Modeling and Preventing TOCTTOU Vulnerabilities in Unix-Style File Systems

Jinpeng Wei[1] and Calton Pu[2]
[1]Florida International University, 11200 SW 8th Street, Miami, FL, 33199 USA, weijp@cs.fiu.edu
[2]Georgia Institute of Technology, 266 Ferst Dr, Atlanta, GA 30332 USA, calton@cc.gatech.edu

Abstract

TOCTTOU (Time-of-Check-To-Time-Of-Use) is a file-based race condition in Unix-style systems and characterized by a pair of file object access by a vulnerable program: a *check* operation establishes certain condition about the file object (e.g., the file exists), followed by a *use* operation that assumes that the established condition still holds. Due to the lack of support for transactions in Unix-style file systems, an attacker can modify the established file condition in-between the *check* and *use* steps, thus causing significant harm. In this paper, we present a model of the TOCTTOU problem (called STEM), which enumerates all the potential file system call pairs (called exploitable TOCTTOU pairs) that form the check/use steps. The model shows that a successful TOCTTOU attack requires a change in the mapping of pathname to logical disk blocks between the check and use steps. We apply STEM to POSIX and Linux to demonstrate its practical value for Unix-style file systems. Then we propose a defense mechanism (called EDGI) that prevents an attacker from tampering with the file condition between exploitable TOCTTOU pairs during a vulnerable program's execution. EDGI works at the file system level and does not require existing applications to change. We have implemented EDGI on Linux kernel 2.4.28 and our evaluation shows that EDGI is effective and incurs little overhead to application benchmarks such as Andrew and Postmark.

Key words: race condition; TOCTTOU; vulnerabilities; invariant; modeling; defense; kernel

1 Introduction

TOCTTOU (Time-Of-Check-To-Time-Of-Use) is a well known security problem [1, 2]. An illustrative example is *sendmail*, which used to check for a specific attribute of a mailbox file (e.g., it is not a symbolic link) before appending new messages. However, the checking and appending operations do not form an atomic unit. Consequently, if an attacker (the mailbox owner) is able to replace his mailbox file with a symbolic link to /etc/passwd between the checking and appending steps by *sendmail*, then he may trick *sendmail* into appending emails to /etc/passwd. As a result, an attack message consisting of a syntactically correct /etc/passwd entry with root access would give the attacker root access. TOCTTOU is a serious threat: A search of the U.S. national vulnerability database [3] for the keywords "symlink attack" returns more the 600 hits .Our survey of the CERT advisories [4] from 2000 to 2004 discovered 20 TOCTTOU vulnerabilities; in 11 out of these 20 cases, the attacker was able to gain unauthorized *root* access. These 20 advisories cover a wide range of applications from system management tools (e.g., /bin/sh, *shar*, *tripwire*) to user level applications (e.g., *gpm*, Netscape browser), and they affected many operating systems including: Caldera, Conectiva, Debian, FreeBSD, HP-UX, Immunix, MandrakeSoft, RedHat, Sun Solaris, and SuSE. A similar list of vulnerable applications reported by the BUGTRAQ mailing list [5] is shown in Table 1. TOCTTOU vulnerabilities are widespread and cause serious consequences.

The *sendmail* example shows the structural complexity of a TOCTTOU attack, which requires (unintended) shared access to a file by the attacker and the victim (the *sendmail*), plus the two distinct steps (check and use) in the victim. This complexity, plus the non-deterministic nature of TOCTTOU attacks, makes the detection difficult. For example, TOCTTOU attacks usually result in escalation of privileges, but no immediately recognizable damage. Furthermore,

TOCTTOU attacks are inherently non-deterministic and not easily reproducible, making post mortem analysis also difficult. These difficulties are illustrated by the TOCTTOU vulnerabilities found in *vi* and *emacs* [6], which appear to have been in place since the time those venerable programs were created.

Although general TOCTTOU problems are not limited to file access [7], in this paper we focus on file-related TOCTTOU problems. Our first contribution is an abstract model of such TOCTTOU problems (called STEM – **S**tateful **T**OCTTOU **E**numeration **M**odel) that captures all potential vulnerabilities. The model is based on two mutually exclusive *invariants:* a file object either does not exist, or it exists and is mapped to a logical disk block. For each file object, one of these invariants must remain true between the check and use steps of every program. Otherwise, potential TOCTTOU vulnerabilities arise. This model allows us to enumerate all the file system call pairs of check and use (called exploitable TOCTTOU pairs), between which the invariants may be violated. A protection mechanism derives from this model and maintains the invariants across all the exploitable TOCTTOU pairs, by preventing access from other concurrent processes/users that may change the invariants. The practical value of STEM is demonstrated by the mapping of concrete Unix-style file systems to it. We have exhaustively analyzed the file system calls of POSIX and Linux and classified them according to the STEM model. From this classification we enumerated all the exploitable TOCTTOU pairs for POSIX (485 pairs) and Linux (224 pairs).

The second contribution of this paper is an event-driven defense mechanism, the Event Driven Guarding of Invariants (EDGI), a mechanism based on the STEM model, for preventing exploitation of TOCTTOU vulnerabilities. The EDGI defense has several advantages over previously proposed solutions. First, based on the STEM model, EDGI is a systematically

developed defense mechanism with careful design (using Event-Condition-Action (ECA) rules) and implementation. Assuming the completeness of the STEM model, EDGI can stop all TOCTTOU attacks. Second, with explicit modeling of users, EDGI is able to prevent potential abuse of the defense mechanism by malicious users, thus overcoming the limitation of existing solutions such as pseudo-transactions [8] and RPS [9]. Third, it does not require changes to applications or file system API. Fourth, our implementation on Linux kernel and its experimental evaluation show that EDGI carries little overhead.

The rest of the paper is organized as follows. Section 2 introduces the Abstract File System, on which the STEM model is defined. Section 3 describes the STEM model, defines the TOCTTOU problem, and enumerates the TOCTTOU pairs in the Abstract File System. Section 4 applies the STEM model to concrete file systems such as POSIX and Linux. Section 5 presents the EDGI defense mechanism against TOCTTOU attacks. Section 6 describes a Linux implementation of EDGI and Section 7 presents an experimental evaluation of EDGI. Section 8 outlines related work and Section 9 concludes the paper.

2 The Abstract File System

Due to the complexity of the TOCTTOU problem in real file systems, in this section we define a simplified Abstract File System (AbsFS), on which we define the TOCTTOU problem (see Section 3) and design a defense mechanism (see Section 5). In Section 4 we map concrete file systems (POSIX and Linux) to AbsFS and translate the results from the AbsFS to the concrete file systems.

2.1 Definition of Abstract File System

The Abstract File System (AbsFS) manages a set of file system (FS) objects. Each file system object consists of a pathname, an ordered set of logical disk blocks, and a mapping of the pathname to the corresponding set of logical disk blocks.

An AbsFS pathname *f* has the form $/d_1/d_2/.../d_n/e$ in which the first "/" represents the *root* directory of the file system, d_i $(1 \le i \le n)$ is a directory or a link to a directory, and *e* can be a regular file, a link, or a directory. For simplicity of presentation, we use *f* to denote the entire pathname $/d_1/d_2/.../d_n/e$ in most of the discussion.

For simplicity we also assume the AbsFS to contain only contiguous files, i.e., the set of logical disk blocks is sequential for every file, and the AbsFS only needs to map the pathname to the address (block number) of the initial logical disk block. Let F denote the set of all pathnames and B denote the set of all logical disk blocks, the pathname mapping function **resolve** is defined as:

$resolve: F \to B \cup \{\varnothing\}, \varnothing \notin \mathrm{B}$.

Given a pathname $f \in F$, if the AbsFS object corresponding to *f* exists, with the initial logical disk block number $b \in B$, then we define $resolve(f) = b$. If the AbsFS object corresponding to *f* does not exist, we define $resolve(f) = \varnothing$.

The AbsFS defines an Application Programming Interface consisting of 4 operations on file objects.

Definition 1: *creation(pathname)* is the operation that creates new FS objects in the AbsFS by changing the mapping for pathname *f* from $resolve(f) = \varnothing$ to $resolve(f) = b$, for some $b \in B$.

Definition 2: *removal(pathname)* is the operation that changes the mapping for pathname *f* from $resolve(f) = b$ to $resolve(f) = \varnothing$.

Definition 3: *normal use(pathname)* is the operation that works on an existing file system object and does not remove it.

Definition 4: *check(pathname)* is the operation that returns a predicate about the named FS object. The predicate may be $resolve(f) = b$ or $resolve(f) = \varnothing$. The file *f* has to be in one of these two states.

An application uses the *creation* operation to create a new FS object, the *check* operation to determine the invariant $resolve(f) = b$ or $resolve(f) = \varnothing$, the *normal use* operation to read or write the FS object, and the *removal* operation to delete an FS object. Currently, these four kinds of operations (*creation*, *normal use*, *removal*, and *check*) are defined as AbsFS operations. The *creation* and *removal* operations change the **resolve** mapping, while the *check* and *normal use* operations do not change the **resolve** mapping. The AbsFS operations and FS object states can be represented in a state transition diagram shown in Figure 1.

2.2 Concurrent Access to AbsFS

Since the TOCTTOU vulnerability happens with concurrent access by a victim process and an attacker process, we extend the notation above to include explicit modeling of concurrent file system object access.

Definition 5: Safe sequence of AbsFS operations. Given a sequence *O* of AbsFS operations invoked by a process/user on FS object *f*, $O(f) = o_1(f), o_2(f), ..., o_n(f)$, $n > 1$, if $\forall i, 1 \le i \le n-1$, $resolve(f)$ remains an invariant between the end of $o_i(f)$ and the beginning of $o_{i+1}(f)$, we say the sequence $O(f)$ is a *safe* sequence of AbsFS operations, with regard to concurrent access. Since in most cases all the operations in the sequence belong to the same process/user, for notational

simplicity, we omit the process/user id from the sequence. In case of interleaved operations, we will add a superscript to denote the different processes/users.

It is straightforward to see that the exclusive access by a single process to files is safe, i.e., the state of each FS object persists from the end of each AbsFS operation to the beginning of the next AbsFS operation under exclusive access.

