Portably Preventing File Race Attacks with User-Mode Path Resolution

DAN TSAFRIR
IBM T.J. Watson Research Center
TOMER HERTZ
Microsoft Research
DAVID WAGNER
University of California, Berkeley
and
DILMA DA SILVA
IBM T.J. Watson Research Center

The filesystem API of contemporary systems exposes programs to TOCTTOU (time of check to time of use) race-condition vulnerabilities, which occur between pairs of check/use system calls that involve a name of a file. Existing solutions either help programmers to detect such races (by pinpointing their location) or prevent them altogether (by altering the operating system). But 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 safely accomplished in a portable straightforward manner. The recent "filesystem maze" attack further worsens the problem by allowing adversaries to deterministically win races and thus refuting the common perception that the risk is small. In the face of this threat, we develop a new algorithm that allows programmers to effectively aggregate a vulnerable pair of distinct system calls into a single operation that is executed "atomically". This is achieved by emulating one kernel functionality in user mode: the filepath resolution. The surprisingly simple resulting algorithm constitutes a portable solution to a large class of TOCTTOU vulnerabilities, without requiring modifications to the underlying operating system. In contrast to our previous work, the new algorithm is deterministic and incurs significantly less overhead.

Categories and Subject Descriptors: D.4.6 [**Operating systems**]: Security and Protection—*Access controls*; D.4.3 [**Operating Systems**]: File Systems Management—*Access methods*; K.6.5 [**Management of Computing and Information System**]: Security and Protection—*Unauthorized access*

General Terms: Security, Algorithms, Performance, Measurement

Additional Key Words and Phrases: Race conditions, time of check to time of use, TOCTTOU

A preliminary version of parts of this paper appeared in [Tsafrir et al. 2008].

Authors' addresses: Dan Tsafrir, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: `dants@us.ibm.com`; Tomer Hertz, 1 Microsoft Way, Redmond, WA 98052; email: `hertz@microsoft.com`; David Wagner, University of California, Berkeley, CA 94720; email: `daw@cs.berkeley.edu`; Dilma Da Silva, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: `dilmasilva@us.ibm.com`.

1. INTRODUCTION

The TOCTTOU (time of check to time of use) race condition was characterized 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."* [McPhee 1974]

Dissecting a CERT advisory [CERT Coordination Center 1993], Bishop was the first to systematically show that filesystems with weak consistency semantics (like Unix and Windows) are inherently vulnerable to TOCTTOU races [1995; 1996]: 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 [Bishop and Dilger 1996; Viega et al. 2000; Chess 2002; Chen and Wagner 2002; Schwarz et al. 2005], (2) modifying the kernel to log all relevant system calls and analyzing the log, postmortem, to detect TOCTTOU attacks [Ko and Redmond 2002; Goyal et al. 2003; Lhee and Chapin 2005; Joshi et al. 2005; Wei and Pu 2005; Aggarwal and Jalote 2006], (3) having the kernel speculatively identify offending processes and temporarily suspend them or fail their respective suspected system calls [Cowan et al. 2001; Tsyrklevich and Yee 2003; Park et al. 2004; Uppuluri et al. 2005; Pu and Wei 2006], and finally (4) designing new filesystem interfaces to make it easier for programmers to avoid the races [Schmuck and Wylie 1991; Bishop 1995; Maziéres and Kaashoek 1997; Wright et al. 2007].

