Portably Solving File TOCTTOU Races with Hardness Amplification

Dan Tsafrir
IBM Research
Yorktown Heights, NY
dants@us.ibm.com

Tomer Hertz
Microsoft Research
Redmond, WA
hertz@microsoft.com

David Wagner
UC Berkeley
Berkeley, CA
daw@cs.berkeley.edu

Dilma Da Silva
IBM Research
Yorktown Heights, NY
dilmasilva@us.ibm.com

Abstract

The file-system API of contemporary systems makes programs vulnerable to TOCTTOU (time of check to time of use) race conditions. Existing solutions either help users to detect these problems (by pinpointing their locations in the code), or prevent the problem altogether (by modifying the kernel or its API). The latter alternative is not prevalent, and the former is just the first step: programmers must still address TOCTTOU flaws within the limits of the existing API with which several important tasks can not be accomplished in a portable straightforward manner. Recently, Dean and Hu addressed this problem and suggested a probabilistic hardness amplification approach that alleviated the matter. Alas, shortly after, Borisov et al. responded with an attack termed "filesystem maze" that defeated the new approach.

We begin by noting that mazes constitute a generic way to deterministically win many TOCTTOU races (gone are the days when the probability was small). In the face of this threat, we (1) develop a new user-level defense that can withstand mazes, and (2) show that our method is undefeated even by much stronger hypothetical attacks that provide the adversary program with ideal conditions to win the race (enjoying complete and instantaneous knowledge about the defending program's actions and being able to perfectly synchronize accordingly). The fact that our approach is immune to these unrealistic attacks suggests it can be used as a simple and portable solution to a large class of TOCTTOU vulnerabilities, without requiring modifications to the underlying operating system.

1 Introduction

The TOCTTOU (time of check to time of use) race condition was characterized in 1974 by McPhee as the situation which occurs

> *"if there exists a time interval between a validity-check and the operation connected with that validity-check [such that], through multitasking, the validity-check variables can deliberately be changed during this time interval, resulting in an invalid operation being performed by the control program."* [25]

Dissecting a 1993 CERT advisory [7], Bishop was the first to systematically show that file-systems with weak consistency semantics (like Unix and Windows) are inherently vulnerable to TOCTTOU races [3, 4]: First, a program checks the status of a file using the file's name. Then, depending on the status, it applies some operation to the file, unjustifiably assuming the status has not changed since it was checked. This error is caused by the fact that the mapping between file names and file objects ("inodes") is mutable by design, and might therefore change between a status check and a subsequent operation.

Researchers have put a lot of effort into trying to solve or alleviate the problem, (1) developing compile-time tools to pinpoint locations in the source code that are suspect of suffering from a TOCTTOU race [4, 37, 10, 8, 30], (2) modifying the kernel to log all relevant system calls and analyzing the log, postmortem, to detect TOCTTOU attacks [20, 16, 21, 19, 39, 1], (3) having the kernel speculatively identify offending processes and temporarily suspend them or fail their respective suspected system calls [11, 34, 27, 35, 28], and finally (4) designing new file-system interfaces to make it easier for programmers to avoid the races. [3, 29, 24, 40].

None of the above helps programmers to safely and portably accomplish a TOCTTOU-prone task on *existing* systems, as kernels that prevent races are currently an academic exercise, whereas new-and-improved file-systems are unfortunately not prevalent (and certainly not standard). Thus, regardless of how programmers become aware of the problem, whether through compile-time tools or just by being careful, they must still face the problem with the existing API.

At the same time, resolving a TOCTTOU race is not as easy as, e.g., fixing a buffer overflow bug, because the

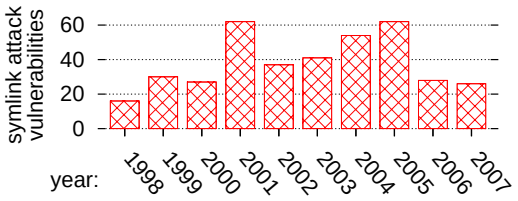


Figure 1: *NVD reports 450 "symlink attack" vulnerabilities, as of September 5, 2007. (In 2001 and 2005 there were 73 and 106 reports, respectively; the associated bars are truncated.)*

programmer must somehow achieve atomicity of two operations using an API that was not designed for such a purpose. In fact, overcoming TOCTTOU races in a portable manner is notoriously hard, sometimes even for experts (see Section 2.3). Hence, it is probably impractical to expect average programmers to successfully accomplish such tasks (or attempt them) on a regular basis.

Indeed, to date, TOCTTOU races pose a significant problem, as exemplified by Wei and Pu, which analyzed CERT [36] advisories between 2000 and 2004 and found 20 reports concerning the issue, 11 of which provided the attacker with unauthorized root access [39]. Figure 1 shows the yearly number of TOCTTOU "symlink attack" vulnerabilities reported by NVD (National Vulnerability Database) [26]. These affect a wide range of mainstream applications and tools (e.g., bzip2, gzip, FireFox, make, OpenOffice, OpenSSL, Kerberos, perl, samba, sh), environments (e.g., GNOME, KDE), distributions (e.g., Debian, Mandrake, RedHat, SuSE, Ubuntu), and operating systems (e.g., AIX, FreeBSD, HPUX, Linux, Solaris).

We contend that the situation can potentially be greatly improved if programmers are able to use some portable, standard, generic, user-mode check_use utility function that, given a 'check' operation and a 'use' operation, would perform the two as a kind of "transaction", in a way that appears atomic for all relevant purposes. This paper takes a significant step towards achieving such a goal.

The first step in this direction was taken in 2004 by Dean and Hu, which implemented a transaction-like access_open routine that set out to solve a single race [12]: the one which occurs between the access system call (used by root to check if a user has adequate privileges to open a file) and the subsequent open. Their idea (later termed *K-race* [5]) was to use *hardness amplification* as found in the cryptology literature [41], but applied to system calls rather than cryptologic primitives. In a nutshell, if an adversary has a probability $p < 1$ to win a race, then the probability p^K to win K races can be made negligible by choosing a big enough K. Indeed, by mandating attackers to win K consecutive races before agreeing to open the file, access_open seemingly accomplished its "transactional" goal of aggregating access and open into a single "atomic" operation.

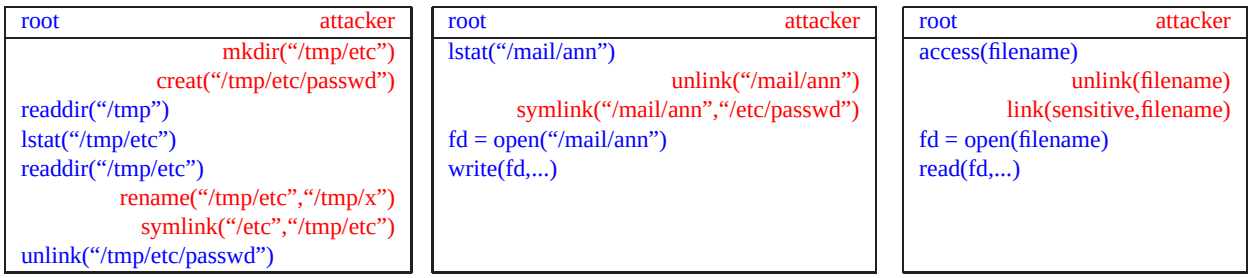
But the new and intriguing K-race defense did not stand the test of time. In 2005, Borisov et al. orchestrated their *filesystem maze* attack and showed that an adversary can in fact win *every* race (hence making the assumption that $p < 1$ wrong) [5]. Roughly speaking, the adversary is able to slow down, and effectively "single step", the proposed algorithm by feeding it with a carefully constructed file name (the "maze") and polling the status of certain components within the name. This induces perfect synchronicity between the adversary and the K-race, thereby enabling the adversary to win all races ($p \approx 1$). Indeed, in his on-line publication list, adjacent to his 2004 paper [12], Alan Hu concedes that

> *"The scheme proposed here has been beautifully and thoroughly demolished by Borisov, Johnson, Sastry, and Wagner [5]. The theory is, of course, still valid, but it relies on an assumption of the attacker having a non-negligible probability of losing races. Borisov et al. came up with ingenious means (1) to force the victim to go to disk on each race, thereby allowing plenty of time for the attacker to win races, and (2) to determine precisely what protocol operation the victim is doing at any point in time, thereby foiling the randomized delays. The upshot is that they can win these TOCTTOU races with almost complete certainty."* [17]

Dean and Hu were only concerned with finding a way to correctly use the access system call; likewise, the explicit goal of Borisov et al. was to prove that access should never be used. But the consequences of the filesystem maze attack are much more general. In fact, mazes constitute a generic way to consistently win a large class of TOCTTOU races. This is true because any 'check' operation can be slowed down and single-stepped, if provided with a filesystem maze as an argument. Consequently, the common belief that "TOCTTOU vulnerabilities are hard to exploit, because they [...] rely on whether the attacking code is executed within the usually narrow window of vulnerability (on the order of milliseconds)" [39] is no longer true: With filesystem mazes, the attacker can often proactively prolong the vulnerability window, while simultaneously finding out when it opens up.

Motivated by the alarmingly wide applicability of the filesystem maze attack, we set out to search for an effective defense, with the long-term goal of providing programmers with a generic and portable check_use utility function that would allow for a pseudo-atomic transaction of the 'check' and 'use' operations. Importantly this should work on existing systems, without requiring changes to the kernel or the API it provides.

This paper is structured as follows: After exemplifying the TOCTTOU problem in detail, surveying the existing



(a) garbage collector *(b) mail server* *(c) setuid*

Figure 2: *Three canonical file TOCTTOU examples. The Y-axis denotes the time (future is downwards). The left-justified operations, performed by root, suffer from a TOCTTOU vulnerability. The right-justified operations show how an attacker can exploit this vulnerability to circumvent the system's protection mechanisms and to gain illegal access.*

solutions, and pointing out their shortcomings and the elusiveness of a contemporary practical solution (Section 2), we go on to explain how hardness amplification was applied to solve file TOCTTOU races, and why it has failed (Section 3). We then show how to turn this failure to success (Section 4) and experimentally evaluate our solution by subjecting it to a hypothetical attack far more powerful than filesystem mazes (Sections 5–6). We discuss how to generalize our solution, its limitations, and how/when its probabilistic aspect can be eliminated (Section 7). Finally, we present our conclusions (Section 8).