Definition 6: Unsafe sequence of AbsFS operations: Given a sequence of operations $O(f) = o_1(f), o_2(f), ..., o_n(f)$, $n > 1$, if $\exists i, 1 \le i \le n-1$, $resolve(f)$ is not invariant between the end of $o_i(f)$ and the beginning of $o_{i+1}(f)$, i.e., $resolve_{o_i}(f) \neq resolve_{o_{i+1}}(f)$, $O(f)$ is an *unsafe* sequence of AbsFS operations.

3 STEM Model of TOCTTOU

3.1 Exclusion of Careless Programming

Before we start the discussion of the TOCTTOU problem, we point out that the TOCTTOU vulnerability is not due to a naively careless programming style. Consider the *sendmail* example. Hypothetically, the *sendmail* could simply open the file name that is the user's mailbox by naming convention (e.g., /usr/mail/username) and then append emails to that file. This simplistic approach fails immediately because the naming convention may or may not hold for all names (e.g., a user may have created a symbolic link from /usr/mail/username to /etc/passwd). To avoid this kind of problems, many system programmers have adopted a more careful programming style. In case of files, this careful programming style establishes a predicate on the file before using it. For example, *sendmail* establishes the predicate $resolve(f) = b$, where b belongs to a regular file, not a symbolic link, before appending messages to f. The predicate $resolve(f) = b$ is an invariant that should remain true as long as the *sendmail* keeps appending messages. We call the predicate an *invariant* instead of pre-condition, because the normal

connotation of pre-condition is that it must be true before entering a function, but it may become false after the function has started. In contrast, our invariant must remain true through the duration of file usage.

In the rest of this paper we exclude the careless programming style and assume that all system utilities of interest will establish an invariant on a pathname before using it. This is represented in our notation by dividing a sequence of AbsFS operations $O(f) = o_1(f),...,o_i(f),o_{i+1}(f),...,o_n(f)$ into two subsequences. The first subsequence $o_1(f),...,o_i(f)$ is called the "Check" part, and the second subsequence $o_{i+1}(f),...,o_n(f)$ is called the "Use" part. The "Check" part establishes the invariant $resolve_{o_i}(f)$ and the "Use" part of the sequence relies on the invariant remaining true, i.e., $O(f)$ is a safe sequence of AbsFS operations.

3.2 TOCTTOU Attacks in AbsFS

Definition 7: A TOCTTOU (Time-Of-Check-To-Time-Of-Use) attack on file object *f* consists of two concurrent processes, victim *v* and attacker *a*, with interleaved AbsFS operations that make *v*'s sequence unsafe. Consider the victim *v* executing the sequence $O^v(f) = o_1^v(f),...,o_i^v(f),o_{i+1}^v(f)...,o_n^v(f)$, divided into the "Check" and "Use" parts. Concurrent with *v*, attacker *a* is able to change the mapping $resolve_{o_i}(f)$ established by *v* during the execution of the sequence $O^v(f)$, transforming it into an unsafe sequence. This is achieved by inserting the sequence $O^a(f) = o_1^a(f),o_2^a(f),...,o_k^a(f)$ between the "Check" and "Use" parts of $O^v(f)$. The result becomes: $o_1^v(f),...,o_i^v(f),o_1^a(f),o_2^a(f),...,o_k^a(f),o_{i+1}^v(f)...,o_n^v(f)$.

To illustrate the definition with concrete scenario, we temporarily move from AbsFS to a Unix-style file system environment. In such an environment, the function *resolve(f)*, in which *f* $= /d_1/d_2/.../d_n/e$, depends on the name to disk block mappings for *all* elements along *f*, since

the pathname resolution needs to start from the root directory and use each intermediate directory name d_i to locate the next level on the directory tree, until *e* is located. Therefore, the final mapping for *f*, $resolve(f) = b$ or $resolve(f) = \varnothing$, implicitly depends on the mappings for all pathname elements along *f*.

Suppose the invariant established by *v* is $resolve_{o_i}(f) = b$, there can be two possible attack sequences $O^a(f)$ of *a*: the first case modifies the last element *e* of *f*, and the second case modifies some element on the pathname ahead of *e*, i.e., d_i where $1 \le i \le n$. In the first case, an attack would remove *f* and then create a symbolic link named *f* which points to another file object *t* ($resolve(t) = b', b \ne b'$), resulting in $resolve^a_{o_k}(f) = b'$. In the second case, an attack would change the target directory of a symbolic link d_i from td_i to some other directory $td_i^{'}$, where both $td_i / d_{i+1} / ... / d_n / e$ and $td_i^{'} / d_{i+1} / ... / d_n / e$ are valid file pathnames. Again this can only be achieved by first removing d_i and then creating a symbolic link named d_i which points to $td_i^{'}$. Here we only worry about symbolic links for the following reasons: (1) d_i cannot be a hard link because hard links to directories are not allowed in Unix style file systems; and (2) if d_i is a regular directory, it cannot be removed unless it is empty. Since we know that d_i contains at least one element *e*, d_i cannot be removed without first removing *e* – then it reduces to the first type of attack.

If the invariant established by *v* is $resolve_{o_i}(f) = \varnothing$, a possible attack sequence $O^a(f)$ is to create the file object *f*, making $resolve^a_{o_k}(f) \ne \varnothing$. Note that an attack that switches symbolic links to directories also works: the trick is to prepare $td_i^{'} / d_{i+1} / ... / d_n / e$ first, then modify d_i so it points to $td_i^{'}$, where $td_i / d_{i+1} / ... / d_n / e$ still does not exist. However, there is one more possibility:

the attack can remove d_n as a regular directory if it is empty and then create a symbolic link under the same name (i.e., d_n) which points to $td_n^{'}$ that contains a file object named *e*.

The TOCTTOU attack is successful if $resolve_{o_i}^{v}(f) \neq resolve_{o_k}^{a}(f)$ and victim *v* continues execution without realizing the invariant created by the subsequence $o_1^v(f),...,o_i^v(f)$ (the "Check" part) has been violated. Consequently, the subsequence $o_{i+1}^v(f)...,o_n^v(f)$ (the "Use" part) will execute under the assumption of the original invariant, which is no longer true.

The side effect of *v* executing the "Use" subsequence $o_{i+1}^v(f)...,o_n^v(f)$ after a successful TOCTTOU attack is that *v* is actually working on some other unintended file object. For example, if *t* = /etc/passwd in the *sendmail* example, emails may be appended to /etc/passwd.

Proposition 1: Violation of an invariant is a necessary condition for a successful TOCTTOU attack.

The proposition 1 follows from Definition 7. If there is no violation of invariants, the sequence $O^v(f)$ is a safe sequence, so there would be no TOCTTOU attack. Consequently, through the entire duration of $O^v(f)$, we can prevent TOCTTOU attacks by preserving the invariant established by $O^v(f)$ and making the sequence a safe sequence.

3.3 An Enumeration of TOCTTOU pairs

Definition 8: Consider an unsafe sequence of AbsFS operations $O(f) = o_1(f),o_2(f),...,o_n(f)$, where $resolve_{o_i}(f) \neq resolve_{o_{i+1}}(f)$. The two operations surrounding the violation of the original invariant (the last operation of the "Check" part and the first operation of the "Use" part), $o_i(f)$ and $o_{i+1}(f)$, are called a *TOCTTOU pair*.

It is useful to identify the TOCTTOU pairs explicitly, since the combinations that yield such pairs are non-trivial but manageable. The diagram in Figure 1 shows all the AbsFS operations and the two states in which a file may be. On the left side of diagram is the *non-existent* state, denoted by $resolve(f) = \varnothing$ and on the right side of the diagram is the *existent* state, denoted by $resolve(f) = b$.

Let us consider first the *non-existent* state and the invariant $resolve(f) = \varnothing$. The first term of a TOCTTOU pair is an operation that results in the *non-existent* state of *f*. From the state transition diagram in Figure 1, we see that two operations lead to the *non-existent* state: {*check, removal*}. The *removal* operation explicitly makes *f* non-existent, while the *check* operation also ends in the *non-existent* state if it does not find the pathname mapping. The second term of the TOCTTOU pair is an operation that starts from the invariant $resolve(f) = \varnothing$ (the *non-existent* state). The two operations that start from the *non-existent* state are: {*check, creation*}. Therefore, the TOCTTOU pairs associated with the *non-existent* state are contained in the set produced by the Cartesian product of {*check, removal*}×{*check, creation*}.

While the Cartesian product contains all the TOCTTOU pairs, we will refine the second term, which corresponds to the "Use" part of the TOCTTOU pair. For an attacker to exploit a TOCTTOU vulnerability for some gain (e.g., escalation of privileges), the victim must be tricked into doing something useful for the attacker in the "Use" part. Examples of useful actions are: (1) set or modify the status information of an existing file object (e.g. make /etc/passwd world-writable); (2) modify the runtime environment of the victim application (e.g. change the current directory); and (3) release the content of a sensitive file object (e.g. read the content of /etc/shadow into memory). Since the *check* operation does not produce any useful results for the

attacker, we define *exploitable* TOCTTOU pairs by eliminating the *check* operation from the second term of TOCTTOU pairs.

Now we consider the *existent* state of *f*, characterized by the invariant $resolve(f) = b$. The state transition diagram in Figure 1 shows that the set of operations that lead into the *existent* state is {*creation, check, normal use*}, and the set of operations that start from the *existent* state is {*check, normal use, removal*}. So the TOCTTOU pairs associated with this invariant are in the set {*creation, check, normal use*}×{*check, normal use, removal*}. As a second term of the TOCTTOU pairs, *check* will not produce useful results for the attacker. Consequently, we also eliminate *check* from the list of exploitable TOCTTOU pairs.

By deleting *check* from the second terms, the exploitable TOCTTOU pairs are {*check, removal*}×{*creation*} for the first invariant and {*creation, check, normal use*}×{*normal use, removal*} for the second invariant. Since there are only two invariants in AbsFS, we have enumerated all the exploitable TOCTTOU pairs in Table 2.

Proposition 2: The enumeration of TOCTTOU pairs in Table 2 is complete, i.e., it contains all the exploitable TOCTTOU pairs in AbsFS.