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 filesystems 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 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 advisories between 2000 and 2004 and found 20 reports concerning the issue, 11 of which provided the attacker with unautho-

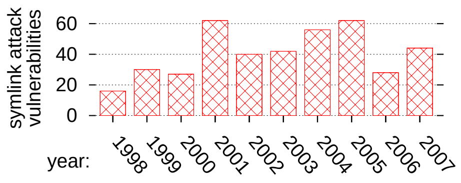

Fig. 1. *The National Vulnerability Database reports on 462 "symlink attack" vulnerabilities between the years 1998 and 2007. In 2001 and 2005 there were 73 and 106 reports, respectively; the associated bars are truncated. (Data retrieved on 22 Jan, 2008.)*

rized root access [US-CERT 2005; Wei and Pu 2005]. Figure 1 shows the yearly number of TOCTTOU "symlink attack" vulnerabilities reported by the National Vulnerability Database [NVD 2008]. 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 by Dean and Hu [2004], who implemented a transaction-like access_open routine that set out to solve a single race: the one that 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* [Borisov et al. 2005]) was to use *hardness amplification* as found in the cryptology literature [Yao 1982], 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 requiring 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. Shortly after, Borisov et al. [2005] orchestrated their *filesystem maze* attack and showed that an adversary can in fact win *every* race (hence making the assumption that $p < 1$ wrong). 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 [Dean and Hu 2004], Alan Hu concedes that

> *"The scheme proposed here has been beautifully and thoroughly demolished by Borisov et al. [2005]. 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." [Hu 2005]

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)" [Wei and Pu 2005] 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, surveying the existing solutions, and pointing out their shortcomings (Section 2), we go on to explain how probabilistic hardness amplification was applied to solve file TOCTTOU races, and why it has failed (Section 3). We then describe how the probabilistic approach can be made to work (Section 4), show how to turn it into a completely deterministic algorithm (Section 5), and quantify the associated overheads (Section 6). Finally, we generalize the solution (Section 7) and conclude (Section 8). The supplementary appendices A and B experimentally evaluate the robustness of the working version of the probabilistic solution.

2. MOTIVATION AND RELATED WORK

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

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,

root	attacker
	mkdir(/tmp/etc)
	creat(/tmp/etc/passwd)
readdir(/tmp)	
lstat(/tmp/etc)	
readdir(/tmp/etc)	
	rename(/tmp/etc,/tmp/x)
	symlink(/etc,/tmp/etc)
unlink(/tmp/etc/passwd)	

(a) garbage collector

root	attacker
lstat(/mail/ann)	
	unlink(/mail/ann)
	symlink(/etc/passwd,/mail/ann)
fd = open(/mail/ann)	
write(fd,...)	

(b) mail server

root	attacker
access(filename)	
	unlink(filename)
	link(secret,filename)
fd = open(filename)	
read(fd,...)	

(c) setuid

Fig. 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.*

the script can be tricked into deleting any arbitrary file, even if it attempts to prevent this from happening by explicitly ignoring symbolic links [Maziéres and Kaashoek 1997]. This is illustrated in Figure 2(a): 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 that appends a new message to the corresponding user's Inbox file [Bishop 1995; Bishop and Dilger 1996]. Before opening the Inbox, the server lstats it to rule out the possibility the user has replaced it with some symbolic link pointing to a file that lies elsewhere. Figure 2(b) 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 explains 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 archetypical, and arguably the most infamous, TOCTTOU flaw.[1] Figure 2(c) 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" [Man access(2) 2001].

2.2 Existing Solutions

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

[1]This race was reported by what is believed to be the first formal documentation of a file TOCTTOU vulnerability [CERT Coordination Center 1993]; it is described by almost all papers that address the TOCTTOU issue (see Section 2.2) when exemplifying the problem.

of 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 non-trivial vulnerabilities and bugs [Engler et al. 2000; Engler et al. 2001; Ashcraft and Engler 2002; Engler and Ashcraft 2003]. 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 [Bishop 1995; Bishop and Dilger 1996]. The tools ITS4 [Viega et al. 2000], Eau Claire [Chess 2002], and MOPS [Chen and Wagner 2002; Schwarz et al. 2005] 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 [2002] patched the kernel to log the required information and utilized it, postmortem, to feed a model that detects TOCTTOU flaws. A similar approach was later adopted by many following projects [Goyal et al. 2003; Lhee and Chapin 2005; Joshi et al. 2005; Wei and Pu 2005; Aggarwal and Jalote 2006]. Notable of these is the work by Wei and Pu [2005], which exhaustively enumerated all of Linux's TOCTTOU pairs,[2] and the IntroVirt tool, which supports virtual-machine checkpoint and replay, and could also be used for postmortem identification of TOCTTOU attacks [Joshi et al. 2005].