2 Motivation

Much of the administrative and security-crucial tasks of Unix-like systems is performed by root-privileged programs. Since such programs often interact with and affect the system by means of file manipulation, they are susceptible to TOCTTOU vulnerabilities. A successful exploitation of these vulnerabilities would allow a non-privileged user to circumvent the system's normal protection mechanisms and unlawfully execute some operation as root.

2.1 Classic Examples

For example, many sites periodically delete files residing under the `/tmp` directory. If a file was not accessed for a certain amount of time, the "garbage collection" script deletes it. Maziéres and Kaashoek noted that this policy might contain a TOCTTOU window between the 'check' statement (of the file access time) and the subsequent 'use' statement (the file removal); if a name/inode mapping changes within this window, the script can be tricked into deleting any arbitrary file, even if it attempts to prevent this from happening by explicitly ignoring symbolic links [24]. This is illustrated in Figure 2a: The garbage collector uses lstat to verify that `/tmp/etc` is not a symbolic link. But as with all TOCTTOU flaws, this check is fruitless in case `/tmp/etc` is manipulated just after.

Another well known TOCTTOU example, initially documented by Bishop, is that of a mail server which appends a new message to the corresponding user's Inbox file [3, 4]. Before open-ing the Inbox, the server lstat-s it to rule out the possibility the user has replaced it with some symbolic link pointing to a file that lies elsewhere. Figure 2b shows how the inevitable associated TOCTTOU race can be exploited to add arbitrary data to the `/etc/passwd` file, providing the attacker with the ability to obtain permanent root access.

A third example concerns the *setuid bit* that Unix-like systems associate with an executable to indicate it should run with the privileges of its *owner*, rather than the user that *invoked* it (as is the normal case). Of course just handing off root privileges is not a good idea, which is why the access system call conveys setuid programs the ability to check whether an invoker has adequate privileges:

```
if( access(filename,R_OK) == 0 )
   fd = open(filename,O_RDONLY);
```

Alas, the access/open idiom constitutes the archetypal, and arguably the most infamous, TOCTTOU flaw.[1] Figure 2c illustrates how this race can be exploited to access any file; access was therefore deemed unusable, as e.g. indicated by its FreeBSD manual, explicitly stating that "the access system call is a potential security hole due to race conditions and *should never be used*." [22]

2.2 Existing Solutions

Considerable research effort have been put into providing solutions for TOCTTOU vulnerabilities like the ones described above. In order to highlight the contribution of

[1] This race was reported by what is believed to be the first formal documentation of a file TOCTTOU vulnerability [7]; it is described by almost all papers that address the TOCTTOU issue (see Section 2.2) when exemplifying the problem.

this paper we first survey this work, which can be subdivided into four categories:

Static Detection Some groundbreaking work has been done in recent years to statically analyze the source code of programs and pinpoint the locations of nontrivial vulnerabilities and bugs [14, 15, 2, 13]. This type of analysis is rooted in Bishop's work, which used pattern matching to locate pairs of TOCTTOU system calls in root-privileged programs on a per-function basis [3, 4]. The tools ITS4 [37], Eau Claire [10], and MOPS [8, 30] have later superseded Bishop's work by being more general, accurate, and scalable.

Dynamic Detection Static analysis can be very effective and has the advantage of (1) not incurring runtime overheads, (2) covering all the code (in a reasonable amount of time), and (3) locating the bugs before the system is deployed. But the code is not always available, and even if it is, the static doctrine is inherently missing key information that is often only available at runtime, which might result in many false positives. To solve this, Ko and Redmond patched the kernel to log the required information and utilized it, postmortem, to feed a model that detects TOCTTOU flaws [20]. A similar approach was later adopted by many following projects [16, 21, 19, 39, 1]. Notable of these is the work by Wei and Pu [39] that exhaustively enumerated all of Linux's TOCTTOU pairs[2] and the revolutionary IntroVirt tool by Joshi et al. [19] that made ubiquitous virtual-machine checkpointing and replaying a realistic alternative that can e.g. be used to identify TOCTTOU attacks, postmortem.

Dynamic Prevention The kernel can be modified to apply the principles of dynamic detection on-the-fly, as discovering TOCTTOU attacks while they occur allows for on-line prevention. This approach was first taken by Cowan et al. in 2001, when implementing "RaceGuard" [11]. Their technique tackles one TOCTTOU flaw that occurs between (1) a check if a candidate name for a temporary file doesn't match an exist file, and (2) the new file's creation (stat/open). They modify the kernel to maintain a cache of files that have been stated and found not to exist; if a subsequent open finds an existing file, it fails.

In 2003, Tsyrklevich and Yee developed a more general approach that was capable of generically preventing most TOCTTOU attacks [34]. They patched the kernel to suspend any process that interferes with a "pseudo transaction" (check/use pair that agree on the target file), such that the worst outcome of a false-positive detection is a temporary suspension of the corresponding process. Several similar solutions followed [27, 35, 28]; the latter of which, by Pu and Wei, was argued to be "complete", being based on their aforementioned earlier work [39].

New API All of the above are solutions that respect the existing file-system API so as to accommodate existing applications and operating systems. The complementary approach is to augment or change the API, such that tasks that currently suffer from TOCTTOU issues are made easier to safely accomplish. For example, to resolve the access/open race, Dean and Hu suggested that open would accept an O_RUID flag, which would instruct it to use the real (rather than effective) user ID of the process [12]; alternatively, Bishop suggested to add a new faccess system call that would operate on a file-descriptor rather than a file name [3].[3] Likewise, the O_NOFOLLOW flag supported by Linux and FreeBSD makes open fail if its argument refers to a symbolic link, which may help in certain cases (e.g. Figure 2b). However, aside from being non-portable, it relates only to the last component of the file path: earlier components may still be symbolic links, and hence be juggled by an attacker (e.g. Figure 2a).

To obtain a more general solution, a bigger change is needed, such as replacing (or augmenting) Unix semantics with that of a transactional file-system [29, 40]: Atomicity would then insure that a check/use pair that was annotated by the programmer as a single transaction would be executed with no interference.

A more radical approach was suggested by Maziéres and Kaashoek [24]. They proposed to use the fact that the binding between file descriptors and inodes is immutable (and thus cannot be exploited) to devise a safer programming paradigm that would make it harder for the programmer to make mistakes. By this paradigm,

1. all access checks would be done on file descriptors rather than on names,
2. users would be given explicit control of whether symlinks are followed when files are opened, and
3. each system call invocation would be provided with the user credentials with which the system call should operate.

We contend that some of this vision can be realized in user-mode on current systems.

[2] Wei and Pu (and later Lhee and Chapin [21]) augmented the definition of check/use TOCTTOU pairs to also refer to use/use pairs. With this, they found a bug in rpm that (1) generated a script that was writable by all (first use of open), and (2) executed it with root privileges (second use of open). While such bugs can be very hard to detect, they are nevertheless very easy to fix and therefore are of no interest in this paper.

[3] We note in passing that even though this suggestion was raised again by Dean and Hu , we contend it is impossible: the corresponding inode can possibly be refereed to by multiple paths, among which some are accessible to the user and some are not.

2.3 The Problem

Notice that all the existing solutions surveyed above *do not help programmers in resolving a known TOCTTOU flaw within existing systems*. Static detection techniques are invaluable in locating such flaws, but what are programmers to do if/once they are aware of the vulnerability? Surely they cannot wait until all contemporary kernels employ dynamic prevention (if ever, as significant complexity and performance penalty might be involved). Likewise, programmers cannot wait until all contemporary OSs portably support transactional file-systems (or constructs like the aforementioned API suggested by Maziéres and Kaashoek).

The fact of the matter is that, in order to achieve a portable solution, programmers are bound to handling the matter with a decades-old API. Importantly, as mentioned earlier, a portable user-mode solution to a given TOCTTOU race (if exists) is often much harder and more elusive than e.g. fixing a buffer overflow bug: even experts that explicitly target a specific TOCTTOU problem are prone to getting it wrong.

Consider for example the access/open race depicted in Figure 2c. Tsyrklevich and Yee suggested two solutions to this flaw [34]. The first argues that "to avoid this race condition, an application should change its effective id [with set∗uid system calls] to that of a desired user and then make the open system call directly." However, after carefully evaluating this suggestion, Dean and Hu found that

> *"Unfortunately, the setuid family of system calls is its own rats nest. On different Unix and Unix-like systems, system calls of the same name and arguments can have different semantics, including the possibility of silent failure [9]. Hence, a solution depending on user id juggling can be made to work, but is generally not portable."* [12]

The second suggestion by Tsyrklevich and Yee was "to use fstat after the open instead of invoking access". As the input of fstat is a file descriptor, the latter is permanently mapped to the underlying inode and hence can never be abused by an attacker; the user is then expected to inspect the ownership information returned by fstat and check if the invoker was indeed allowed to open the file. But this will not work, as file access permissions can *not* be deduced in such a way; rather, they are the conjunction of all the (inode) permissions associated with each component in the respective path. For example, if a file's name is x/y such that x is solely accessible by its owner, then other users are forbidden from reading y even if fstat indicates it is readable by all (which may very well be the case when root invokes the fstat).

A third alternative is to fork a child that permanently drops all extra privileges and then attempts to open the file; if successful, the child can then pass the open file descriptor across a Unix-domain socket and exit. Borisov et al. [5] have mistakingly attributed the claim that this version is portable, to Dean and Hu [12]. But the latter have actually argued the contrary, stating that, with respect to the Unix-domain approach, "some of the above [user id juggling] caveats still apply". Indeed, as mentioned earlier, dropping privileges is a non-portable operation [9]. (Regardless of whether it is being done by a parent or a forked child.) Furthermore, we find that passing an open descriptor alone, even without dropping privileges, suffers from serious portability issues.[4]

A fourth failed attempt will be discussed next.