Proof: according to Definition 8, for a TOCTTOU pair $o_i(f)$ and $o_{i+1}(f)$, *resolve(f)* should remain an invariant between the end of $o_i(f)$ and the beginning of $o_{i+1}(f)$. Mapping this to Figure 1, this means that the edge representing $o_i(f)$ should arrive in a state that the edge representing $o_{i+1}(f)$ leaves from. Therefore, the enumeration of TOCTTOU pairs reduces to the problem of enumerating all combinations of one in-edge and one out-edge at each state in Figure 1. More formally, the set of TOCTTOU pairs is $\bigcup_s (in(s) \times out(s))$, where *s* is a state in Figure 1, *in(s)* is the set of AbsFS operations that end in *s*, and *out(s)* is the set of operations that begin in *s*. Table

2 is the concrete instantiation of this formula, with the two invariants being the two states in Figure 1 and by taking the *check* operation out of *out(s)* (because it does not produce any useful results for the attacker).

3.4 Prevention of TOCTTOU Attacks

In the rest of this section, we will focus on the preservation of invariants across the exploitable TOCTTOU pairs. This protection will be done in two steps. First, we will maintain explicitly the invariant for each file object on behalf of a certain user (called the *holder* of the invariant). Second, for every file system operation that may change the invariant, we check whether the invoker of the operation is the holder. The operation is allowed if it's invoked by the holder. It is disallowed if it belongs to another process/user.

In Figure 1, we described the state transitions of a file with a single process/user. Figure 2 shows the state transitions of a file under concurrent access by multiple processes/users. Without loss of generality, we adopt the policy that the first process/user accessing the file object becomes the invariant holder. (Intuitively, we consider the invariant as an exclusive lock.) The goal of our protection mechanism is to reject any changes to the invariant except by the invariant holder.

The main difference between Figure 1 and Figure 2 is the addition of three states. Two of the states (on the top part of Figure 2) are due to the explicit representation of the cases of invariants with a holder (same as Figure 1) and without a holder (new states). These transitions are allowed, since the pathname is free and the invariant holder is not in competition with any other process/user. The third new state is at the bottom of Figure 2, representing a potential attack since those transitions would change the invariant for the holder. These transitions are rejected

as an error. The original invariant holder maintains the hold on the invariant and the invariant remains unchanged.

The implementation of invariant holder lock relies on a lock table and maps the invariant holder id to the invariant across all TOCTTOU pairs. Consider a TOCTTOU pair $< o_1, o_2 >$. When a process *u* accesses a pathname *f* through $o_1(f)$, *u* becomes the invariant holder, moving from the top states of Figure 2 to one of the middle states. (Note that all four AbsFS operations are allowed in this step.) Our protection mechanism uses the lock table to remember this invariant/holder mapping. The lock is released when the invariant holder process ends. These state transitions are denoted as *exit(u)*, in which case *u* releases the invariant.

While the pathname *f* is in one of the middle states, with invariant holder *u*, another process/user (*u'*) may attempt to change the invariant, which will result in "error". Other operations that do not affect the invariant (e.g., *check* and *normal use*) are allowed, as shown in Figure 2. Thus this mechanism implements the assumption required in Proposition 2 to protect the invariants across TOCTTOU pairs.

Changes to the invariant: as discussed in Section 3.2, there are multiple ways to change the invariant $resolve(f) = \varnothing$ or $resolve(f) = b$: (1) changing the end point of the pathname (i.e., *e* of $f = /d_1/d_2/.../d_n/e$); and (2) changing intermediate path components (e.g., symbolic links to directories). For clarity, Figure 2 only depicts attacks against the end point.

For practical purposes, we note that our protection mechanism does not require explicit request and release of invariant-related locks. The management of invariant locks can be done automatically on behalf of applications. Furthermore, the implementation can be simplified with the following proposition.

Proposition 3: Blocking the *creation* and *removal* of a file object *f* across a sequence $o_1(f), o_2(f), ..., o_n(f)$ is sufficient to make the sequence safe. Specifically, for the file object $f = /d_1/d_2/.../d_n/e$, this blocking forbids the *creation* or *removal* of *e* and the *removal* of symbolic links d_i and regular directory d_n (Section 3.2).

By Definition 5, a sequence of execution $o_1(f), o_2(f), ..., o_n(f)$ is safe if $\forall i, 1 \le i \le n-1$, $resolve(f)$ is an invariant between the end of $o_i(f)$ and the beginning of $o_{i+1}(f)$. If we forbid *creation* or *removal* of any element of *f* (as described in Proposition 3) across $o_1(f), o_2(f), ..., o_n(f)$, we forbid such operations between $o_i(f)$ and $o_{i+1}(f)$, and since we have blocked all possible operations that can change $resolve(f)$, $resolve(f)$ must be an invariant between the end of $o_i(f)$ and the beginning of $o_{i+1}(f)$. So $o_1(f), o_2(f), ..., o_n(f)$ is guaranteed to be a safe sequence of execution.

This proposition is the basis for the EDGI defense described in Section 5.

Proposition 4: Making all exploitable TOCTTOU pairs safe is sufficient to make all file access sequences safe and prevent TOCTTOU attacks.

Proof: Proposition 3 shows the preservation of invariants through a file operation sequence suffices in making the sequence safe. Proposition 2 shows that all exploitable TOCTTOU pairs have been enumerated. Combining the two propositions we have the assurance that making all file operation sequences safe (for each process/user) can prevent all TOCTTOU vulnerabilities from being exploited.

4 Concrete File System Examples

4.1 Exclusion of Careless File Attribute Settings

The AbsFS contains a simplified model of file system objects, with a very simple mapping of pathname to logical disk blocks, without any additional file system attributes such as access

control. In concrete file systems, appropriate access control attributes need to be set to prevent trivial unauthorized file access. For example, Unix files with world writable settings can be easily exploited by many kinds of attacks. In our modeling and analysis of TOCTTOU attacks, we assume that appropriate file access control settings are being used by careful system administrators.

4.2 Unix-Style File Systems

Table 2 gives a complete list of TOCTTOU pairs in AbsFS. Now we map the AbsFS into Unix-style file systems. The first observation in the mapping is that Unix-style file systems have several kinds of file system objects: regular files, directories, and links. The second observation is that the abstract operations of *check, creation, normal use,* and *removal* may be implemented by several system calls. Therefore, we map these abstract operations into sets of system calls (CreationSet, NormalUseSet, RemovalSet and CheckSet) and divide these sets into operations on each kind of file system objects.

- CreationSet = FileCreationSet ∪ DirCreationSet ∪ LinkCreationSet
- NormalUseSet[1] = FileNormalUseSet ∪ DirNormalUseSet
- RemovalSet = FileRemovalSet ∪ LinkRemovalSet ∪ DirRemovalSet
- CheckSet = FileCheckSet ∪ LinkCheckSet ∪ DirCheckSet

The third observation is that the *removal* operation in Unix-style file systems does not produce any useful results for the attacker. This is because in Unix-style file systems, under the assumption of careful file attribute settings (Section 4.1), there are only two ways for the attacker to make $resolve(f) = resolve(t)$ in a TOCTTOU attack (t is an existing security sensitive file object

[1] On Unix-style file systems, the normal use of a link (symbolic or hard) is actually on the regular file or directory that the link refers to, so we do not need to define a separate NormalUseSet for link.

such as /etc/passwd and *f* is the file object accessed by a TOCTTOU pair $< o_1, o_2 >$ in the victim application): via symbolic link or hard link. If the attacker replaces *f* with a symbolic link to *t*, then the victim's *removal* operation on *f* only removes *f* itself, but not *t*; If the attacker replaces *f* with a hard link to *t*, this will increase the number of hard links of *t* by 1, and when the victim performs the *removal* operation on *f*, it decreases the number of hard links of *t* by 1 (restores the original hard link number of *t*, but never decreases it). Since *t* is physically removed only when its hard link number becomes 0, given *t*'s initial hard link number is nonzero, the attacker cannot cause *t* to be removed.

Thus for Unix-style file systems we can eliminate those TOCTTOU pairs with *removal* as the second term from Table 2. The remaining AbsFS TOCTTOU pairs can be mapped to Unix-style file systems as shown in Table 3. For an actual file system, we can map the actual file system calls to these sets to obtain the concrete TOCTTOU pairs.

4.3 Study of POSIX and Linux

We focus on POSIX [10] and Linux as representative examples of Unix-style file systems with TOCTTOU vulnerabilities. We believe the same mapping can be done with the other flavors of Unix file systems. The POSIX mapping is shown in Figure 3 and the Linux mapping is shown in Figure 4. Compare Figure 4 to Figure 3 we can see that the sets are almost the same due to the fact that Linux is POSIX-compliant. We do see some discrepancy though, notably the FileNormalUseSet. For example, POSIX has 6 different system calls on executing a program (**execl, execle, execlp, execv, execve, execvp**), but Linux only has one (**execve**). A closer look at the Linux implementation reveals that Linux implements only **execve** as a system call and uses library calls to implement the remaining 5 POSIX interfaces, which are different wrappers on top of this basic system call.

Applying the mapping of Figure 3 to the mapping in Table 3, we have identified 485 exploitable TOCTTOU pairs for POSIX. Similarly, by applying Figure 4 to the mapping in Table 3, we get 224 exploitable TOCTTOU pairs for Linux.

Proposition 5: If the classification of a concrete file system's operations is complete, then the enumeration of exploitable TOCTTOU pairs is complete for the concrete file system. By complete we mean the classification contains all the concrete file system calls that operate on file objects, and all the concrete file system calls are classified into *check*, *creation*, *normal use*, and *removal* functions on the file objects. (File system calls that have multiple functions appear in multiple categories.)

Proof: The Proposition 2 guarantees the completeness of exploitable TOCTTOU pairs for the AbsFS. Assuming that we have exhaustively analyzed the concrete file system calls and classified them, Proposition 5 follows from Proposition 2.

By exhaustively analyzing the POSIX file system calls (Figure 3), we can apply Proposition 5 to the enumeration of exploitable TOCTTOU pairs based on Table 3 and Figure 3 and conclude that we have enumerated all the exploitable TOCTTOU pairs in POSIX. Analogously, we apply Proposition 5 to the enumeration of exploitable TOCTTOU pairs in Linux, based on Table 3 and Figure 4, and the result is in Table 4.

4.4 Example of TOCTTOU Attacks

Table 5 shows some existing TOCTTOU vulnerabilities of applications running on Unix-style Systems. The TOCTTOU pairs appear in the second column and their associated invariants in the third column. Two of the under examination applications, the *sendmail* and the *logwatch* 2.1.1, are described in detail in the next paragraphs.