Dynamic Prevention. The kernel can be modified to discover TOCTTOU attacks as they occur and block them immediately. This approach was first taken by Cowan et al. [2001], in their "RaceGuard" system. They address stat/open TOCTTOU flaws where the program (1) checks to make sure that a candidate name for a temporary file does not yet exist, and then (2) creates the file under that name. 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, the open is aborted.

Later on, Tsyrklevich and Yee [2003] developed a more general approach that was capable of generically preventing most TOCTTOU attacks. They patched the kernel to suspend any process that interferes with a "pseudo transaction" (a check/use pair that access the same file), so that the worst outcome of a false-positive detection is a temporary suspension of the corresponding process. Others have built on their ideas [Park et al. 2004; Uppuluri et al. 2005; Pu and Wei 2006].

New API. The approaches outlined above all work within the existing filesystem API, to accommodate existing applications and operating systems. The complementary approach is to augment or change the API to make it easier to safely

[2]Wei and Pu [2005] (and later Lhee and Chapin [2005]) 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 easy to fix and therefore are outside the scope of this paper.

accomplish tasks that currently suffer from TOCTTOU issues. For example, to resolve the access/open race, Dean and Hu [2004] suggested that open could accept an O_RUID flag, which would instruct it to use the real (rather than effective) user ID of the process; alternatively, Bishop [1995] suggested adding a new faccess system call that would operate on a file descriptor rather than a file name.[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 2(b)). However, aside from being non-portable, this protects only the last component of the filepath: earlier components may still be symbolic links, and hence be juggled by an attacker (e.g. Figure 2(a)).

To obtain a more general solution, a bigger change is needed, such as replacing (or augmenting) Unix semantics with that of a transactional filesystem [Schmuck and Wylie 1991; Wright et al. 2007]. Atomicity would then ensure 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 [1997]. 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.

2.3 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 OSes portably support transactional filesystems (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. At the same time, as noted earlier, finding a portable user-mode solution to a given TOCTTOU race (if one exists) is often much harder than e.g. fixing a buffer overflow bug: Even experts that explicitly target a specific TOCTTOU problem are prone to getting it wrong.

[3]This suggestion was later raised again by Dean and Hu [2004]. But even so, we contend that it is impossible to implement, because the corresponding inode can be referred to by multiple paths, among which some are accessible to the user and some are not.

Consider for example the access/open race depicted in Figure 2(c). Tsyrklevich and Yee [2003] suggested two solutions to this flaw. They first suggested 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 [Chen et al. 2002]. Hence, a solution depending on user id juggling can be made to work, but is generally not portable."* [Dean and Hu 2004]

The second suggestion by Tsyrklevich and Yee was "to use fstat after the open instead of invoking access". The input of fstat is a file descriptor that, as such, 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 and 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. [2005] have mistakenly attributed the claim that this version is portable to Dean and Hu [2004]. 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 [Chen et al. 2002] (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. THE FAILURE OF HARDNESS AMPLIFICATION

Noting that no prior art helps programmers to portably resolve TOCTTOU vulnerabilities on existing systems, Dean and Hu [2004] took the first step towards

[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) [Stevens et al. 2003]. The end result is lack of portability and source code that is littered with ifdefs and conditional compilation tricks [Boulet 2002; Stevens and Fenner 2003; Sirainen 2004; Zeilenga et al. 2007].

a portable solution, explicitly focusing their efforts on the aforementioned access/open TOCTTOU race. After formally proving that no algorithm that uses access can ever deterministically overcome this race, they turned to explore a non-deterministic solution.