3 Failure of Hardness Amplification

In 2004, noting that no prior art helps programmers to portably resolve TOCTTOU vulnerabilities on existing systems, Dean and Hu took the first step towards a portable solution [12], explicitly focusing their efforts on the aforementioned access/open TOCTTOU race.

3.1 The *K*-Race Technique

Their solution, termed "K-race", was inspired by the hardness amplification technique that is commonly used in cryptology contexts [41]. The idea underlying hardness amplification is to use a problem which is computationally "somewhat hard", in order do devise another computational problem that is "really hard". In a TOCTTOU access/open scenario, the "somewhat hard" problem is timing and completing the attack (removing one file and linking another) within the exact window of opportunity delimited by the access and open calls (see Figure 2c). The "really hard" problem is requiring the attacker to succeed in doing this for $2K + 1$ consecutive times.

The K-race routine, shown in Figure 3, starts with a standard call to access, followed by an open, followed by K *strengthening rounds*. Each round consists of an additional access check and a corresponding open, which are then followed by a statement that verifies that the currently opened file is the same file that was opened in the previous round. Note that when $K = 0$, the routine degenerates to the standard access/open TOCTTOU race.

[4] This is the result of changes related to the msghdr structure, which is used by the sendmsg and recvmsg system calls to pass an open descriptor through a Unix domain socket. Specifically, (1) in the mid 1990s, POSIX replaced the msg_accrights field with the msg_control array (but commercial OSes such as Solaris and HPUX preferred to keep the earlier version as the default) and (2) more recently, RFC 3542 defined a set of macros to be exclusively used when accessing / manipulating the msg_control array (but despite being mandated by OSes like Linux, some of the macros are not yet standard) [33]. The end result is lack of portability and source code that is littered with ifdefs and conditional compilation tricks [32, 42, 31, 6].

```
#define DO_SYS(call) if((call)==-1) return -1
#define DO_CHK(expr) if( !(expr)  ) return -1
#define DO_CMP(x,y)                          \
     ( ((x)->st_ino == (y)->st_ino) &&  \
       ((x)->st_dev == (y)->st_dev) )


int access_open_2004(char *fname)
{
   int fd1, fd2, i;
   struct stat s1, s2;

   // 1- the access/open idiom
   DO_SYS(       access(fname, R_OK    ) );
   DO_SYS( fd1 = open  (fname, O_RDONLY) );
   DO_SYS(       fstat (fd1  , &s1     ) );

   // 2- the strengthening rounds
   for(i=0; i<K; i++) {
      DO_SYS(       access(fname, R_OK    ) );
      DO_SYS( fd2 = open  (fname, O_RDONLY) );
      DO_SYS(       fstat (fd2  , &s2     ) );
      DO_SYS(       close (fd2            ) );
      DO_CHK(       DO_CMP(&s1  , &s2     ) );
   }

   return fd1;
}
```

Figure 3: *The K-race routine employs hardness amplification to probabilistically solve a TOCTTOU race. Specifically, on each strengthening round, it checks that the caller still has appropriate access permissions and that the underlying file-object, as represented by the inode (st_ino) and IO device (st_dev), remains the same. This attempts to provide programmers with a way to invoke access and open in an "atomic" manner.*

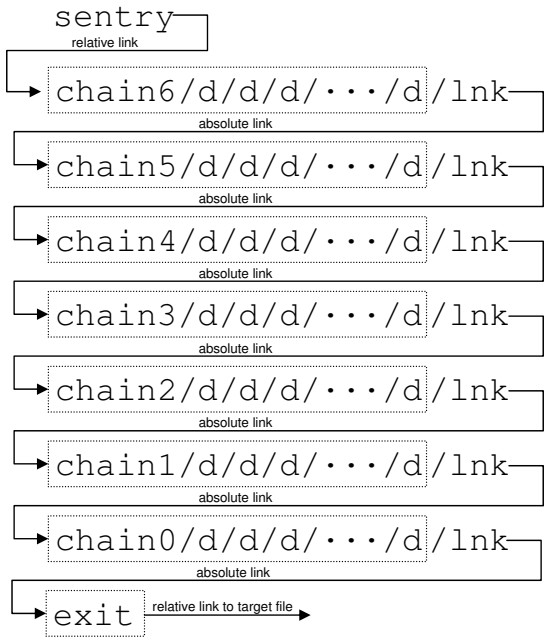


Figure 4: *The structure of a six-chains filesystem maze. Arrows represents symbolic links. (Originally published in [5]; reprinted with permission.)*

To be successful, an attacker must indeed win $2K + 1$ races: This is true because, on each round, the access check must be applied to some user accessible file, or else permission is denied; On the other hand, every open must be applied to the same inaccessible target file, or else the verification that all file-descriptors refer to the same file-object would fail. Thus, assuming each race is an independent random event with some probability $p < 1$ for the attacker to win, the overall probability of tricking a K-race is p^{2K+1}. (Independence of events is supposedly obtained by introducing short random delays between successive system call invocations: as delays are randomized, an adversary wouldn't be able to synchronize with the K-race.) After measuring several systems (among which are SMP systems), Dean and Hu concluded that K=7 is enough to make the probability of success negligible for all practical purposes.

3.2 Filesystem Mazes

In 2005, Borisov et al. defeated the K-race technique [5]. They have done so by refuting the (then widely accepted) assumption that the probability p for an attacker to win a race is significantly smaller than one. In fact, they have managed to effectively make it a certainty ($p \approx 1$). The heart of the attack consists of a *filesystem maze*, which, in simple terms, is the longest and most nested filepath a user can pass as an argument to a system call, without causing it to fail due to hardcoded kernel limits.

Constructing a Maze The basic building block of a maze is a *chain*, defined to be (nearly) the deepest nested directory tree one can define without violating the PATH_MAX constraint imposed by the kernel on the length of file paths (4KB is a typical value). Thus, $chain_0$ would be `chain0/d/d/d/.../d` such that the associated number of characters is a bit less than PATH_MAX. Likewise, $chain_1$ is `chain1/d/d/d/.../d`, etc.

To form a maze, the attacker connects chains by placing a symbolic link at the bottom of $chain_{i+1}$ that points to $chain_i$. The final symlink, at the bottom of $chain_0$, points to an `exit` symlink which, in turn, points to the actual target file. Finally, the entry point to the maze, `sentry`, is a symlink pointing to the highest chain. This is illustrated in Figure 4.

Unix systems impose a limit on the total number of symlinks that a single filename lookup can traverse, e.g., Linux 2.6 limits this number to 40. This places a limit on the number of chains composing the maze. Still, even with this limit, a maze can be composed of nearly 80,000

directories which may require loading about 300MB from the disk, just to resolve the associated name.

Importantly, if even one of the corresponding directory entries is not found in-memory, in the filesystem cache, the process that invoked the system call on behalf of which the path resolution is performed would be put to sleep, blocked-waiting for IO.

The Attack We now describe how to trick the K-race routine (Figure 3) into opening a private inaccessible file. The routine invokes access and open K+1 times. For these total of $2K$+2 invocations, we create $2K$+2 directories `dir1`, `dir2`, ..., `dir2K+2`, each containing a new maze. We arrange thing such that `exit` points of odd mazes point to some public accessible file, whereas `exit` points of even mazes point to the inaccessible protected file we are about to attack. Finally, we generate a new symlink called `activedir` to point to `dir1`.

The attack is started by invoking the access_open K-race routine with the following filepath as an argument

```
activedir/sentry/lnk/lnk/.../lnk
```

This filepath is then passed along to the initial access call, which forces the K-race routine into the first maze. As a result, two things occur

1. The kernel updates the atime (access time) of every symbolic link it traverses during the name resolution, so by repeatedly examining the atime of `activedir/sentry` the attacker can learn that the respective access invocation is already in flight.
2. As mentioned earlier, the filepath being resolved (the maze) is big enough to insure that the kernel would have no choice but to fetch some of the relevant directory entries from disk; whenever this occurs the K-race routine would be suspended and put to sleep, and the attacker would get a chance to run and poll the atime of `activedir/sentry`.

Upon noticing that the atime has been updated, the attacker knows that the first access has begun. The attacker therefore switches `activedir` to point to `dir2`, and begins polling the atime of `dir2/sentry`. The initial access call is not affected by the change to `activedir` because it has already traversed that part of the path.

Eventually, the IO operations complete and the access finishes successfully. When the K-race calls the subsequent open, the exact same scenario occurs: the kernel updates the atime of `dir2/sentry`, the K-race routine sleeps on IO when loading parts of the respective maze that are not cached, the attacker consequently resumes and notices the updated atime of `dir2/sentry`, the attacker switches `activedir` to point to `dir3`, and the K-race routine completes the open successfully. This sequence of events repeats itself until all the system calls composing the K-race complete, and the attacker has managed to fool the K-race and open the protected file.

Enhancements In order to increase the confidence that some directory entries are not cached by the filesystem while the name resolution takes place, an attacker can run in parallel various unrelated IO intensive activities to wipe out the cache. A recursive string search in the filesystem

```
grep -r anystring /usr > /dev/null 2>&1
```

was found to be especially effective in this respect.

Finally, for completeness, Borisov et al. considered a K-race version that randomly flips the order of the calls to access and open within the strengthening loop (this is a valid and technically sound defense against their maze attack). They defeated this approach as well, by deducing which system call is currently being executed with the help of various kernel variables exported through the `/proc` file-system. For example, in Solaris 9, any process can read the current system call number of any other process from `/proc/pid/psinfo`.

4 Making Amplification Work

The maze attack is a generic way to systematically win TOCTTOU races. By utilizing complex file names, an attacker can slowdown the victim application, effectively single-step it, and gain a decisive advantage, which allows it to defeat the probabilistic K-race approach. In this section we show that this advantage is in fact *not* inherent. Defenders need not play by the rules that are dictated by the attacker. Rather, they can impose new rules that make it practically impossible for an attacker to win.

