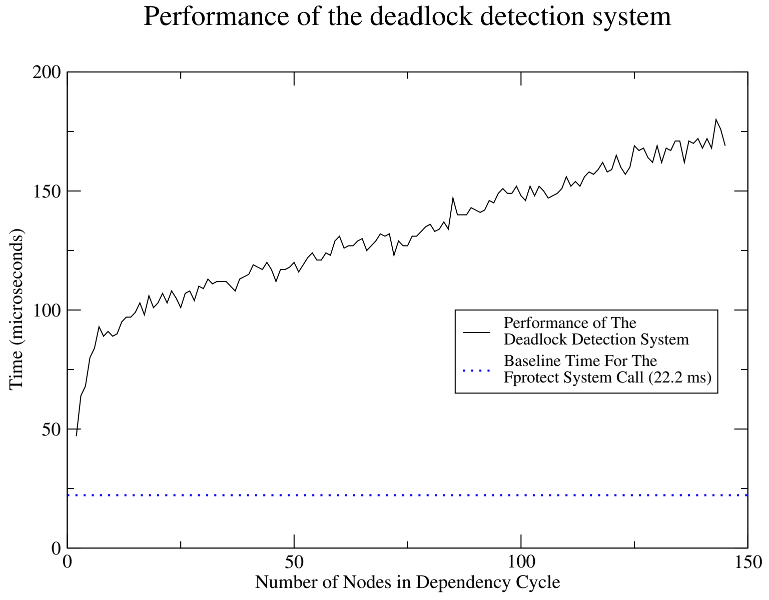

Figure 9: The performance of the deadlock detection system depends on the number of processes involved in a situation where deadlock is possible.

rectly, and makes sure that no processes from that set is waiting on the current process. The system's performance is dependent on the number of processes in a *dependency cycle* – that is, how many processes are involved in a situation where they are all waiting on another. In that case, no process can make progress; they are all deadlocked.

Figure 9 shows the performance of the deadlock detection system in relation to how many processes are involved in a dependency cycle. Since the deadlock detection is done in an `fprotect()` call, each of these data points contains the *baseline* execution time for `fprotect()` of 22.2 $\mu$s, in addition to the time spent in the deadlock detection code. As the figure illustrates, the performance of this functionality degrades linearly, demonstrating good scaling characteristics.

When one hundred processes are involved in a dependency cycle that results in a deadlock, the deadlock detection system takes 148 $\mu$s to determine that a deadlock will occur (and back out of the offending `fprotect()` call. We feel that this is adequate performance: we had to increase the per-user process limits in *login.conf* to be able to run this benchmark.

## 5 Future Work

**Analysis Tools** An interesting avenue of work lies in the automatic insertion of critical sections. There are well established static analysis tools which identify certain classes of potential race conditions in existing applications [4, 23]. Augmenting such a tool to include the ability to insert critical section demarkations by using our interface should be relatively easy.

Additionally, another approach could be taken, utilizing the policies of run-time detection tools [21] to identify potentially dangerous sequences of file system operations and add transaction demarkations to them.

**Optimistic Locking** Currently, the `fprotect` system uses a pessimistic locking strategy; the primary vnode lock is held from the `fprotect()` call until the corresponding `funprotect()` call. This ensures a process exclusive access to an object, as the OpenBSD VFS layer must acquire this lock before every vnode operation. An optimistic locking strategy would allow for more flexibility in the serialization of transactions to produce more optimal schedules. It would also allow us to explore serializable schedules in the context of our system in order to reduce serialization slack even further.

**Vnode Journalling** Our work on copy-on-write sparked our interest in *vnode layer journalling*. That is, a journal describing all operations on a vnode in terms of vnode operations. We would like to explore this idea more as a feature provided by the operating system. For example, allowing applications to implement "undo" functionality using the file system.

## 6 Related Work

The POSIX interface includes mandatory whole file or byte-range locks [17], allowing a mindful programmer to avoid some race conditions. However, there is no infrastructure to guarantee consistency of file system meta data or operations on directories.

Lowery [11] presents a generic format for TOCTTOUs and a broad survey of them in practice. File system object races, as well as database inconsistencies, classloader problems, and networked replay attacks are discussed.

Bishop and Dilger [4] present a formal language description of file system race conditions and a tool for static analysis of executables. This tool detects possible vulnerable situations in sequences of file system operations and can inform the programmer of such events. This differs from our work, as it is only a resource for a programmer, not an API extension, and does not provide a system wide guarantee of consistency. As mentioned in section 5, we will attempt to augment this static analysis tool to utilize our interface.

Anguiano [2] implemented a policy-based static analysis tool that identifies and prevents race conditions of the sort described in section 2.1 and, to a lesser extent, those in section 2.3. The tool is similar to Bishop's and Dilger's [4], but it overcomes some implementation difficulties whereby a long call chain could fool Bishop's and Dilger's tool into missing some race conditions. While Anguiano's tool performs well as a compile-time tool, it misses TOCTTOU race conditions that depend on the runtime environment. Anguiano's tool is another with which we could conceivably integrate our solution.