Sendmail. In *sendmail*, the TOCTTOU vulnerability is a **<stat, open>** pair, the invariant is $resolve(umbox) = b$, and the attack is first removing *umbox* and second creating a symbolic link under the name *umbox.*

Logwatch 2.1.1. *logwatch* is an open-source script for monitoring log files in Linux. *logwatch* 2.1.1 running as root was reported [11] to allow a local attacker to gain elevated privileges, e.g., write access to /etc/passwd. This attack consists of the following steps:

1. Get the running process ID {pid} of *logwatch*;
2. Create a temporary directory named /tmp/logwatch.{pid};
3. Create a symbolic link with a specific name in the temporary directory, which points to /etc/log.d/scripts/logfiles/samba/`cd etc;chmod 666 passwd #`
4. Wait for *logwatch* to use the temporary symbolic link. Although *logwatch* only opens it for writing, the tricky file name causes the shell to execute it as a command line later.

The TOCTTOU pair in *logwatch* is **<stat, mkdir>**. *logwatch* first checks whether the directory /tmp/logwatch.{pid} exists (**stat**) before creating it. However, an attacker may create that directory (as shown above) between the **stat** and **mkdir** system calls. In this case, *logwatch*'s **mkdir** fails, but since *logwatch* does not check the return value of its **mkdir**, it continues blindly and uses the temporary directory. The invariant in *logwatch* is $resolve(tmpdir) = \varnothing$ and the attack is a *creation* operation (**mkdir**) by the attacker. (Here the *tmpdir* is /tmp/logwatch.{pid})

5 The EDGI Defense against TOCTTOU

5.1 Overview

Based on Section 3.4, we propose an event driven mechanism, called EDGI (**E**vent **D**riven **G**uarding of **I**nvariants), to defend applications against TOCTTOU attacks. Due to the large

number of existing applications that suffer from TOCTTOU vulnerabilities and the existence of many different TOCTTOU pairs, it will not be a scalable approach to solve the TOCTTOU problem on a per-application and per-TOCTTOU pair basis. Therefore, we propose a system-level solution that fixes the entire class of TOCTTOU problems. Specifically, the design requirements of EDGI are:

- It should solve the problem within the file system, and does not change the API, so that legacy or future applications need not be modified.
- It should solve the problem completely, i.e., covering all TOCTTOU pairs and with no false negatives.
- It should not add undue burden on the system, i.e., very low rate of false positives.
- It should incur very low overhead on the system.

EDGI prevents TOCTTOU attacks by making a sequence of system calls on a file object safe, as suggested by Preposition 4. Under the STEM model's assumption, the "Check" part of a sequence of operations on a file object creates an invariant that should be preserved through the corresponding "Use" part. Specifically, a file, certified to be *non-existent* ($resolve(f) = \varnothing$) by the "Check" operations, should remain non-existent until the "Use" operations create it. Similarly, a file, certified to be *existent* ($resolve(f) = b$) by the "Check" operations, should remain the same file until the "Use" part (by the same user) accesses it. Identifying and preserving these two invariants ($resolve(f) = \varnothing$ or $resolve(f) = b$) are the main goals of EDGI.

In order to achieve the above goals, EDGI needs to recognize the existence of such a sequence of system calls, including the user who is making this sequence of calls as well as the start and end of the sequence. Since EDGI lacks application-level semantics information, it can do this

only by *automatic inference*. Figure 5 shows the three components of EDGI and their relationship to the STEM model. The STEM model "bootstraps" *invariant scope inference* by dictating which TOCTTOU pairs are possible in a sequence of system calls (arrow 1), while invariant scope inference decides when an invariant holder is valid (arrow 2), and *invariant maintenance* needs the invariant holder id to protect the right user (arrow 3) and only when the invariant is within scope (arrow 4). Finally, all three EDGI components are driven by system level events such as file and process operations.

In the rest of this section, we will describe the three EDGI components: invariant holder inference, invariant scope inference, and invariant maintenance, in more detail.

5.2 Automatic Inference of Invariant Holders

This part of EDGI answers the question of which user should be protected from a TOCTTOU attack. We divide the life cycle of a file object into two kinds of distinct phases: *free* and *actively used*. In the *free* phases, the file object is not operated on by any user in the system. In the *actively used* phases, the file object is operated on by at least one user (through processes started by that user). Obviously, TOCTTOU is an issue only when the file object is in the *actively used* phases. When a file object is in an *actively used* phase, EDGI needs to decide among all the active users which user should be protected. The identity of this user is maintained as a piece of meta-information called *invariant holder* associated with the file object. Once the *invariant holder* is decided, the EDGI defense is straightforward: preventing users other than the *invariant holder* from creating or removing the file object, according to Preposition 3.

Since UNIX-style file systems were designed to be shared, EDGI treats all users equally in terms of who can become the *invariant holder*, except that the *root* user can always preempt the current *invariant holder*. Specifically, when a file object is *free*, the first user who operates on it

becomes its *invariant holder* (the operation also causes the file object to enter an *actively used* phase), and while this *invariant holder* is actively using the file object, other concurrent users of the file object who come later than the *invariant holder* are considered potential attackers (this is called the *incarnation rule*). However, there is one exception to the *incarnation rule*: if the late-comer is the *root* user, it immediately becomes the new invariant holder (this is called the *root preemption rule*). The observation behind the *root preemption rule* is that the *root* user is always trusted and a process running as *root* has more value for an attacker than a normal process.

Once the current *invariant holder* finishes using the file object, the file object enters a *free* phase again in which there is no *invariant holder* associated with it.

5.3 Automatic Inference of Invariant Scopes

This part of EDGI answers the question: how long should the protection be placed? We define the length of this protection as the *scope* of the invariant. A significant technical challenge is to correctly identify this scope - the boundaries of the TOCTTOU vulnerability window of the application. Since current Unix-style file systems are oblivious to application-level semantics, EDGI needs to *infer* the scope, so no changes are imposed on the applications or the file system interfaces.

The inference of invariant scope is guided by the STEM model, which specifies the initial TOCTTOU pair explicitly. The "Use" call of the initial pair becomes the "Check" call of the next pair, completed by the following "Use" call. According to Proposition 2, the STEM model correctly captures the TOCTTOU problem. The invariant of the initial pair is maintained from the "Check" call through the "Use" call, and then to the additional "Use" calls.

EDGI infers the end of an invariant's scope using several heuristics. First, when the *invariant holder*'s process terminates, there is a good reason to believe that the *invariant holder* is done

with the file object, so the protection can be lifted. This results in the *termination rule*. Another heuristic is that when the current *invariant holder* is preempted by the *root* user, it cannot be protected anymore, so the scope of its invariant ends. Finally, there is one more heuristic about invariant scopes: the inheritance of invariants by children processes. For example, after a user checks on a file object and becomes an *invariant holder*, its process spawns a child process, and terminates. In the mean time, the child process continues, and uses the file object. If we only use the first heuristic, the invariant will be removed when the owner (parent) process terminates. In this case an attacker can achieve a TOCTTOU attack before the child process uses the file. Thus we must extend the scope of invariants to the child process at every process creation. This becomes the basis for the *clone rule*.

5.4 Invariant Maintenance Using ECA Rules

The EDGI mechanism keeps track of operations on a file object and dynamically recognizes the need for protection against TOCTTOU attacks. Invariant holder inference (Section 5.2) decides which user should be protected, and invariant scope inference (Section 5.3) decides the sequence of file system calls that needs protection. When an invariant scope and the corresponding holder are recognized, EDGI preserves such invariant by keeping users other than the invariant holder from creating / removing the file object (except for the root user). Intuitively, an EDGI invariant can be regarded as a sophisticated *lock*. While the invariant scope is active, the invariant holder is the owner of the lock, and when the invariant is out of scope, the invariant holder releases the lock. Due to the complications of Unix file system, the invariant handling is more complicated than a normal lock compatibility table. Therefore, we represent the invariant handling using ECA (Event-Condition-Action) rules [12, 13]. We note that we only use ECA rules as a model, since our implementation does not support general-purpose rule processing.

Specifically, the invariant-related information is maintained as extra state information for each file object. When an invariant-related event is triggered, the corresponding set of conditions is evaluated and if necessary, appropriate actions are taken to maintain the invariant (e.g., extending the scope).

Table 6 shows the ECA rules used in EDGI. The specifications of the rules refer to invariant-related information maintained by EDGI, which we describe in more detail below.

For each file object, EDGI maintains its state (*free* or *actively used*), invariant holder user id, and a process list. In detail:

- *refcnt* – the number of active processes using the file object. When *refcnt* = 0, the file object is free.
- *fsuid* – the user id of the processes that are actively using the file object, i.e., the invariant holder id.
- *gh_list* – a doubly-linked list, in which each node contains a process id and the timestamp of the latest system call made by the process on the file object.

In addition, invariant-related information is also associated with every symbolic link, as well as the last regular directory if the invariant is *non-existent* ($resolve(f) = \varnothing$), on the pathname of a file object. Reasons for maintaining information for such objects are discussed in Section 3.2. The information includes:

- *refcnt* – the number of active processes using the path element.
- *gh_list* – a doubly-linked list, in which each node contains a process id, a user id, and the timestamp of the last system call made by the process.

Note that such path elements do not have an *fsuid* field because they may appear on pathnames requested by different users. Instead, the user identity information is added to the *gh_list*. As long as no attempt is made to delete such an element, there will be no conflict. But once there is one, we can use information on the *gh_list* to find all relevant parties, which is the basis for resolving the access conflict. We discuss access conflicts in more detail in Section 5.5.2.

Finally, two kinds of events trigger condition evaluation:

- File system calls such as **access, open, mkdir**, etc.
- Process operations: **fork, execve, exit**.

The conditions evaluated by each event and their associated actions are summarized in Table 6 (f denotes the file object). The conditions refer to the file object status (whether the invariant is $resolve(f) = \varnothing$ or $resolve(f) = b$), and actions include the creation, removal and potentially more complex invariant maintenance actions. Note that Table 6 only covers the "end point" file object to simplify presentation, because the rules for intermediate points (e.g., symbolic links to directories) are similar.

5.5 Discussion

5.5.1 Completeness of EDGI

One important question is whether the EDGI mechanism is a complete solution, capable of stopping all TOCTTOU attacks. For every file system call, the rules summarized in Table 6 are checked and followed. The first time a "Check" call is invoked on a file object in a *free* phase, that user becomes the file object's invariant holder. At any given time there is at most one invariant holder for each file object. Users other than the invariant holder (except for root users) are not allowed to create or remove the file object (including changes to mapping between the

name and disk objects). Therefore, the EDGI defense is able to stop all TOCTTOU attacks identified by the STEM model.