The key observation is simple and well known: system calls like open, stat, chdir, access, chown etc. that operate on a specified file name, are in fact $O(n)$ algorithms, where n is the number of components composing the name (n also embodies symlinks that are part of the name as well as the components of the soft links that must be recursively traversed). And so, in order to resolve an n-component name, the associated system call must sequentially iterate through n inodes. In the case of the K-race approach this is done K times, so the number of traversed inodes is actually $n \cdot K$. The order in which the traversal is performed is crucial for the success of the maze attack; assuming a file name of the form $/f_1/f_2/f_3$ (with no symbolic links along the way) and assuming $K = 2$, this order would be:

$$/,\ f_1,\ f_2,\ f_3,\ /,\ f_1,\ f_2,\ f_3$$

The general case is illustrated in Figure 5 (left); due to this type of a visualization we call this order *row-oriented*.

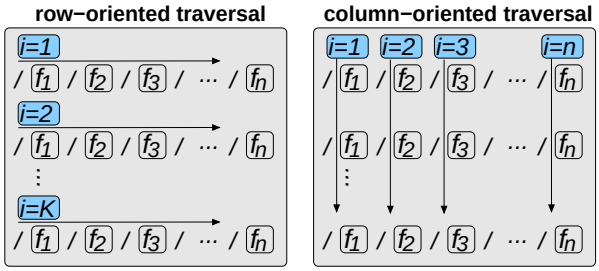


Figure 5: *The original row-oriented K-race traversal suggested by Dean and Hu (left) vs. our newly proposed column-oriented traversal (right). While Dean and Hu traverse the entire path on each access/open invocation, we traverse the path component by component, iterating through each specific element K times.*

The success of the K-race approach relies on the assumption that the rows remain identical from round to round. In contrast, the principle underlying the file-maze attack is to make n so big such that the time period between two "consecutive visits" in the inode associated with f_i would be relatively long; long enough to make it easy to violate the said assumption.

Our approach contends that row-orientated traversal, while seemingly dictated by the system call API, is not carved in stone. There is actually no technical difficulty preventing us from doing a different inode traversal that would better suit our needs. Specifically, column-oriented traversal is perfectly aligned with our intent to make it harder for an adversary to win a race. This approach is illustrated in Figure 5 (right). The idea is to resolve a path one component at a time, atom by atom, such that on each step we effectively conduct a kind of "short race" or "atom race", as part of the K-strengthening doctrine. This approach provides a clear advantage: an adversary no longer has control over the duration of the elapsed time between consecutive visits at f_i, e.g. the traversal order in the above example would be:

$$/,\ /,\ f_1,\ f_1,\ f_2,\ f_2,\ f_3,\ f_3$$

Thus, the race is made "fair" again and the respective inode would most probably be continuously present in the cache throughout the K-race, and almost certainly at least once during two consecutive iterations (which would be enough to defeat an attacker). The next section will show that even under the theoretical scenario where the attacker is *completely* and *instantaneously* synchronized with the defender, the attacker would have to wait tens to millions of years in order to subvert a $K = 9$ column-oriented defense.

We will now describe our algorithm in a bottom-up fashion (*all* source code included, as an indication of its simplicity). Doing a column-oriented traversal entails a price, which is having to handle the parsing of the file path ourselves when splitting it into atoms. For our purposes, however, the chop_1st function (as listed in Figure 6) was all that was needed in this respect. This function gets a relative path and "chops off" the first component while returning the remainder to the caller. By repeatedly invoking this function (using the remainder of the path from the previous invocation as the input to the current invocation), we gradually consume the file path in a column-oriented manner.

A second difficulty one faces when doing a user-level path resolution is having to handle atom components that are in fact symbolic links. To handle this caveat we used the simple is_symlink function (listed in Figure 7) that gets as input the atom that was just chopped off the prefix of the full file path. Note that by applying the lstat system call upon the given atom we make sure that the invoker is not forced to go through a maze. If this atom happens to be a symbolic link, then is_symlink copies the name of the target file to the memory pointed to by the appropriate argument; this would be later processed recursively. However, if the atom is a hard link (read: not a symlink), then the result of the lstat operation (as recorded by the given stat structure) will be used as a reference point within the race, when inodes are compared, as described next.

Having dealt with all the low-level details, we go on to consider how a race would actually be conducted when a hard link is finally encountered. Recall that the access permissions of a file are more than just the per-inode access bits (user/group/all read/write/execute etc.): they are the conjunction of all the permissions of each and every directory component along the path. For example, even if an inode indicates it is readable by all, if it nevertheless resides within a private directory, then obviously no one should be able to access the associated file. Therefore, before descending into the next directory component, the algorithm must verify that the invoker has the appropriate permissions. However, since this entails a TOCTTOU vulnerability, each such check must be K-strengthened.

Figure 8 shows how a per-atom K-race is conducted. Note that the security of our algorithm is reduced to the security of atom_race (all other functions are completely safe). The information encapsulated by the stat structure input was placed there by the is_symlink function that has just been invoked using the very same atom. Thus, it is likely that the inode (that is associated with the atom) is still in the cache. Further, since the atom is in fact an "atom" (one component file) that has just now been verified to be a hard link, it is also likely that the initial call to access and open would operate on the same inode. However, since there is a chance the attacker has managed to (1) unlink the previously lstated atom, and to (2) symlink it to a maze, strengthening steps are still required. The algorithm therefore continues into a K-loop that is almost identical to the one suggested by Dean and Hu (Figure 3).

```
char* chop_1st(char *path)
{
    // Find the end of the first component and
    // null-teminate it
    char *p = strchr(path,'/');

    if( p == NULL )
        return NULL;
    *p++ = '\0';

    // Handle multiple consecutive occurrences
    // of '/'. This ensures that the remainder
    // of the path is returned in a "relative"
    // form (without preceding slashes)
    for(; *p == '/'; ++p)
        ;

    // Returning NULL to indicate end of path
    return *p ? p : NULL;
}
```

Figure 6: *All the parsing is encapsulated in the above function, which gets a relative path as input, chops of the first component, and returns the reminder as a relative path. (A null return value indicates the entire path was consumed and so there is no reminder.)*

```
int is_symlink(const char  *atom,
               char         target[],
               struct stat *s,
               bool        *answer)
{
    int nb, l=PATH_MAX;

    DO_SYS( lstat(atom,s) );

    if( S_ISLNK(s->st_mode) ) {
        DO_SYS( nb = readlink(atom,target,l) );
        target[nb] = '\0';
        *answer    = true;
    }
    else {
        *answer = false;
    }

    return 0;
}
```

Figure 7: *We retrieve the name of the target file in case an atom is a symbolic link. Otherwise, the atom is a hard link in which case we record its inode information in the supplied* stat *structure for future reference. The return value indicates whether the* lstat *operations succeeded.*

All the original operations are still present. The difference is that now, on each iteration, the algorithm also verifies that the atom is still a hard link. This check is necessary in order for the defense to recover, if the attacker somehow managed to win the first race and to force the algorithm into a maze while doing the access and open operations. Since the lstating of an atom is an operation that is not affected in any way by the target that a symbolic link might have, our algorithm is not vulnerable in this respect. The only other additions we have made are (1) to check that fstating the initial file we open (fd1) yields identical information to that pointed to by s0, as the K strengthening rounds utilize s0 for the verification checks, and (2) to check that the lstated inode matches the initial inode, similarly to the original check with regard to the information that is retrieved by fstat.

Note that the two invocations of DO_CMP within the strengthening loop insures that all three stat structures are equal (s0 = s1 = s2), a check that is needed for the following reasons. By verifying that s1 is equal to s2, we know for a fact that the lstated and the opened files are one and the same, which means we deterministically force an adversary to win a race involving a non-symlink atom, on each round. This by itself, however, is not enough, as we must also make sure that s1 and s2 are equal to s0: failing to do so would make the K-loop meaningless, allowing an attacker to unlawfully open the file after winning only two races, as follows

1. The attacker creates a non-symlink file, `myfile`.
2. After is_symlink determines that `myfile` is not a symlink through the s0 stat structure, atom_race is invoked with `myfile` and s0 as arguments.
3. After the initial access in atom_race, the attacker must switch `myfile` to be a symlink to the file he wishes to unlawfully access. (Race #1)
4. After the initial open in atom_race, the attacker must switch back to its original file. (Race #2)
5. All the strengthening rounds can now execute without any further effort from the attacker.