Cowan et al [9] focus solely on temporary file race conditions with RaceGuard, a Linux kernel enhancement that tracks, prevents, and logs pathological cases. RaceGuard is efficient, accurate (very few false positives), and successful within its domain. However, RaceGuard makes no attempt to solve race conditions that are not based on temporary files. Also, RaceGuard heuristically evaluates the window of vulnerability for race conditions – under certain conditions it will incorrectly assume that a file system object is no longer at risk for race-induced inconsistencies.

Tsyrklevich and Yee [21] implemented a system intended to handle a broader range of race conditions than just those based on temporary files, although the design is not without difficulties. While one can specify custom policies in configuring their "pseudo-transactions", this flexibility is achieved at the expense of correctness guarantees, given that an *a priori* system-wide policy is used. Policy-based approaches necessarily follow the leading edge of known exploits; such a reactionary method cannot anticipate future permutations of the problem.

Another shortcoming of Tsyrklevich's and Yee's system lies in the fact that, because it is implemented as a kernel module that intercepts system calls and then passes them into the kernel, the system actually introduces its own race [10]: there could be a context switch between the interception of a system call and the time it is passed into the kernel. The system also suffers from its heuristic assessment of the duration of vulnerability to race conditions, which is based solely on an ad hoc span of time with the system's load average loosely included. As with RaceGuard, there is no definite guarantee of the atomicity of race-prone combinations of file system object operations.

The Alpine file system [6] is a networked file system that provides transactional guarantees. Alpine is intended for direct use by applications via an RPC interface, and is not fitted to a generic VFS interface. This allows Alpine to make fundamental design decisions in favor of transactional operations, whereas our work focuses on retrofitting an existing production system and VFS with the ability to perform transactions.

Other relevant work is found in [20, 12, 24, 13].

## 7 Summary and Conclusion

File system race conditions are a subtle, dangerous class of security exploit. They have received relatively little mainstream attention and, thus, have not yet been appropriately addressed. Moreover, as successful defenses against popular buffer overflow-based code injection exploits become more widely adopted, attackers will likely shift their tactics. To confront these race condition exploits head-on, we present `fprotect`: a slight augmentation to OpenBSD's VFS, as well as a limited extension to the POSIX file system interface that allows application programmers to add transaction semantics to their file system operations.

Our system provides *atomicity*, *consistency* and *isolation* over an arbitrary unit of file system operations. It avoids deadlock, is capable of safely aborting a transaction, integrates seamlessly with legacy applications, and demonstrates good performance characteristics.

Our updated work and related information is available at `http://www.citi.umich.edu/projects/fprotect`.

## 8 Acknowledgements

The authors would like to thank Jason Flinn, Peter Honeyman, Olga Kornievskaia, Joshua Marker, and Rick Wash for reviews and suggestions.

## References

[1] OpenBSD: The proactively secure Unix-like operating system. `http://www.openbsd.org/`.

[2] Ricard Anguiano. A Static Analysis Technique for the Detection of TOCTTOU Vulnerabilities. Master Thesis, University of California Davis, 2001.

[3] CR Attanasio. Virtual Machines and Data Security. In ACM Proceedings of the Workshop on Virtual Computer Systems, 1973.

[4] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. Department of Computer Science, University of California at Davis Technical Report CSE-95-10, October 1995.

[5] Scott Blake. Protecting the Network Neighborhood. `http://www.securitymanagement.com/library/000833.html`, December 2001.

[6] M.R. Brown, K.N. Kolling, and E.A. Taft. The Alpine file system. *ACM Transactions on Computer Systems*, 3(4):261–293, November 1985.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[8] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. 12 USENIX Security Conference*, pages 91–104, Washington DC, aug 2003.

[9] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[10] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Solutions. February 2003.

[11] J. Craig Lowery. A Tour of TOCTTOUs. SANS GSEC practical v.1.4b, August 2002.

[12] K. Akesson M. Tittus. Deadlock Avoidance in Batch Processes. IFAC World Congress, China, August 1999.

[13] Joshua P. MacDonald. File system support for application-level transactions. Department of Electrical Engineering and Computer Science, University of California at Berkeley.

[14] Ronald L. Mendell. Matching Wits Against Bits. `http://www.securitymanagement.com/library/000675.html`, December 2001.

[15] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *Proceedings of the 1999 USENIX Technical Conference, FREENIX track*, June 1999.

[16] Kristen Noakes-Fry. Unmasking Social-Engineering Attacks. `http://security2.gartner.com/story.php.id.38.s.1.jsp`, December 2001.

[17] The Institute of Electrical and Inc. Electronics Engineers. IEEE Std 1003.1-1990 ("POSIX"), 1990.

[18] W. Purczynski. rm - recursive directory removal race condition. bug-fileutils mailing list, March 2002.

[19] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2002. `http://www.cs.wisc.edu/~dbbook/`.

[20] S. Reveliotis and P. Ferreira. Deadlock avoidance policies for automated manufacturing cells. IEEE Trans. on Robotics & Automation, 12:845–857, 1996.

[21] Eugene Tsyrklevich and Bennet Yee. Dynamic Detection and Prevention of Race Conditions in File Accesses. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[22] Ted Unangst. *Personal communication*.

[23] D. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder`.

[24] Detlef Zimmer. A Locking Algorithm with Premature Unlocks. CADLAB at Paderborn, Germany, 1992.