5.5.2 Discussion about Access Conflict

EDGI's agnostic to application-level semantics may cause false positives. If we consider the invariants as similar to locks, then the question of dead-lock and live-lock arises. For example, it is possible that an invariant holder is a long-running process which only touches a file object at the very beginning and then never uses it again. Consequently, a legitimate user may be prevented from creating/deleting the file object for a long time - a situation that can be considered a false positive. This is particularly an issue if the invariant holder in question is malicious: by blocking a legitimate user, the malicious invariant holder is essentially mounting a denial of service attack. In order to address this issue, EDGI maintains enough information about the current invariant holder (including user id and process id) and logs such access conflicts (in terms of creating / deleting the file object). Such logged information can help an analyst identify the parties involved in the conflict and identify the malicious user if there is one. This logging facility helps deter a malicious user from holding on to a file object just to deny its creation/deletion by another user, and also helps two benign users resolve the access conflict between them.

Below we argue that EDGI correctly addresses the access conflicts, whether the file in question is shared or not. A summary of our proof is also presented in Figure 6.

If the file is private, the legitimate user (e.g., the owner) should be chosen as the invariant holder by accessing the file object first. This is typically expected because only the legitimate user knows which pathname to use to refer to the file. In this case, another user's attempt to create / delete the file object (a possible attack step) will be denied by EDGI, a correct behavior. If another user happens or manages to learn about the pathname of such a file and leverages the

EDGI mechanism to become the holder (i.e., by accessing the file before the legitimate user does), it should be considered an anomaly. Specifically,

- If the invariant is *non-existent*, this means that the legitimate user is about to create a file and the second user anticipates the pathname already – most likely the second user is malicious. In this case, it is a correct behavior for the system to warn the legitimate user that some other user has "owned" the pathname.
- If the invariant is *existent*, this means that the second user accesses the (private) file first (an anomaly). In this case, it is a correct behavior for the system to warn the legitimate user that some other user may have tampered with the pathname when the legitimate user tries to delete the file.

If the file is intended for sharing, we can have two situations: (1) neither user needs to create or delete the shared file. In this case, there will be no access conflicts in terms of creating / deleting the shared file; (2) at least one user needs to do that. While one user creates or deletes the shared file, the other user must be aware that the file may or may not exist, and may appear or disappear suddenly, all depending on the first user's activities. Purely relying on file system interfaces to implement synchronization among different users is awkward. Obviously, a good programmer should design the applications in a way that they coordinate with each other using some other application-level protocols. To better support coordination, we can add a system call for registering *friends group* on a file object which lets EDGI know that two users are friends so the lock does not apply to a friend. For example, P_1 checks the status of a file; then P_2 tries to create the same file, which should be allowed if P_1 and P_2 are in a friends group.

6 Linux Implementation of EDGI

We have implemented the mechanism described in the previous section in the Linux file system. The implementation consists of modular kernel modifications to maintain the invariants for every file object and its user/owner. We outline the process that remembers the invariant holder of each file object (Section 6.1) and then the maintenance of the invariants (Section 6.2).

6.1 Invariant Holder Tracking

Invariant holder tracking is accomplished by maintaining a hash table that keeps track of the processes that are actively using each file object. The index to this hash table is the file pathname, and for each entry, a list of process ids is maintained. Our modular implementation augments the existing directory entry (dentry) cache code and extends its data structures with the fields introduced in Section 5.4: *fsuid*, *refcnt*, and *gh_list*.

Before a system call uses a file object by name, it first needs to resolve the pathname to a dentry. Our implementation instruments the Linux kernel to call the invariant holder tracking algorithm during each such pathname resolution. There are two possible approaches to implementing this algorithm. The first is to instrument the body of every system call (e.g., **sys_open**) that uses a file pathname as argument. The second is to instrument the pathname resolution functions themselves (in the Linux case, **link_path_walk** and **lookup_hash**).

The first approach has the disadvantage that instrumented code has to spread over many places, making testing and maintenance difficult. Although techniques such as Aspect Oriented Programming (AOP) [14] could help, we were unable to find a sufficiently robust C language aspect weaver tool that can work on Linux kernel. The second approach has the advantage that only a few (in the Linux case, exactly two) places need to be instrumented, making the testing and maintenance relatively easy. We chose the second approach for our implementation.

The invariant holder tracking algorithm GH is shown in Figure 7. This algorithm effectively implements the rules summarized in Table 6, and it is called right before **link_path_walk** and **lookup_hash** successfully returns.

Line 1-2 of the invariant holder tracking algorithm addresses the situation where a new invariant holder is identified: invariant related data structure is initialized, including the invariant holder user id (*fsuid*), the invariant holder process id, and a timestamp. After these steps, the invariant maintenance part (Section 6.2) will start applying this invariant. We can see that the similar sequence also occurs in Line 6, where a new invariant holder is decided due to preemption.

Lines 3-4 address the situation in which an existing invariant holder accesses the file object again.

Lines 5-6 correspond to the preemption of invariant from a normal user to the root discussed in Section 5.2.

The invariant holder tracking algorithm needs the current process id and current user id runtime information, which are obtained from the *current* global data structure maintained by the Linux kernel.

6.2 Invariant Maintenance

The second part of implementation is invariant maintenance by thwarting the attacker's attempt to change the name to disk binding of a file object, which in turn is achieved by deleting or creating a file object. We instrumented two kernel functions to perform invariant checks:

- **may_delete**(d): this function is called to do permission check before deleting a file object d. We add invariant checking *after* all the existing checks have been passed: If d is an end point, d.refcnt > 0, and the current user id is not the same as d.fsuid, return –EBUSY (file object in

use and cannot be deleted) and log the access conflict; otherwise return 0. If d is a symbolic link to a directory or a regular directory, d.refcnt > 0, and there are users other than the current user on d.gh_list, return –EBUSY (file object in use and cannot be deleted) and log the access conflict; otherwise return 0. Note that the root user is always able to pass this check because d.fsuid becomes root irrespective of its initial value due to the root preemption rule in Table 6.

- **may_create**(d): this function is called to do permission check before creating a file object, similar invariant checking (as **may_delete** above) is added *after* all the existing checks have been passed. The difference is that the new checks are performed only on end point file objects, because symbolic link replacement attacks must first delete the symbolic link, which are stopped by the checks added to **may_delete**.

The **may_create** kernel function is called by all the system calls in the CreationSet (Section 4.2) and the **may_delete** function is called by all the system calls in the corresponding RemovalSet (Section 4.2). These invariant checks implement the Invariant Maintenance Rules 1 and 2 in Table 6.

6.3 Engineering of EDGI Software

Table 7 shows the size of EDGI implementation in Linux kernel 2.4.28. The changes were concentrated in one file (dcache.c), which was changed by about 55% (LOC means lines of code). The other changes were small, with less than 5% change in one other file (namei.c), plus single-line changes in three other files. This implementation of less than 1000 LOC was achieved after careful control and data flow analysis of the kernel, plus some tracing. We consider this implementation to be highly modular and relatively easily portable to other Linux releases.

From top-down point of view, the methodical design and implementation process benefited from the STEM model as a starting point. Then, the ECA rules facilitated the reasoning of invariant maintenance. The rules were translated into the Invariant Holder Tracking algorithm. These steps give us the confidence that the invariants are maintained by EDGI software.

Conversely, from a bottom-up point of view, the Linux kernel was organized in a methodical way. For example, it has exactly two functions (**may_delete** and **may_create**) controlling all file object status changes. By guarding these two functions, we were able to guard all 224 TOCTTOU pairs identified by the STEM model. This kind of function factoring in the Linux kernel contributed to the modular implementation of EDGI.

7 Experimental Evaluation of EDGI

7.1 Discussion of False Negatives

EDGI has to correctly identify the application that needs to be protected against TOCTTOU attacks. Our root preemption rule is very critical in this respect because by allowing the root process to become the invariant holder, EDGI prevents privilege escalation attacks through shared access to files.

The EDGI system design follows the STEM model, and the completeness of the STEM model is given in Proposition 5. If the ECA rules, summarized in Table 6, captures all the TOCTTOU pairs identified by the STEM model, and the invariant holder tracking algorithm in Figure 7 implements all the rules in Table 6, and our Linux kernel implementation (Section 6) is correct, then our implementation should have zero false negatives.

We have run all the attack experiments we could find, including known TOCTTOU vulnerabilities such as *logwatch* 2.1.1 [11] and new vulnerabilities recently detected, including

rpm, *vi/vim*, and *emacs*. In all the experiments the EDGI system is able to stop the attacker program.

7.2 Discussion of False Positives

As mentioned in Section 5.5.2, our conservative maintenance of invariants may introduce unnecessary denials of creation / deletion of file objects, if an invariant holder runs for a long time. These unnecessary denials can be considered a kind of false positives. Theoretically, one major source of false positives seems to center on symbolic links to directories that may appear in pathnames of many different user applications, especially when such directories are system-wise (e.g., containing system libraries). By analyzing the log generated by EDGI, we find that such symbolic links are just a few (less than ten) and despite the fact that they are widely shared (by more than ten processes concurrently), we have not found any false positives because an attempt to remove / create such symbolic links is rejected. For example, one such symbolic link is /usr/lib/X11 that points to /usr/X11R6/lib/X11, and this symbolic link appears in the pathnames of 17 processes. However, we have not observed any user application that wants to remove this symbolic link in a normal Linux environment.

To help reduce the false positives, EDGI logs the users and their processes in an access conflict, as a way to identify real access conflicts among legitimate users.

7.3 Overhead Measurements

We use a variant of the Andrew benchmark [15] to evaluate the overhead introduced by EDGI defense mechanism. The benchmark consists of five stages. First, it uses **mkdir** to recursively create 110 directories. Second it copies 744 files with a total size of 12MB. Third, it stats 1715 files and directories. Fourth, it *grep*s (scan through) these files and directories, reading a total

amount of 26M bytes. Fifth, it does a compilation of around 150 source files. For every stage, the total running time is calculated and recorded.