We now have everything we need in order to implement a column-oriented K-race traversal. The access_open procedure we implement does this in a straightforward manner, as is shown in Figure 9. The first chunk of code simply makes sure that the traversal is only conducted with the help of relative names (that do not start with a slash). The second chunk is the traversal per-se. This part simply iterates through the atom components, one component at a time, and takes the necessary action according to whether the atom is a symbolic link or not. The latter is the simpler alternative: if the atom is a hard link, a short atom_race is conducted and the atom is directly opened. However, if the atom is a symbolic link, the algorithm calls itself recursively to handle the newly encountered composite path. In both cases, if a valid file descriptor is returned, the algorithm is allowed to continue to the next step after fchdiring to the current directory component. This strategy ensures us that there is a high probability that all relevant inodes reside in the cache during the time in which this is critical: when the K-race takes place.

```
int atom_race(const  char *atom,
              struct stat *s0)
{
  int i, mode;
  int fd1, fd2;
  struct stat s1 , s2;


  mode = S_ISDIR(s0->st_mode)
       ? X_OK /* directory */
       : R_OK /* regular   */  ;


  // 1- The initial access/open
  DO_SYS(       access(atom, mode     ) );
  DO_SYS( fd1 = open  (atom, O_RDONLY ) );
  DO_SYS(       fstat (fd1 , &s1      ) );
  DO_CHK(       DO_CMP(s0  , &s1      ) );


  // 2- The k strengthening rounds
  for(i=0; i<K; i++) {

    DO_SYS(       lstat  (atom, &s1      ) );
    DO_CHK(     ! S_ISLNK(s1.st_mode     ) );
    DO_SYS(       access (atom, mode     ) );
    DO_SYS( fd2 = open   (atom, O_RDONLY ) );
    DO_SYS(       fstat  (fd2 , &s2      ) );

    DO_SYS(       close  (fd2            ) );
    DO_CHK(       DO_CMP (s0  , &s1      ) );
    DO_CHK(       DO_CMP (s0  , &s2      ) );
  }


  return fd1;
}
```

Figure 8: *The given atom was just* lstat*ed and found to be a hard link, thus it is unlikely that an attacker would manage to set things up such that above would be thrown into a maze. If this has nevertheless happened, an additional* lstat *upon each iteration allows the algorithm to recover (compare with Figure 3).*

```
int access_open_2008(char *fname)
{
  int fd;
  char *suffix, target[PATH_MAX];
  struct stat s;
  bool is_sym;

  // 1- Handle the case where 'fname'
  //    is an absolute path.
  if( *fname == '/' ) {
    DO_SYS( chdir("/") );
    do { ++fname; } while(*fname == '/');
    if( *fname == '\0' ) // fname is rootdir...
      return open("/",O_RDONLY);
  }

  // 2- 'fname' is now relative
  while( true ) {

    suffix = chop_1st(fname);
    DO_SYS( is_symlink(fname,target,&s,&is_sym) );

    DO_SYS( fd = (is_sym
               ? access_open_2008(target)
               : atom_race(fname,&s)) );

    if( suffix ) {
      DO_SYS( fchdir(fd) );
      DO_SYS( close (fd) );
      fname = suffix;
    }
    else
      break;
  }

  return fd;
}
```

Figure 9: *A one-component-at-a-time column-oriented traversal prevents* access_open *from being abused and insures a fair atom-race is conducted when necessary. The heart of the function is the "? :" construct that decides whether to recurse over the next component (symlink) or to consume it (hard link).*

4.1 Implementation Notes

For brevity, the presented algorithm does not handle several minor details that should be addressed in a real implementation:

First, it lacks a defense mechanism against circular symbolic links. This can be easily incorporated within the procedure shown in Figure 9 in the exact same manner as it is done within the kernel, that is, by counting the number of traversed symbolic links and aborting the procedure if the count violates some predefined threshold.

Second, our algorithm opens a file for reading only. It does not allow the caller to specify other / additional flags to be passed along to open (such as O_RDWR, O_APPEND, etc). There is no technical difficulty preventing us from adding a "flags" parameter that allows this, as long as we provide special treatment for file truncation (O_TRUNC) and forbid file creation (O_CREAT). Truncation is problematic as the first open would truncate the file regardless of whether the real user has adequate permissions to do so; the solution is to access/open the file without O_TRUNC and, if successful, to ftruncate the resulting descriptor. File creation raises other (independent and well-known) TOCTTOU issues that are commonly associated with the problem of creating temporary files [11]; these are outside the scope of this paper.

Additional details that should be handled are (1) setting errno to EACCES when appropriate, namely, when DO_CMP and DO_CHK fail, (2) closing already opened file descriptors (if exist) upon errors, e.g., when fstat fails in Figure 8, and (3) saving and restoring the working directory before and after the invocation of access/open, to undo the effect of using fchdir.

The final item raises an important point we wish to make explicit: our access/open implementation is inadequate for multithreaded applications if some other thread

(different than the one performing the access/open) requires the working directory to remain unchanged, as this directory is shared by all threads. We note in passing that the relatively new system call openat (which opens a filepath relative to a given directory file descriptor [23]) would solve this problem, as it will eliminate the need for using fchdir; openat is proposed for inclusion in the next revision of POSIX [18].

5 Crafting the Hypothetical Attack

It should come as no surprise that the new access_open algorithm is completely immune from the maze attack, as the latter completely lost its timing ability: the attacker colossally fails to synchronize with the activities of the defender, and has no clue about when it would be most beneficial to unlink/link the targeted file in order to fool the defense. Nevertheless, while we believe it is improbable, it is still possible that somebody someday would come up with some surprising approach that would allow an attacker to achieve synchronicity once again. Hence, we seek a much stronger result.

To this end, we run an experiment in which the defender is completely "exposed": any attacker would be able to precisely know *which* actions are taken by the defender and *when*. In other words, our experiment fully reinstates the synchronicity capabilities to potential attackers, make these capabilities orders of magnitude more powerful and precise, and measures the probability attackers have to win a single round in light of the new approach; the bigger question being: Do file TOCTTOU races still pose a problem in the face of a column-oriented traversal? And if so, to what extent?

5.1 Exposed Defender

To answer this question we have implemented a defender program that provides information regarding its activities to any interested party through a shared-memory integer variable (instated with the help of SysV IPC facilities). The code of the defender is listed in Figure 10. It essentially does all of the defense-steps that are listed in Figure 8, but now each step is executed only after the defender publishes (through the shared integer) the next action to be performed. Note that the DO_SYS macro is redefined to record a system-call failure (instead of returning). This is done so that the defender process will not terminate. But it also means the defender maintains a fixed order of operations and thereby simplifies the code of the attacker (which is exempt from considering various corner cases). Importantly, an attacker may safely assume that the defender performs the same exact operations in the same exact order within each iteration.

In accordance to the column-oriented doctrine, the defender is operating on a file which is an atom, namely, composed of only one component that is arbitrarily called "target". Upon each iteration, after the operation sequence is over, the defender checks whether the attack was successful, and if so increments its losses count to be printed at the end of the run. The conditions that are asserted at the *end* of each iteration are identical to those that are checked *on the fly* within Figure 8, with only one addition: the defender is made aware beforehand of the inode of the private file that the attacker wants to read; obviously, an attack is successful only if it managed to fool the defender into opening this file.

5.2 Synchronized Attacker

We now go on to review the attacker's code, as given in Figure 11. Initially, the attacker must make sure that the file to be lstated is not a symbolic link. Additionally, since the defender is going to compare the inode of the lstated file to that of the opened file (which is the private file if the attacker gets his way), the 'target' file should point to the private file at this point. The attacker then waits until the defender is ready to lstat. As explained, the attacker's interest dictates that the defender would be able to successfully lstat the private file, and so the attacker must give it enough time to do so. This is also the reason for the next 'while' loop that ends when the defender finishes the lstat, or before, depending on the heuristic we have chosen to prematurely terminate the busy-waiting: We have evaluated a wide range of $T1$ values (see next section); Note that when $T1 = 0$, the busy wait period continues until the shared variable changes. But when $T1 > 0$ waiting may be shorter, as $T1$ bounds the number of busy-wait iterations and so the smaller it is, the shorter the wait.

After the defender lstats the private file, the real race is on, as the defender is about to check access and so the attacker must arrange things such that 'target' will point to an appropriate location. Additionally, the attacker aspires to slow down the defender by forcing him into a maze, in order to have a better chance of winning future races. The attacker therefore symlinks the target to a maze. Much like with the initial lstat operation, the attacker must now speculate when the access operation is already in flight. Once again, it may be advisable to end the busy waiting before the shared variable changes, and so another timer limit – $T2$ – is employed; We allow for two different limits so as to maximize the chances of success. The attacker is now hopeful that the defender has been forced into the maze, which would mean he can safely prepare towards the next open by linking to the private file. But even if the attacker was not successful, this is the correct thing to do in preparation for the defender's next lstat at the beginning of the next round.

```
bool sysfail;
#define DO_SYS( syscall ) \
    if( (syscall)==-1 ) \
       sysfail = true


void exposed_defender(ino_t private)
{
  struct stat s1, s2;
  int fd;

  sleep(1); // grace period for the attacker

  while( true ) {

    sysfail = false;

    *shared=LSTAT  ; DO_SYS(   lstat ("target", &s1     ));
    *shared=ACCESS ; DO_SYS(   access("target", R_OK    ));
    *shared=OPEN   ; DO_SYS(fd=open  ("target", O_RDONLY));
    *shared=FSTAT  ; DO_SYS(   fstat (fd      , &s2     ));
    *shared=CLOSE  ; DO_SYS(   close (fd                ));

    // The attacker is victorious only if all the
    // following conditions hold
    if( (! sysfail                ) &&
        (! S_ISLNK(s1.st_mode)    ) &&
        (  s1.st_ino == s2.st_ino ) &&
        (  s1.st_dev == s2.st_dev ) &&
        (  s2.st_ino == private   ) )
      defender_loss++;
  }
}
```

Figure 10: *The defender publicizes the operations about to be performed using a shared variable accessible to all.*

```
void synchronized_attacker()
{
  volatile int timer1, timer2;

  unlink( "target"           );
  link  ( "private", "target" );

  while( true ) {

    timer1 = timer2 = 0;

    // must wait for attacker to
    // lstat private file
    while( *shared != LSTAT )
       ;

    while( *shared == LSTAT )
      if(T1 && (++timer1 >= T1))
        break;

    // now we're really racing...
    // defender is about to access
    unlink ( "target"         );
    symlink( "maze", "target" );

    while( *shared == ACCESS )
      if(T2 && (++timer2 >= T2))
        break;

    unlink( "target"           );
    link  ( "private", "target" );
  }
}
```

Figure 11: *The attacker achieves synchronicity by polling the shared variable.*

6 Experimental Results

Our goal is to find out whether the column-oriented traversal technique is effective against the above hypothetical attack. (If this turns out to be the case, we can be reasonably sure that our solution would be effective in real-life scenarios where the defender is not exposed.)