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 [Yao 1982]. The idea underlying hardness amplification is to use a problem which is computationally "somewhat hard", in order to 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 2(c)). 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.

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 (including 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 [Borisov et al. 2005] 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 filepaths (4KB is a

```
#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 *f)
{
  int fd1, fd2, i;
  struct stat s1, s2;

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

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

  return fd1;
}
```

Fig. 3. *The K-race routine checks on each strengthening round that the underlying file object, as represented by the inode (st_ino) and IO device (st_dev), remains the same.*

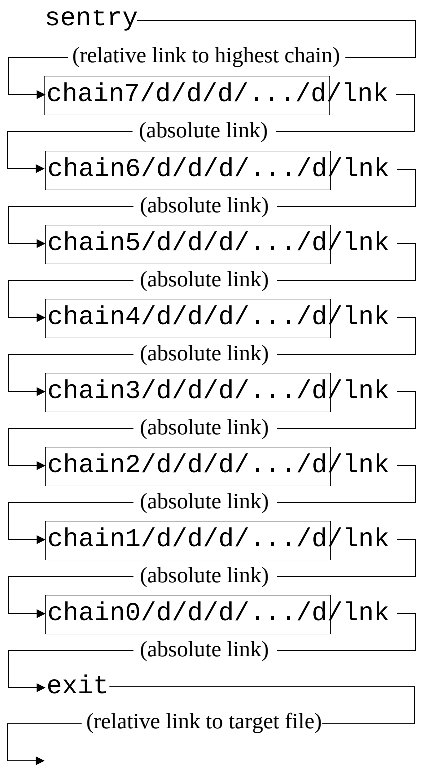

Fig. 4. *The structure of a maze with 8 chains. Arrows represents symbolic links. (Originally published by Borisov et al.; reproduced with permission.)*

typical value). Thus, chain$_0$ would be the longest path `chain0/d/d/d/.../d` that fits within the PATH_MAX limit. 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 that initiated the path resolution 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 things so 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 ensure 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 routine 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 system calls complete and the attacker has managed to fool the K-race routine 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` filesystem. For example, in Solaris 9, any process can read the current system call number of any other process from `/proc/pid/psinfo`.

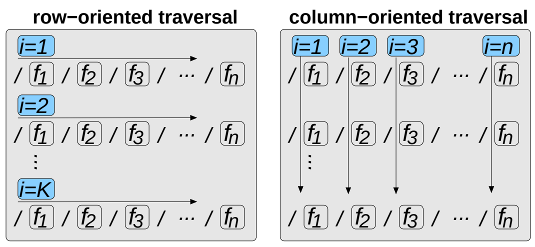

Fig. 5. *The original row-oriented K-race traversal suggested by Dean and Hu (left) vs. our newly proposed column-oriented traversal (right). While Dean and Hu traverse the entire path on each access/open invocation, we traverse the path component by component, iterating through each specific element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

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 e.g. 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.

4.1 Column-Oriented Traversal

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 $K = 2$, the K-race algorithm would traverse inodes in the order:

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

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

While row-oriented traversal might seem to be dictated by the system call API, we observe that it is not carved in stone. Nothing prevents us from traversing inodes in some other order 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 the path one component at a time, atom by atom, so 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. For instance, the traversal order in the above example would become:

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

Thus, the respective inode would 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). We will provide evidence 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.

4.2 Implementation

We now describe our algorithm in a bottom-up fashion (*all* the source code is included). Doing a column-oriented traversal entails a price: we must parse the filepath ourselves when splitting it into atoms. For our purposes, however, the chop_1st function (as listed in Figure 6) is all that we need. This function accepts 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 filepath in a column-oriented manner.

A second difficulty one faces when implementing user-level path resolution is having to handle atom components that are in fact symbolic links. We addresss this with 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 filepath. 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 saves the name of the target file to be processed later recursively. However, if the atom is not a symlink, then the result of the lstat operation (as recorded in the given stat structure) will be used as a reference point when inodes are compared, as described next.