The experiments ran on a Pentium III 800MHz laptop with 640MB memory, where Red Hat Linux in single user mode was installed. We report the average and standard deviation of 20 runs for each experiment in Table 8, which compares the measurements on the original Linux kernel and on the EDGI-augmented Linux kernel. The same data is shown as bar chart in Figure 8.

The Andrew benchmark results show that EDGI generally has a moderate overhead. The only exception is **stat**, which has 47% overhead. The explanation is that **stat** takes less time than other calls (such as **mkdir**), but the extra processing due to invariant holder tracking (now part of pathname resolution) has a constant factor across different calls. This constant overhead weighs more in short system calls such as **stat**. Fortunately, **stat** is used relatively rarely, thus the overall impact remains small.

PostMark benchmark [16] is designed to create a large pool of continually changing files and to measure the transaction rates for a workload approximating a large Internet electronic mail server. PostMark first tests the speed of creating new files, and then it tests the speed of transactions. Each transaction has a pair of smaller transactions, which are either read/append or create/delete.

On the original Linux kernel the running time of this benchmark is 40.0 seconds. On EDGI-augmented kernel, with all the same parameter settings, the running time is 40.1 seconds (Again these results are averaged over 20 rounds). So the overhead is 0.25%. This result corroborates the moderate overhead of EDGI.

8 Related Work

TOCTTOU is a well known security problem [1, 2, 17]. Bishop and Dilger [18, 19] gave the first comprehensive exploration of this problem, developed a prototype analysis tool that used pattern matching to look for TOCTTOU pairs in the application source code, and suggested solutions to TOCTTOU problem including the modifications of file system interfaces. We have carried out a study on TOCTTOU vulnerabilities detection [6], experimental risk analysis [20], and prevention [21]. This paper is extended from our previous work with the addition of the Abstract File System model, the STEM model, the mapping of concrete Unix-style file systems to it, and an improved design and implementation of EDGI that addresses attacks that leverage symbolic links to directories. We note that the STEM model is extended from the CUU model in [6] with more theoretical rigor.

Static analysis of source code has shown some success in finding bugs in systems software recently. For example, Meta-compilation [22] and RacerX [23] uses compiler-extensions to find software bugs, and MOPS [24, 25] uses model checking to verify that a program preserves certain security properties. These static analysis tools could be used to detect TOCTTOU pairs in programs. However, they are limited in the detection of real TOCTTOU problems because of dynamic states (e.g., file names, ownership, and access rights) and the inherent limitations of static analysis (e.g., pointer analysis [26]).

In contrast to static analysis, dynamic detection monitors application execution to find software bugs without access to source code. These tools can be further classified into dynamic online analysis tools such as [27, 28] and post mortem analysis tools such as the one proposed by Ko et al. [29]. However, [29] can only detect the result of exploiting a TOCTTOU vulnerability and cannot locate the error.

The difficulty of detection contrasts with the simplicity of some of the technical suggestions in advisories and reports on TOCTTOU exploits from US-CERT [4] and BUGTRAQ [5], including setting proper file/directory permissions and checking the return code of function calls. However, some other suggested programming fixes are varied and non-trivial: using random numbers to obfuscate file names, replacing mktemp() with mkstemp(), and using a strict umask to protect temporary directories. More significantly, none of these fixes can be considered a comprehensive solution for TOCTTOU vulnerabilities.

Several research projects have tried to prevent subset of TOCTTOU vulnerabilities. RaceGuard [30] prevents the temporary file creation race condition in UNIX systems, specifically the **<stat, open>** TOCTTOU pair where open is used to create a file. It detects a potential race attack when a file already exists at open time and aborts the open operation. *k-race* [31] protects another specific TOCTTOU pair: **<access, open>**. The idea is to add to the original pair multiple **<access, open>** pairs (called strengthening rounds). An attack has to succeed in all the rounds. Due to the non-deterministic nature of TOCTTOU attacks, their approach makes the probability of successful attack exponentially lower with the number of rounds. Interestingly however, Borisov [32] described an effective attack against *k-race* which uses extremely long pathnames. Then Tsafrir proposed column-oriented k-race or *atomic k-race* [33] to counter Nikita's attack, and later Cai showed that *atomic k-race* can still be defeated [34]. The story is likely to continue, which demonstrates the challenging nature of TOCTTOU problems.

Several more generic defense mechanisms are TxOS [35], pseudo-transactions [8], and RPS [9]. Their basic ideas are the same: wrapping known susceptible TOCTTOU pairs inside real or pseudo-transactions, which can be used to prevent some classes of TOCTTOU vulnerabilities from being exploited. TxOS [35] adds two system calls for an application to specify code

regions that needs to run in *system transactions*, including TOCTTOU pairs. However, TxOS requires existing vulnerable applications to change their implementation. Pseudo-transactions [8] support a flexible specification of allowed and denied file system call sequences. However, they were only able to generate a set of specifications from empirical refinement through practical use, and they do not consider abuse of pseudo-transactions by malicious users. RPS [9] used a similar idea as [8] (also called pseudo transactions) to protect pre-defined TOCTTOU pairs. RPS's classification of TOCTTOU pairs is similar to ours, e.g., according to the existence of the file object. However, RPS, again, does not model user at all, which enables abuse of RPS by malicious users to attack legitimate ones. The last main difference between the STEM model and the transactions work [8, 9, 35] is the complete enumeration of exploitable TOCTTOU pairs by the STEM model. To the best of our knowledge, this complete enumeration has not been achieved before.

9 Conclusion

Due to its structural complexity (a victim process with a checking step and a use step, concurrent with an attacker process that interleaves fortuitously with the victim), TOCTTOU is a well-known and difficult problem. It is difficult to detect and reproduce because of its non-deterministic nature and typically non-obvious damages to the system. It is also difficult to prevent due to its complex interactions with the file system.

The first contribution of this paper is the STEM model of TOCTTOU vulnerabilities. The STEM model divides file system operations into four categories: *check*, *creation*, *normal use*, and *removal*. The model considers two states (*existent* and *non-existent*, defined by the mapping from a pathname to logical disk blocks) for each file object, and carefully analyzes the transitions between the two states (see Figure 1). The model is able to capture all the important state

transitions between vulnerable file system operations, called TOCTTOU pairs. By enumerating all the TOCTTOU pairs, we are able to capture all the potential TOCTTOU vulnerabilities of a file system.

The second contribution of this paper is the EDGI mechanism that prevents TOCTTOU attacks. EDGI keeps track of operations on a file object and automatically recognizes and preserves the two file invariants in the STEM model. Assuming the completeness of the STEM model, EDGI can defeat all possible TOCTTOU attacks, while not requiring any existing applications to change. Furthermore, with automatic inference of invariant holders and scopes, EDGI ensures that only a legitimate user's (e.g., the root user's) applications are protected, not those of a malicious user. Finally, a prototype of EDGI has been designed and implemented on Linux. The implementation is relatively small (less than 1000 lines of code) and carries little overhead (a few percent for application-level benchmarks).

10 References

[1] McPhee, W. S. "Operating system integrity in OS/VS2." IBM Systems Journal 13(3): 230-252, 1974.

[2] Abbott, R. P., Chin, J.S., Donnelley, J.E., Konigsford, W.L., Tokubo, S., and Webb, D.A. "Security Analysis and Enhancements of Computer Operating Systems." NBSIR 76-1041, Institute of Computer Sciences and Technology, National Bureau of Standards, April 1976.

[3] NIST. "National Vulnerability Database." NIST website. http://nvd.nist.gov/, accessed July 2010.

[4] United States Computer Emergency Readiness Team. "US-CERT Vulnerability Notes." US-CERT website. http://www.kb.cert.org/CERT_WEB\services\vul-notes.nsf/bypublished, accessed July 2010.

[5] SecurityFocus. "Bugtraq Archive." SecurityFocus website. http://www.securityfocus.com/archive/1, accessed July 2010.

[6] Wei, J., Pu, C. "TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study." Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05), San Francisco, CA, December 2005.

[7] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. "Non-Control-Data Attacks Are Realistic Threats." Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, August 2005.

[8] Tsyrklevich, E. and Yee, B. "Dynamic detection and prevention of race conditions in file accesses." Proceedings of the 12th USENIX Security Symposium, pages 243–256, Washington, DC, August 2003.

[9] Park, J., Lee, G., Lee, S., and Kim, D. "RPS: An Extension of Reference Monitor to Prevent Race-Attacks." In K. Aizawa, Y. Nakamura, and S. Satoh (Eds.): PCM 2004, LNCS 3331, pp. 556-563, 2004. Springer-Verlag Berlin Heidelberg 2004.

[10] The Open Group. "The Single UNIX Specification, Version 3, 2004 Edition." Open Group Publications website. http://www.unix.org/single_unix_specification/, accessed July 2010.

[11] IBM Internet Security Systems. "LogWatch /tmp directory race condition." IBM Internet Security Systems website. http://xforce.iss.net/xforce/xfdb/8652, accessed July 2010.

[12] McCarthy, D. R., Dayal, U. "The Architecture of an Active Data Base Management System." ACM SIGMOD Record 18 (June 1989): 215-224.

[13] Harel, D. "Statecharts: A visual formalism for complex systems." Science of Computer Programming 8 (June 1987): 231–274.

[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., and Irwin, J. "Aspect-Oriented Programming." Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[15] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. "Scale and performance in a distributed file system." ACM Transactions on Computer Systems 6 (February 1988): 51-81.

[16] FreshPorts. "PostMark 1.51_1." FreshPorts website. http://www.freshports.org/benchmarks/postmark/, accessed July 2010.

[17] Bisbey, R. and Hollingsworth, D. "Protection Analysis Project Final Report." ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, May 1978.

[18] Bishop, M. and Dilger, M. "Checking for Race Conditions in File Accesses." Computing Systems 9(Spring 1996):131–152.

[19] Bishop, M. "Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux." Technical Report 95-8, Department of Computer Science, University of California at Davis, September 1995.

[20] Wei, J., Pu, C. "Multiprocessors May Reduce System Dependability under File-Based Race Condition Attacks." In Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks 2007 (DSN '07), pp.358-367, 25-28 June 2007. doi: 10.1109/DSN.2007.67.

[21] Pu, C., Wei, J. "A Methodical Defense against TOCTTOU Attacks: The EDGI Approach." Proceedings of the International Symposium on Secure Software Engineering (ISSSE '06), Arlington, Virginia, March 13-15, 2006.