6.1 Methodology

We obtain our goal by quantifying the expected time that a hypothetical attack should run in order to achieve k consecutive wins. Let this time be denoted B_k. If p is the probability for an attacker to win one round (iteration) within the exposed defender's loop, and t is the time it takes to conduct one round, then

$$B_k = t \cdot p^{-k} \tag{1}$$

because p^k is the probability for "success", and thus, $1/p^k$ is the mean of the geometric random variable that counts the number of trials until success is observed for the first time. For example, if a round takes one millisecond ($t = 1ms$), and the probability to win a round is 1/10 ($p = 0.1$), then B_2, B_3, B_4, and B_5 are 100 millisecond, 1 second, 167 minutes, and 28 hours, respectively. We approximate t and p by running the attack scenario and, upon termination, outputting (1) the duration of the attack, (2) the number of rounds conducted, and (3) the number of rounds lost. (We set t to be the average round duration, and p to be the ratio of rounds-lost to rounds-conducted.)

In order to increase the attackers' chances to win, we run the experiments on multiprocessors only. This way, attackers will have processors of their own to continuously and repeatedly attempt to fool the defender. In an effort to generalize the results, the experiments are conducted on older and recent machines, from different vendors, running different operating systems, as follows

Processor	Operating system	CPUs	Clock	Mem
UltraSPARC-II	Solaris 8	4	448 MHz	2 GB
Pentium-III	Linux 2.4.26	4	550 MHz	1 GB
Power4	AIX 5.3	8	1450 MHz	16 GB
Dual Core AMD	Linux 2.6.22	4	2200 MHz	8 GB
Intel Core 2 Duo	Linux 2.6.20	2	2400 MHz	4 GB

The 'maze' file we use is constructed to be the biggest that is possible on the respective OS, considering the aforementioned limits on the size of a filepath and the number of symbolic links it entails. Like Dean and Hu [12] and Borisov et al. [5] before us, we use a local file

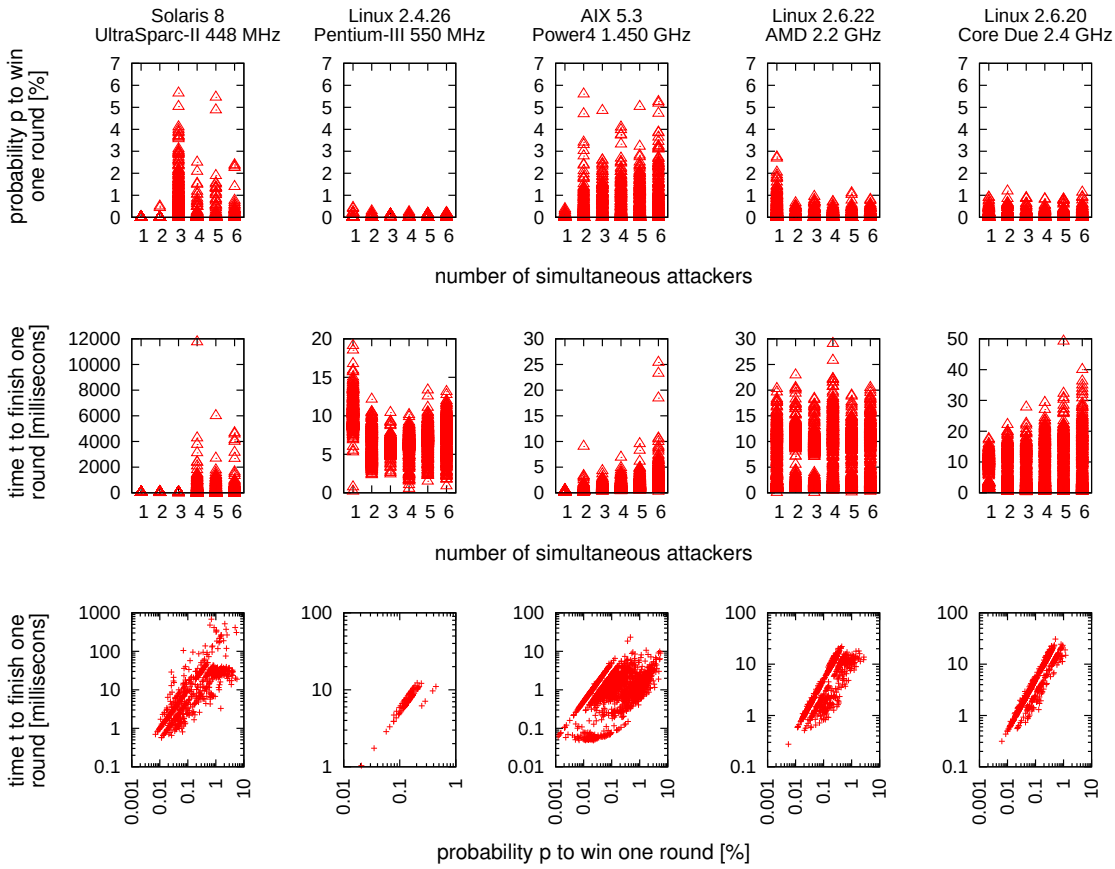


Figure 12: *The probability p for a synchronized-attacker to win a single round within the loop executed by the exposed-defender (top), the time t it takes an exposed-defender to complete a single round (middle), and the connection between the two (bottom).*

system for our experiments. These are the results we next describe; Afterwords, we also describe our additional findings from when running the experiments across NFS.

All the machines we use have a relatively big memory (that is, relative to the size of mazes), which as argued by Borisov et al., works against the attacker (more inodes can reside in core). However, we had appropriate permissions to change the Linux kernel running on the Pentium-III machine to one that only utilizes 256MB of the available memory. Other techniques we have experimented with in an attempt to increase the chances of the attacker to win are to simultaneously run multiple recursive grep-s during attacks in accordance to the suggestion by Borisov et al. [5], to launch attacks from within a huge directory that contains tens of thousands of files in accordance to Maziéres and Kaashoek's suggestion [24], and to simultaneously run several exposed-defenders on the same machine. We found that none of these techniques had a significant affect on the results, and therefore we do not report them here.

Conversely, Wei and Pu have recently shown that simultaneously running multiple identical attackers (attacking the same file) on a multiprocessor system, dramatically increases the chance of a TOCTTOU attack to prevail [38]. This technique turned out to be rather successful (from the attackers' perspective) and is therefore explicitly addressed below.

6.2 Results

Recall that the synchronized attacker has two tunable parameters — $T1$ and $T2$ — that place an upper bound on the two busy-wait loops the attacker must employ. We have independently set each of these two values to be either zero (no upper bound) or 2^j, where $j = 0, 1, 2, ..., 20$. This means that we conduct 484 (= 22^2) experiments for any specified number of simultaneous attacker (1–6), amounting to a total of 2,904 runs, per machine.

Local FS The top of Figure 12 shows the per-machine probability (expressed as percents) for multiple simultaneous synchronized attackers to win a single round. This is plotted as a function of the number of attackers, such that each point represents one of the aforementioned 2,904

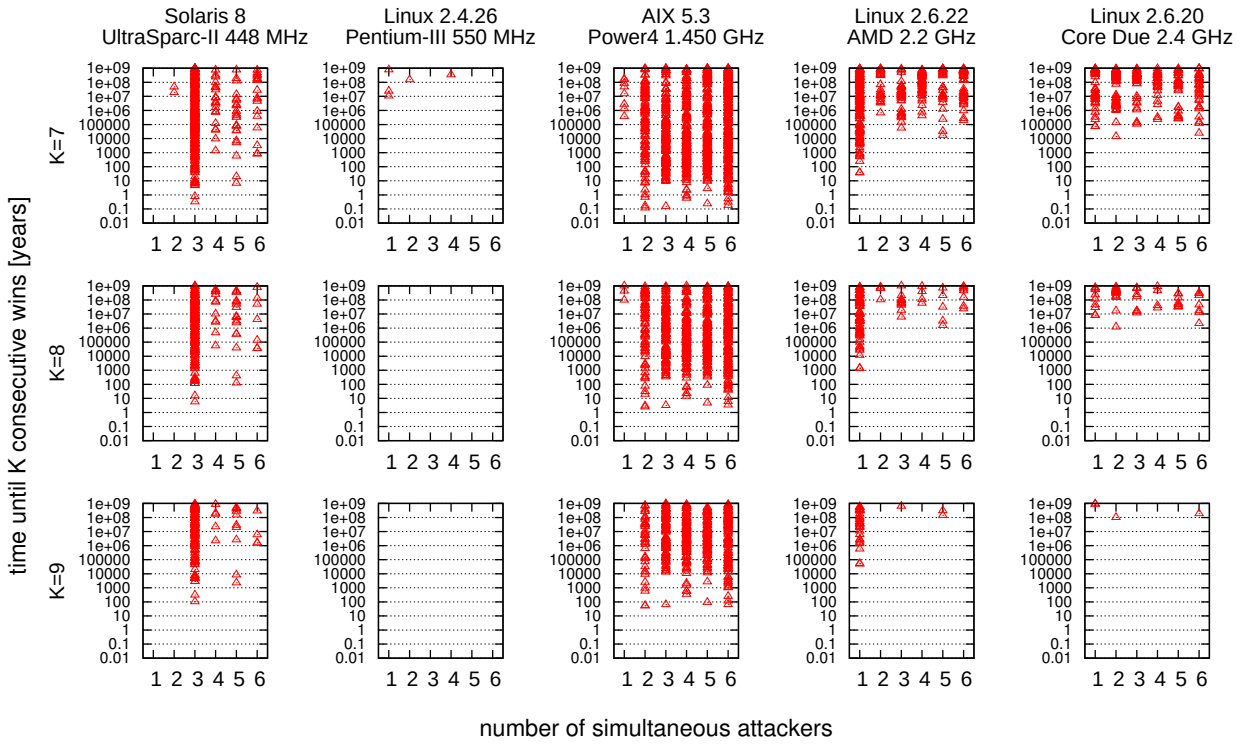


Figure 13: *The expected runtime of an exposed-defender loop until k consecutive rounds are won by the attacker (B_k), for k values of 7 (top), 8 (middle), and 9 (bottom).*

per-machine runs. Evidently, the probability can be quite high, culminating at nearly 6% on Sparc/Solaris (with three attackers) and on Power4/AIX (with two). Indeed, engaging more than one attacker appears beneficial, at least for these two machines.