If we determine that the atom is not a symlink, we must check its permissions explicitly. 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 introduces 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

```
char* chop_1st(char *path)
{
  // Find end of 1st component
  // and null-terminate it
  char *p = strchr(path,'/');

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

  // Strip multiple slashes,
  // ensuring 'p' points to
  // a relative path (no
  // preceding '/')
  for(; *p == '/'; ++p)
    ;

  // NULL means end of path
  return *p ? p : NULL;
}
```

Fig. 6. *All the parsing is encapsulated in the above function, which gets a relative path, chops of its first component, and returns the reminder as a relative path. (A null return value indicates the entire path was consumed.)*

```
int is_symlink(const char  *atom,
               char         target[],
               struct stat *s,
               bool        *is_sym)
{
  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';
    *is_sym    = true;
  }
  else {
    *is_sym = false;
  }

  return 0;
}
```

Fig. 7. *We retrieve the name of the target file if the atom is a symbolic link. Otherwise, we record the file's inode information in the supplied* stat *structure for future reference. The return value indicates whether the* lstat *operations succeeded.*

same atom. Thus, it is likely that the inode (that is associated with the atom) is still in the cache. Further, since we just checked that the atom is not a symlink, 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). All the original operations are still present. The difference is that now, on each iteration, the algorithm also verifies that the atom is still not a symlink. 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 ensures that all three stat structures are equal (s0 = s1 = s2), a check that is needed for the following reason. 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

```
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;
}
```

Fig. 8. *The given atom was just* lstat*ed and found not to be a symlink. Thus, it is unlikely that an attacker would manage to set things up such that* atom_race *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 *f)
{
  int fd;
  char *suffix, target[PATH_MAX];
  struct stat s;
  bool is_sym;

  // 1- If f is an absolute path
  if( *f == '/' ) {
    DO_SYS( chdir("/") );
    do { ++f; } while(*f == '/');
    if( *f == '\0' ) // it's root
      return open("/",O_RDONLY);
  }