[22] Engler, D., Chelf, B., Chou, A., and Hallem, S. "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions." Proceedings of Operating Systems Design and Implementation (OSDI), San Diego, CA, October 23-25, 2000.

[23] Engler, D. and Ashcraft, K. "RacerX: Effective, Static Detection of Race Conditions and Deadlocks." Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), Lake George, NY, October 19-22, 2003.

[24] Chen, H. and Wagner, D. "MOPS: an Infrastructure for Examining Security Properties of Software." Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), Washington, DC, November 2002.

[25] Schwarz, B., Chen, H., Wagner, D., Morrison, G., West, J., Lin, J., and Tu, W. "Model Checking an Entire Linux Distribution for Security Violations." Proceedings of the 21th Annual Computer Security Applications Conference, December 6, 2005.

[26] Hind, M. "Pointer analysis: haven't we solved this problem yet?" Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Snowbird, Utah, June 18-19, 2001.

[27] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." ACM Transactions on Computer Systems 15 (November 1997): 391-411.

[28] Lhee, K. and Chapin, S. J. "Detection of File-Based Race Conditions." International Journal of Information Security 4 (February 2005): 105-119.

[29] Ko, C., Fink, G., and Levitt, K. "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring." Proceedings of the 10th Annual Computer Security Applications Conference, Orlando, FL, December 1994.

[30] Cowan, C., Beattie, S., Wright, C., and Kroah-Hartman, G. "RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities." Proceedings of the 10th USENIX Security Symposium, Washington DC, August 2001.

[31] Dean, D. and Hu, A. J. "Fixing Races for Fun and Profit: How to use access(2)." Proceedings of the 13th USENIX Security Symposium, San Diego, CA, August 2004.

[32] Borisov, N., Johnson, R., Sastry, N., and Wagner, D. "Fixing Races for Fun and Profit: How to Abuse atime." Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, July 31-August 5, 2005.

[33] Tsafrir, D., Hertz, T., Wagner, D., and Silva, D. "Portably Solving File TOCTTOU Races with Hardness Amplification." In Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008), Mary Baker, Erik Riedel (Eds.), February 26-29, 2008, San Jose, CA, USA. USENIX 2008, ISBN 978-1-931971-56-0, pp. 189-206.

[34] Cai, X., Gui, Y., and Johnson, R. "Exploiting Unix File-System Races via Algorithmic Complexity Attacks." Proceedings of the 30th IEEE Symposium on Security and Privacy, Berkeley, CA, May 17-20, 2009.

[35] Porter, D., Hofmann, O., Rossbach, C., Benn, A., and Witchel, E. “Operating System Transactions.” In Proc. of the SOSP'09, October 11-14, 2009, Big Sky, Montana, USA, pp. 161-176.

Vitae

Jinpeng Wei received a PhD in Computer Science from Georgia Institute of Technology, Atlanta, GA in 2009. He is currently an assistant professor at the School of Computing and Information Sciences, Florida International University, Miami, FL. His research interests include malware detection, information flow security in distributed systems, could computing security, and file-based race condition vulnerabilities. He is a member of the IEEE and the ACM.

Calton Pu received the PhD degree from the University of Washington in 1986. He is a professor and the John P. Imlay Jr. chair in software at Georgia Institute of Technology, Atlanta, GA. He has published more than 60 journal papers and book chapters, 170 refereed workshop and conference papers in operating systems, transaction processing, systems reliability and security, and Internet data management. He has served on more than 100 program committees for more than 50 international conferences and workshops. He is a member of the ACM, a senior member of the IEEE, and a fellow of the AAAS.

Figures

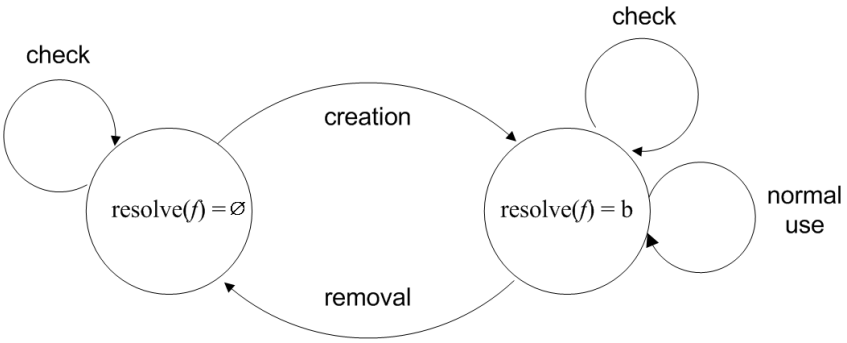

Figure 1: State Transition Diagram for FS Object *f*

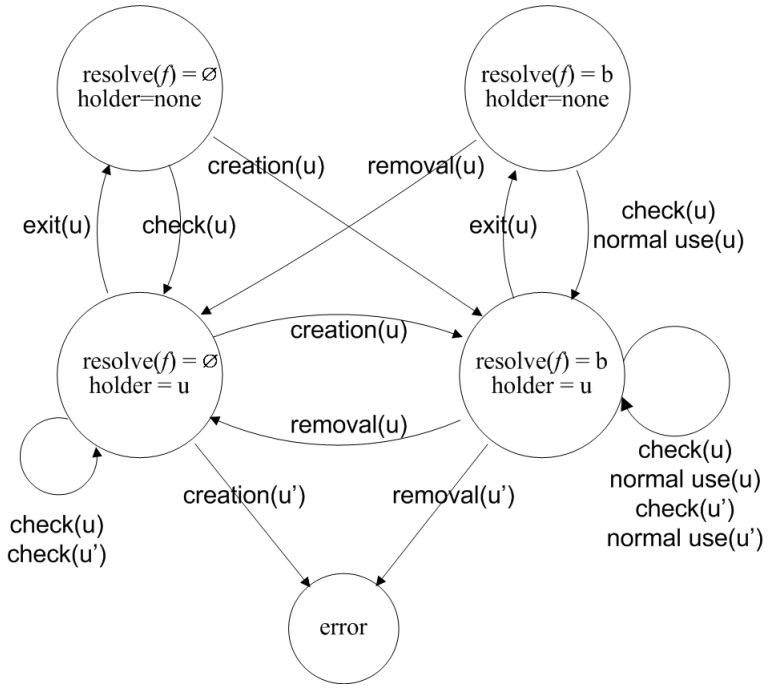

Figure 2: The Enhanced State Transition Diagram with Two Users

FileCreationSet = {**creat, open, mknod, mkfifo, rename**}
DirCreationSet = {**mkdir, rename**}
LinkCreationSet = {**link, symlink, rename**}
FileNormalUseSet = {**chmod, chown, truncate, utime, open, fopen, fdopen, popen, execl, execle, execlp, execv, execve, execvp, pathconf**}
DirNormalUseSet = {**chmod, chown, utime, chdir, pathconf**}
FileRemovalSet = {**remove, unlink, rename**}
DirRemovalSet = {**remove, rmdir, rename**}
LinkRemovalSet = {**remove, unlink, rename**}
FileCheckSet = {**access, stat**}
DirCheckSet = {**access, stat**}
LinkCheckSet = {**lstat, readlink**}

Figure 3: POSIX File Operations

FileCreationSet = {**creat, open, mknod, rename**}
DirCreationSet = {**mkdir, rename**}
LinkCreationSet = {**link, symlink, rename**}
FileNormalUseSet = {**chmod, chown, truncate, utime, open, execve**}
DirNormalUseSet = {**chmod, chown, utime, mount, chdir, chroot, pivot_root**}
FileRemovalSet = {**unlink, rename**}
DirRemovalSet = {**rmdir, rename**}
LinkRemovalSet = {**unlink, rename**}
FileCheckSet = {**stat, access**}
DirCheckSet = {**stat, access**}
LinkCheckSet = {**stat, access**}

Figure 4: Linux File Operations

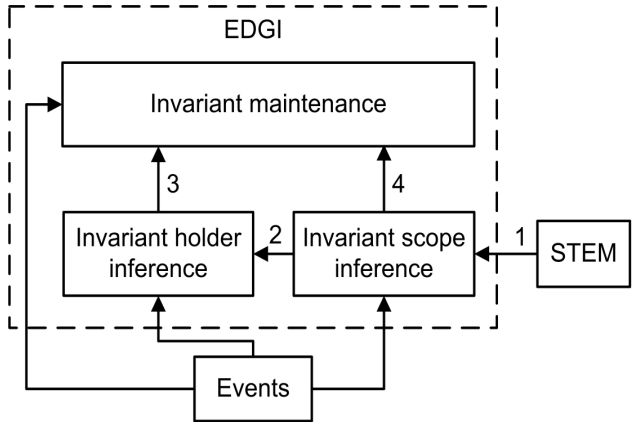

Figure 5: EDGI Modules and Their Interconnections

Let the file object be *f*, the legitimate user be *u*, and the other user be *u'*.

if *f* is private to *u* (*u'* is the attacker),

if *u* becomes the invariant holder, *u'* attempts will be denied by EDGI, correct;

if *u'* becomes the invariant holder (by guessing and accessing *f* first),

if invariant is *non-existent*, *u* gets warning when trying to create *f*, correct,

if invariant is *existent*, *u* gets warning when trying to delete *f*, correct.

if *f* is shared between *u* and *u'* **,**

if neither *u* nor *u'* needs to create / delete *f*, no conflict;

if at least one needs to create / delete *f*, requires application level cooperation between *u* and *u'*.