The probability p to win a round is only one of two factors that determine the expected time B_k until a successful attack, as shown in Equation 1; The other factor is the time t it takes to complete the round, such that the bigger t is, the longer it would take to accomplish a successful attack. The middle of Figure 12 plots the values of t and shows that they too can be rather high with top values typically at tens of milliseconds, and outrageously, a few seconds in the case of Sparc/Solaris.

Importantly, the time to complete a round and the probability to win it are far from being independent variables. In fact, as shown at the bottom of Figure 12, there is a distinct linear connection between the two, which means the bigger the probability to win the round, the longer the round takes. Indeed, this makes perfect sense, as the prime objective of an attacker is to slow down the defender by throwing it into a maze. These are the two opposing side effects of the attacker's actions: maximizing p immediately translates to maximizing t, and so whatever ends up happening, the attacker inevitably contributes, to some extent, to making B_k larger.

Figure 13 assigns the t and p values of each of our experiments into Equation 1 in order to finally compute B_k, namely, the expected number of years an attack should execute until k consecutive rounds are won, for three different k values. When using $k = 7$ (the value recommended by Dean and Hu [12]) we see that a successful attack is potentially possible after a bit more than a month, in the case of Power4/AIX. Increasing k to be 8 and 9 raises the minimal expected duration to be more than 2.5 and 53 years, respectively, making the latter a safer choice in the face of our theoretical attack.

NFS Dean and Hu constrained their K-race evaluation to a local filesystem, saying that they did

> *"run some limited experiments attacking files across NFS and observed substantial numbers of successes. We chose not to continue these experiments, however, because NFS-accessed files are usually not the most security-critical, root privileges typically don't extend across NFS, the data displayed enormous variance depending on network and fileserver load."* [12]

But the set of attack experiments we conducted across NFS reveals that, while individual machines behave differently, the overall conclusion regarding the value of k does not dramatically change. The following table com-

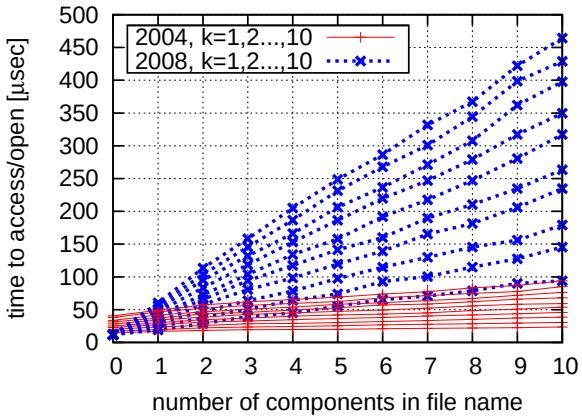


Figure 14: *Overheads of* access_open *(AMD / Linux 2.6).*

pares between minimal B_k values devised when running the attack on local and a networked filesystems (each table entry is the minimal result obtained across the 2,904 respective runs; values denote years, and, if bigger than 1000, are rounded down to the closest power of ten):

Platform		*Local FS*			*NFS*		
		k=8	k=9	k=10	k=8	k=9	k=10
SPARC	Solaris 8	5.8	103	10^3	0.3	2.6	21
P-III	Linux 2.4	10^9	10^{11}	10^{13}	0.1	0.8	5.8
Power4	AIX 5.3	2.5	53	951	10^8	10^{11}	10^{13}
AMD	Linux 2.6	10^3	10^4	10^6	∞	∞	∞
Intel	Linux 2.6	10^6	10^8	10^9	9.9	129	10^3

We see that machines can become less or more vulnerable to the hypothetical attack when it is conducted across NFS. The Pentium-III machine demonstrates the most notable change, being the least susceptible to the attack within a local file system (see also Figure 13) and becoming the most vulnerable with NFS. Conversely, with the Power4 machine, it's exactly the opposite, as it transitioned from being the most vulnerable to being nearly the least, second to only the AMD machine for which no attacker wins were observed with NFS.

Robustness We note that our evaluation methodology does not constitute a proof that the proposed solution is robust. Recall, however, that the attack described here is purely hypothetical, as defenders are not likely to publish their actions through shared memory for the sake of helping attackers. We therefore argue that it is reasonable to expect that real attackers will not do better. The assumption underlying this rationale is the following: Under the newly purposed access/open idiom, where system calls are repeatedly applied to a single-component relative filepath, attackers will be unable to systematically and consistently slow down the defender. If this assumption is true, then our method is robust, even in the face of slow devices and multiple attackers.

Overhead Figure 14 compares the overhead of the new access_open to that of Dean and Hu's, as a function of the opened file's number of components. The overhead is unsurprisingly linear. Clearly the older version is faster, due to the fewer system calls it invokes. But we contend that this is tolerable, considering the older solution is unsafe and that no other portable alternative exists.

7 Generalizing

A Check-Open Utility While the above ideas were demonstrated through the access/open race, their applicability is broader. The maze attack is a general method to deterministically win TOCTTOU races: given a check-use pair, if an attacker can manipulate the filename being checked (or any of its components), the attacker can utilize a maze to (1) synchronize with and (2) slowdown the defender, generating the ideal conditions for the attack to succeed. Conversely, the Column-oriented K-Race (CKR) is a general method to prevent this from happening by executing the check-use pair "atomically".

Nevertheless, programmers can not be expected to tailor a CKR for every legitimate check-use scenario. We therefore aspire to devise a generic utility function that can e.g., be added to libc. A first immediate step is to convert our access_open into a check_open function, by allowing the caller to pass the check operation as a pointer-to-function argument (getting an atom hardlink filename and returning zero upon success.) This operation would replace the call to access in Figure 8, allowing programmers to pass along access, or stat, or any other conceivable filename check operation they may require.

Note that the focus on open as the 'use' operation is not as limited as might initially seem: Recall that bindings of file descriptors to file objects are immutable and therefore completely immune from TOCTTOU attacks. Thus, once a valid file descriptor is safely opened and returned, the programmer can securely use the wealth of system calls that operate on file descriptors (fchown, fchmod, fchdir, fstat, ftruncate, etc.), rather than their respective insecure TOCTTOU-prone counterparts that operate on file names (chown, chmod, chdir, stat, truncate etc.).

A Check-Use Utility A completely different approach would be to convert access_open into a general purpose check_use utility. Here is how such an approach might work: Hardness amplification would be removed from the core algorithm and turned into a pluggable policy to be used by programmers at will. The part that remains is a user-mode path resolution traversal. As before, the algorithm would consume one component at time, fchdiring from component to component, and recursing on symlinks. The algorithm would *deterministically* make sure

it fchdirs to atom hard-links only (never directly to symlinks), by lstating the next atom directory (s_1), opening it, fstating the returned file descriptor (s_2), and making sure the s_1 and s_2 point to the same file object.

In addition to the filepath, check_use would get four pointer-to-function arguments F_{chk}^{dir}, F_{chk}^{link}, F_{chk}^{last}, and F_{use}^{last}. The first three are 'check' operations, respectively applied to each directory, symlink, and the last component in the given filepath, at the time the associated atom component is consumed by the path resolution traversal. Their input arguments are the atom name and the respective 'stat' structure and file descriptor (-1 for symlinks); their return value is zero to indicate the path-resolution may continue, or nonzero to indicate it should fail. The F_{use}^{last} encapsulates the 'use' operation, but otherwise has the same input and output as of the 'check' operations. All operations are invoked while the working directory of check_use is that of the atom that is currently being processed. Finally, the return value of check_use is the return value of the last operation that has failed, or that of F_{use}^{last} if all other operations succeeded.

With this design it is trivial to solve e.g., the race in Figure 2a. The garbage collector defines F_{chk}^{dir} and F_{chk}^{last} to always return 0, F_{chk}^{link} to always return -1, and F_{use}^{last} to unlink the atom file; thus, any symlink that is encountered along the way would make check_use fail, thereby insuring all deleted files are under the `/tmp/` directory, as required. Importantly, it does not matter whether the last (unlinked) atom is juggled by the attacker (symlink/hardlink to some sensitive file), as in this case the outcome would merely be that some link created by an attacker is deleted, a fact that does not affect the target file.

Eliminating the Probabilistic Aspect To reapply the probabilistic access/open solution under the check_use design, one would simply define F_{chk}^{link} to always return 0, F_{use}^{last} to return the file descriptor it gets as input, and F_{chk}^{dir} and F_{chk}^{last} to be (a slightly modified version of) atom_race from Figure 8. Notice, however, that there is actually no technical difficulty preventing us from going the extra mile and providing programmers with a library function that fully implements a deterministic and completely safe access check, in user mode: While the filepath is traversed, the associated 'stat' structure of each component, which is handed to the 'check' functions, contains the user and group ownership information as well as the user/group/world access permissions. Thus, given an arbitrary user and an atom's 'stat' structure (which is associated with an already opened file descriptor), we can deterministically decide whether the user has appropriate access permissions. While possibly a tedious task, portably implementing such a routine is nonetheless straightforward; as a library function, a single implementation would be shared by all and may have an additional benefit of potentially being more efficient than the probabilistic approach, which involves an $O(K)$ linear loop per filepath component. We are currently in the process of evaluating this alternative (as well as the one mentioned in the following paragraph) and expect to publish the results in the near future.

Adding Credentials to the Interface In contrast to the access/open race that has a satisfactory probabilistic solution, the race depicted in Figure 2b can only be solved with the help of a deterministic user-mode access (as was just described), since there is no system-call equivalent to access that a non-setuid program can use.[5] Indeed, defining F_{chk}^{dir} and F_{chk}^{last} to make use of the user-mode access and return 0 only if user "ann" has adequate permission, would suffice. Alternatively, instead of requiring the 'check' predicates to handle these details, check_use can be augmented to optionally get another parameter — a user id — and fail the path resolution process when an atom that the user is not allowed to open is encountered.

Summary By trading off some performance, we are able to devise a simple, yet powerful and expressive, interface that enables programmers to intuitively and securely combine a check-use pair into a single pseudo transaction, executed atomically for all practical purposes. While the entire implementation is straightforward portable user-mode, we effectively accomplish the vision of Maziéres and Kaashoek (Section 2.2) regarding a new "flexible" filesystem [24]. Notably, programmers gain explicit control of whether symlinks are followed when a file is opened, and are able to specify the credentials with which relevant system calls would operate.

A facility similar to the check_use function suggested above, if made a standard library function, would serve three purposes. First, it will allow programmers and designers to make conscientious decisions regarding the efficiency-safety tradeoff, e.g., between insecurely opening a file with a single open call, or doing it in user-mode, component by component, while enforcing repeated credential checks to avoid TOCTTOU races, or maybe making the effort to develop another alternative. Second, a well-designed check_use facility would encapsulate the execution of vulnerable check-use pairs. When the time comes and e.g. transactional filesystems (or other relevant improvements) are made more prevalent, the internal implementation can be replaced with a more efficient alternative. Thirdly, the inclusion of a check_use routine in the standard API would serve educational purposes, as new programmers get familiar with the API and through it become aware of the TOCTTOU problem.

[5] An attacker can choose to link `/mail/ann` to `/etc/passwd`, rather than to symlink. Thus, not following symlinks will not help.

Limitations Like the maze-attack, our approach works on already-existing-files only. The TOCTTOU problem associated with creating new files (notably, when wanting to create a new temporary file [11]) is still unresolved.

8 Conclusions

The POSIX API is broken: Its semantics inherently promote TOCTTOU races between check-use operations and make systems vulnerable to malicious attacks. Existing solutions can help locate these problems, but otherwise relate to future non-prevalent systems, leaving programmers to individually come up with solutions from scratch, to numerous variants of what is provably a hard and elusive problem. We suggest to alleviate the situation by providing programmers with standard generic abstractions that effectively bind check-use pairs into a single pseudo-atomic transaction. We further show that this goal can be obtained, to a large extent, in a portable manner, in user-mode, without changing the kernel.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and to Mary Baker, the shepherd of this paper. The first author would also like to thank Nikita Borisov, Alan Hu, Ethan Miller, Wietse Venema, and Erez Zadok for providing valuable and much appreciated feedback on earlier versions of this manuscript.

References

[1] A. Aggarwal and P. Jalote, "*Monitoring the security health of software systems*". In 17th *IEEE Int'l Symp. on Software Reliability Engineering (ISSRE)*, pp. 146–158, Nov 2006.

[2] K. Ashcraft and D. Engler, "*Using programmer-written compiler extensions to catch security holes*". In *IEEE Symp. on Security and Privacy (S&P)*, p. 143, May 2002.

[3] M. Bishop, *Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux*. Technical Report CSE-95-8, University of California at Davis, Sep 1995.

[4] M. Bishop and M. Dilger, "*Checking for race conditions in file accesses*". *Computing Systems* **9(2)**, pp. 131–152, Spring 1996.

[5] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, "*Fixing races for fun and profit: how to abuse atime*". In 14th *USENIX Security Symp.*, pp. 303–314, Jul 2005.

[6] D. Boulet, "*UNIX domain sockets*". URL http://everything2.com/index.pl?node_id=955968, Oct 2002. (Accessed Sep 2007).

[7] CERT Coordination Center, "*CERT Advisory CA-1993-17 xterm Logging Vulnerability*". URL http://www.cert.org/advisories/CA-1993-17.html, Nov 1993. (Accessed Jun 2007).

[8] H. Chen and D. Wagner, "*MOPS: an infrastructure for examining security properties of software*". In *ACM Conf. on Comput. & Communi. Security (CCS)*, pp. 235–244, Nov 2002.

[9] H. Chen, D. Wagner, and D. Dean, "*Setuid demystified*". In 11th *USENIX Security Symp.*, pp. 171–190, Aug 2002.

[10] B. Chess, "*Improving computer security using extended static checking*". In *IEEE Symp. on Security and Privacy (S&P)*, p. 160, May 2002.

[11] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, "*RaceGuard: kernel protection from temporary file race vulnerabilities*". In 10th *USENIX Security Symp.*, pp. 165–172, Aug 2001.

[12] D. Dean and A. J. Hu, "*Fixing races for fun and profit: how to use access(2)*". In 13th *USENIX Security Symp.*, pp. 195–206, Aug 2004.

[13] D. Engler and K. Ashcraft, "*RacerX: effective, static detection of race conditions and deadlocks*". In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 237–252, Oct 2003.

[14] D. Engler, B. Chelf, A. Chou, and S. Hallem, "*Checking system rules using system-specific, programmer-written compiler extensions*". In *USENIX Symp. on Operating Syst. Design & Impl. (OSDI)*, p. 1, Oct 2000.

[15] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "*Bugs as deviant behavior: a general approach to inferring errors in systems code*". In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 57–72, Oct 2001.

[16] B. Goyal, S. Sitaraman, and S. Venkatesan, "*A unified approach to detect binding based race condition attacks*". In *Int'l Workshop on Cryptology & Network Security (CANS)*, Sep 2003.

[17] A. J. Hu, "*On-line publication list*". URL http://www.cs.ubc.ca/spider/ajh/pub-list.html. (Accessed Jun 2007).

[18] A. Josey, "*The Open Group new API set proposals*". URL http://www.opengroup.org/ ... austin/plato/uploads/40/9756/NAPI_overview.txt, Feb 2006. (Accessed Dec 2007).

[19] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "*Detecting past and present intrusions through vulnerability-specific predicates*". In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 91–104, Oct 2005.

[20] C. Ko and T. Redmond, "*Noninterference and intrusion detection*". In *IEEE Symp. on Security and Privacy (S&P)*, pp. 177–187, May 2002.

[21] K-S. Lhee and S. J. Chapin, "*Detection of file-based race conditions*". *Int'l J. of Information Security (IJIS)* **4(1–2)**, Feb 2005.

[22] *The access(2) manual, FreeBSD*. URL http://www.freebsd.org/cgi/man.cgi?query=access.

[23] *openat(2) — Linux man page*. URL http://linux.die.net/man/2/openat.

[24] D. Maziéres and F. Kaashoek, "*Secure applications need flexible operating systems*". In *IEEE Workshop on Hot Topics in Operating Syst. (HOTOS)*, p. 56, 1997.

[25] W. S. McPhee, "*Operating system integrity in OS/VS2*". *IBM Systems Journal* **13(3)**, pp. 230–252, 1974. URL http://www.research.ibm.com/journal/sj/133/ibmsj1303D.pdf.

[26] "*National vulnerability database (NVD)*". URL http://nvd.nist.gov/. (Accessed Sep 2007).

[27] J. Park, G. Lee, S. Lee, and D-K. Kim, "*RPS: an extension of reference monitor to prevent race-attacks*". In 5th *Advances in Multimedia Information Processing (PCM)*, pp. 556–563, 2004. Lect. Notes Comput. Sci. vol. 3331.

[28] C. Pu and J. Wei, "*A methodical defense against TOCTTOU attacks: the EDGI approach*". In *IEEE Int'l Symp. on Secure Software Engineering (ISSSE)*, Mar 2006.

[29] F. Schmuck and J. Wylie, "*Experience with transactions in QuickSilver*". In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 239–253, 1991.

[30] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West, "*Model checking an entire linux distribution for security violations*". In *Ann. Comput. Security Applications Conf. (ACSAC)*, pp. 13–22, IEEE, Dec 2005.

[31] T. Sirainen, "*fdpass.c — File descriptor passing between processes via UNIX sockets*". URL http://code.softwarefreedom.org/projects/backports/browser/external/standalone/dovecot/current/src/lib/fdpass.c, 2002–2004. (Accessed Dec 2007).

[32] W. R. Stevens and B. Fenner, *UNIX Network Programming Volume 1: The Sockets Networking API*. Addison Wesley, 3rd ed., Nov 2003. Section 15.7.

[33] W. R. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, "*RFC 3542 – advanced sockets application program interface (API) for IPv6*". URL http://www.faqs.org/rfcs/rfc3542.html, May 2003. (Accessed Dec 2007).

[34] E. Tsyrklevich and B. Yee, "*Dynamic detection and prevention of race conditions in file accesses*". In 12th *USENIX Security Symp.*, pp. 243–256, Aug 2003.

[35] P. Uppuluri, U. Joshi, and A. Ray, "*Preventing race condition attacks on file-systems*". In *ACM Symp. on Applied Comput. (SAC)*, pp. 346–353, Mar 2005.

[36] "*United states computer emergency readiness team (US-CERT)*". URL http://www.kb.cert.org/vuls. (Accessed Sep 2007).

[37] J. Viega, J. Bloch, Y. Kohno, and G. McGraw, "*ITS4: A static vulnerability scanner for C and C++ code*". In *Ann. Comput. Security Applications Conf. (ACSAC)*, pp. 257–267, IEEE, Dec 2000.

[38] J. Wei and C. Pu, "*Multiprocessors may reduce system dependability under file-based race condition attacks*". In 37th *IEEE/IFIP Ann. Int'l Conf. on Dependable Syst. & Networks (DSN)*, Jun 2007.

[39] J. Wei and C. Pu, "*TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study.*". In 4th *USENIX Conf. on File & Storage Technologies (FAST)*, pp. 155–167, Dec 2005.

[40] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok, "*Extending ACID semantics to the file system*". *ACM Trans. on Storage (TOS)* **3(2)**, p. 4, Jun 2007.

[41] A. C. Yao, "*Theory and applications of trapdoor functions*". In 23rd *IEEE Symp. on Foundations of Computer Science*, pp. 80–91, 1982.

[42] K. Zeilenga, H. Chu, and P. Masarati, "*libraries/libutil/getpeereuid.c*". OpenLDAP source code URL http://www.openldap.org/devel/cvsweb.cgi, 2000–2007. (Accessed Dec 2007).