Figure 6: Proof that EDGI Correctly Handles Access Conflicts

```
    Input: dentry d
    Output: 0 – succeed, -1 – the binding of d is tainted.
1   if d.refcnt = 0
2   then d.fsuid ← current user id, record current pid and current time in d.gh_list, d.refcnt++, return 0.
    else
3       if d.fsuid = current user id
4       then record current pid and current time in  d.gh_list, return 0.
        else
5         if current user id = root
6         then remove all invariants on d.gh_list,  d.fsuid ← root, record current pid and current time in
          d.gh_list, d.refcnt ←1, return 0.
7         else
8           return 0.
```

Figure 7: Invariant Holder Tracking Algorithm

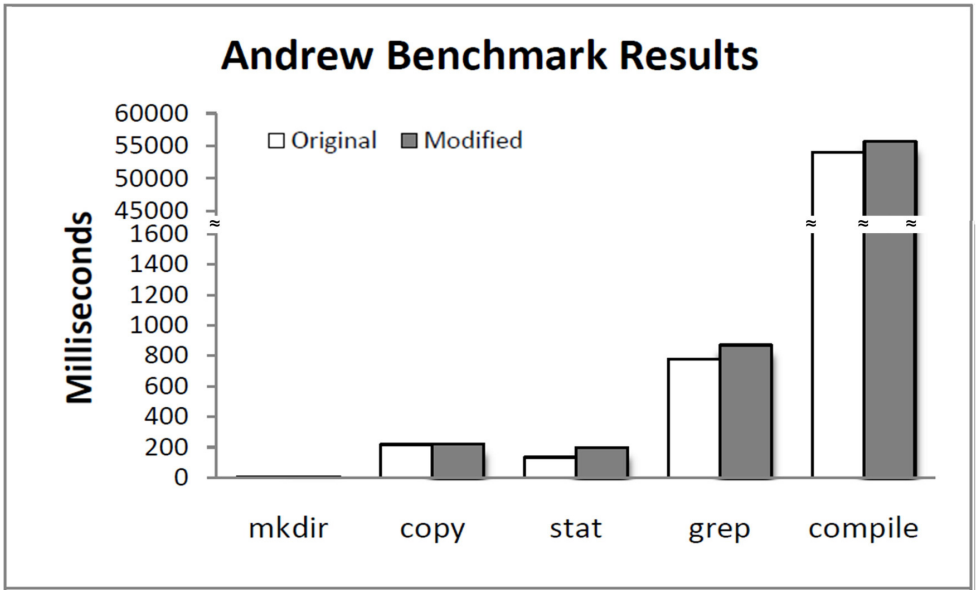

Figure 8: Andrew Benchmark Results

Tables

Table 1: Applications with TOCTTOU Vulnerabilities Reported at the Bugtraq Mailing List [5]

Domain	Application Name
Enterprise applications	Apache, bzip2, gzip, getmail, Imp-webmail, procmail, openldap, openSSL, Kerberos, OpenOffice, StarOffice, CUPS, SAP, samba
Administrative tools	at, diskcheck, GNU fileutils, logwatch, patchadd
Device managers	Esound, glint, pppd, Xinetd
Development tools	make, perl, Rational ClearCase, KDE, BitKeeper, Cscope

Table 2: Exploitable TOCTTOU Pairs (AbsFS)

Invariant	**TOCTTOU Pairs**
$resolve(f) = \varnothing$	<check, creation> <removal, creation>
$resolve(f) = b$	<creation, normal use> <check, normal use> <normal use, normal use> <creation, removal> <check, removal> <normal use, removal>

Table 3 Enumeration of Exploitable TOCTTOU Pairs (Unix-Style File Systems)

Invariant	Exploitable TOCTTOU Pairs
$resolve(f) = \varnothing$	(FileCheckSet × FileCreationSet) ∪ (FileRemovalSet × FileCreationSet) ∪ (DirCheckSet × DirCreationSet) ∪ (DirRemovalSet × DirCreationSet) ∪ (LinkCheckSet × LinkCreationSet) ∪ (LinkRemovalSet × LinkCreationSet)
$resolve(f) = b$	(FileCheckSet × FileNormalUseSet) ∪ (FileCreationSet × FileNormalUseSet) ∪ (LinkCreationSet × FileNormalUseSet) ∪ (FileNormalUseSet × FileNormalUseSet)∪ (DirCheckSet × DirNormalUseSet) ∪ (DirCreationSet × DirNormalUseSet) ∪ (LinkCreationSet × DirNormalUseSet) ∪ (DirNormalUseSet × DirNormalUseSet)

Table 4: Exploitable TOCTTOU Pairs in Linux

Invariant	TOCTTOU Pairs
$resolve(f) = \varnothing$	<stat, creat> <stat, open> <stat, mknod> <stat, rename> <access, creat> <access, open> <access, mknod> <access, rename> <unlink, creat> <unlink, open> <unlink, mknod> <unlink, rename> <rename, creat> <rename, open> <rename, mknod> <rename, rename> <stat, mkdir> <access, mkdir> <rmdir, mkdir> <rmdir, rename> <rename, mkdir> <stat, link> <stat, symlink> <access, link> <access, symlink> <unlink, link> <unlink, symlink> <rename, link> <rename, symlink>
$resolve(f) = b$	<stat, chmod> <stat, chown> <stat, truncate> <stat, utime> <stat, open> <stat, execve> <access, chmod> <access, chown> <access, truncate> <access, utime> <access, open> <access, execve> <creat, chmod> <creat, chown> <creat, truncate> <creat, utime> <creat, open> <creat, execve> <open, chmod> <open, chown> <open, truncate> <open, utime> <open, open> <open, execve> <mknod, chmod> <mknod, chown> <mknod, truncate> <mknod, utime> <mknod, open> <mknod, execve> <rename, chmod> <rename, chown> <rename, truncate> <rename, utime> <rename, open> <rename, execve> <link, chmod> <link, chown> <link, truncate> <link, utime> <link, open> <link, execve> <symlink, chmod> <symlink, chown> <symlink, truncate> <symlink, utime> <symlink, open> <symlink, execve> <chmod, chmod> <chmod, chown> <chmod, truncate> <chmod, utime> <chmod, open> <chmod, execve> <chown, chmod>

	<chown, chown> <chown, truncate> <chown, utime> <chown, open> <chown, execve> <truncate, chmod> <truncate, chown> <truncate, truncate> <truncate, utime> <truncate, open> <truncate, execve> <utime, chmod> <utime, chown> <utime, truncate> <utime, utime> <utime, open> <utime, execve> <open, chmod> <open, chown> <open, truncate> <open, utime> <open, open> <open, execve> <execve, chmod> <execve, chown> <execve, truncate> <execve, utime> <execve, open> <execve, execve> <stat, mount> <stat, chdir> <stat, chroot> <stat, pivot_root> <access, mount> <access, chdir> <access, chroot> <access, pivot_root> <mkdir, chmod> <mkdir, chown> <mkdir, utime> <mkdir, mount> <mkdir, chdir> <mkdir, chroot> <mkdir, pivot_root> <rename, mount> <rename, chdir> <rename, chroot> <rename, pivot_root> <link, chmod> <link, chown> <link, utime> <link, mount> <link, chdir> <link, chroot> <link, pivot_root> <symlink, chmod> <symlink, chown> <symlink, utime> <symlink, mount> <symlink, chdir> <symlink, chroot> <symlink, pivot_root> <chmod, mount> <chmod, chdir> <chmod, chroot> <chmod, pivot_root> <chown, mount> <chown, chdir> <chown, chroot> <chown, pivot_root> <utime, mount> <utime, chdir> <utime, chroot> <utime, pivot_root> <mount, chmod> <mount, chown> <mount, utime> <mount, mount> <mount, chdir> <mount, chroot> <mount, pivot_root> <chdir, chmod> <chdir, chown> <chdir, utime> <chdir, mount> <chdir, chdir> <chdir, chroot> <chdir, pivot_root> <chroot, chmod> <chroot, chown> <chroot, utime> <chroot, mount> <chroot, chdir> <chroot, chroot> <chroot, pivot_root> <pivot_root, chmod> <pivot_root, chown> <pivot_root, utime> <pivot_root, mount> <pivot_root, chdir> <pivot_root, chroot> <pivot_root, pivot_root>

Table 5: Some Existing TOCTTOU Vulnerabilities on Unix-style Systems

Applications	TOCTTOU pair	Classification and Invariant
BitKeeper, Cscope 15.5, CUPS, get-mail 4.2.0, glint, Kerberos 4, openl-dap, OpenOffice 1.0.1, patchadd, procmail, samba, Xinetd	**<stat, open>**	FileCheckSet × FileCreationSet $resolve(f) = \varnothing$
Rational ClearCase, pppd	**<stat, chmod>**	FileCheckSet × FileNormalUseSet $resolve(f) = b$
Sendmail	**<stat, open>**	
logwatch 2.1.1	**<stat, mkdir>**	DirCheckSet × DirCreationSet $resolve(f) = \varnothing$
bzip2-1.0.1, gzip, SAP	**<open, chmod>**	FileCreationSet × FileNormalUseSet $resolve(f) = b$
Mac OS X 10.4 – launchd	**<open, chown>**	
StarOffice 5.2	**<mkdir, chmod>**	DirCreationSet × DirNormalUseSet $resolve(f) = b$

Table 6: Invariant Maintenance Rules in EDGI

Name	Event	Condition	Action
Incarnation rule	Any system call on f	refcnt == 0	Set f's state as actively used (refcnt++); fsuid as current user id, record current pid and current system time in the gh_list.
Reinforcement rule	Any system call on f	refcnt > 0 and fsuid == current user id	Record current pid and current system time in the gh_list.
Root preemption rule	Any system call on f	refcnt > 0 and fsuid != current user id and current user id == root	Remove all invariant holder information from the gh_list; set f's fsuid as current user id, set refcnt as 1, record current pid and current system time in the gh_list.
Invariant maintenance rule 1	Any system call in the RemovalSet (4.2) on f	refcnt > 0 and fsuid != current user id	Deny the current request.
Invariant maintenance rule 2	Any system call in the CreationSet (4.2) on f	refcnt > 0 and fsuid != current user id	Deny the current request.
Clone rule	fork (parent, child)	True	For each file object that has parent in its gh_list, record child and current system time, and increment the refcnt.
Termination rule	exit	True	Remove current pid from the gh_list of each file object that has it on its gh_list, and decrement the corresponding refcnt.
Distract rule	execve	True	Remove current pid from the gh_list of each file object that has it on its gh_list, and decrement the corresponding refcnt.

Table 7: Linux Implementation of EDGI

Source File	Modified Places	Original LOC	Added LOC
fs/dcache.c	5	1307	793
fs/namei.c	5	2047	118
fs/exec.c	1	1157	1
kernel/exit.c	1	602	1
kernel/fork.c	1	896	1

Table 8: Andrew Benchmark Results (in milliseconds)

Functions	Original Linux	Modified Linux	Overhead
mkdir	6.35 ±0.21	6.43 ±0.19	1.3%
copy	217.0 ±1.5	218.6 ±1.4	0.7%
stat	132.0 ±1.9	193.6 ±0.8	47%
grep	777.0 ±4.3	870.1 ±5.3	12%
compile	53,971 ±434	55,615 ±367	3.0%