  // 2- f is now relative
  while( true ) {

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

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

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

  return fd;
}
```

Fig. 9. *A one-component-at-a-time traversal prevents* access_open *from being abused. The heart of the function is the "? :" construct that decides whether to recurse over the next component (symlink) or to consume it (non-symlink).*

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 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. If the atom is not a symlink, we call atom_race, which opens the atom. 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 that, with high probability, all relevant inodes reside in the cache during the time at which it is most crucial: when the K-race takes place.

4.3 Evaluation

It should come as no surprise that the new access_open algorithm is immune from the maze attack, as the attacker can no longer 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 might come up with some surprising approach that would allow an attacker to achieve synchronicity once again. Hence, we seek a stronger result: we want to show that our algorithm does not rely upon the presumed hardness of synchronization.

To this end, we design an experimental framework in which the defender publicizes each system call it is about to execute through a shared memory variable that the attacker can read. This allows us to simulate attacks where the attacker manages to continuously synchronize with our defense. Details can be found in Appendix A.

We use this framework to experimentally evaluate the expected elapsed time such an attack should take before it prevails. We run these experiments on a range of platforms, operating systems, and runtime scenarios that try to maximize the attackers' chances to win. We find that a K value of 8–9 is enough to make the time to defeat our algorithm large enough for all practical purposes. A full description of our experiments is provided in Appendix B.

5. A DETERMINISTIC SOLUTION

Recall that the safety of our probabilistic solution reduces to the safety of the atom_race routine (Figure 8). We argued above that atom_race is probabilistically secure. However, if instead of atom_race we plug in a completely safe replacement, the entire algorithm would become deterministically secure. It turns out that such a replacement is, in fact, within reach. In this section we develop this replacement by first understanding how the kernel implements the required functionality (Section 5.1) and then by emulating it in user-mode (Section 5.2).

5.1 Enforcing File Permissions Within the Kernel

After coming up with the initial idea of how to fix the faulty hardness amplification algorithm published in [Dean and Hu 2004], we requested the authors of that paper to provide us with feedback regarding the new approach. Dean's skeptical response was that

> *"Trying to implement* namei *in user-space is going to be frighteningly difficult to get right."* [Dean 2007]

The term "namei" often refers to the centralized kernel mechanism that is used by all relevant system calls to resolve file names. This is inevitably done by sequentially breaking up the path to atom components and handling each component in turn. Dean's response correctly associates this kernel subsystem with our suggestion to emulate the mechanism by resolving file names ourselves in user-mode.

The namei mechanism is indeed quite complex: For example, at the time of this writing, the latest Linux kernel tree includes 47 source files named `namei.c` or `namei.h`, amounting to more than 20,000 lines of code [Torvalds 2008].[5] However, the complexity of the code arises from the need to handle many low-level details, such as mount points of different filesystems along the path, cache management of directory entries, name to inode association, locking and synchronization in the face of concurrency, failure scenarios, and more. Fortunately, these details remain exclusively the kernel's responsibility, making it possible for a user-mode algorithm to emulate the namei functionality in about 50 lines of portable code (Figures 6–9), and putting Dean's aforementioned concerns at ease.[6]

It is important to note in this context that, aside from resolving individual path components and following symbolic links, namei has an additional, crucial, responsibility: it makes sure that file permissions are honored. Indeed, *all* the system calls that accept a filepath as an argument depend on namei in this respect. But this should come as no surprise, as there is simply no other suitable mechanism that can accomplish the task. Namely, by definition, in order to make sure that a user has appropriate permissions to access a path, the user's credentials *must* be checked against *each* path component.

The POSIX standard associates a process with two types of credentials: "real" and "effective". The purpose of the real credentials is to identify the user that invoked the program. The effective credentials, on the other hand, are used by namei for enforcing file permissions. By default, these two credentials are initialized

[5]Nearly 3,000 lines of code implement the generic algorithm (`fs/namei.c`), and the rest are filesystem specific (e.g., `fs/ext3/namei.c`, `fs/ntfs/namei.c` etc.) and architecture specific (e.g., `include/asm-i386/namei.h`, `include/asm-powerpc/namei.h` etc).

[6]As further indication of the feasibility of the task of implementing namei in user-mode, we note that there exists a shell utility called `namei` that prints the type of each component along a given path. (Its manual states that "[t]his program is useful for finding a 'too many levels of symbolic links' problems" [Man namei(1) 1990].) The source file that implements the utility is 347 lines long, less than half belonging to the actual algorithm [Southwick 1990].

to the same value, which means namei actually uses the credentials of the *invoker* of the executable (=real). In contrast, if an executable is setuid, then the effective credentials are set to be those of the *owner* of the executable (which is typically root), regardless of the invoker's real credentials.

In light of this understanding, let us now consider how access is implemented by the kernel. (Recall that this system call was designed to allow privileged setuid programs to check whether the invoker has adequate permissions to access a given file.) A review of the relevant code within several operating systems quickly reveals that access does little other than invoking the namei mechanism,[7] such that it may even be considered (at least according to an associated comment in the referenced Darwin kernel) as merely a "tweak". Specifically, the access algorithm

(1) temporarily copies the real credentials into the effective credentials,
(2) invokes the namei mechanism (which thus uses the real credentials), and
(3) restores the original effective credentials.

The return value of access is determined by whether the namei invocation was successful or not, and inadequate permissions are a primary reason for failure.

5.2 Enforcing File Permissions in User-Mode

Equipped with a thorough understanding of the relevant kernel subsystems involved, let us now reconsider the original problem. We want to build a race-free version of

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

The above code ultimately invokes namei twice with exactly the same filepath as argument. The difference is that the first call executes with the real credentials applied, whereas the second does so with the effective ones. It is this redundancy in namei calls that is the source of our problem. If we could somehow merge the two (by, say, eliminating access and instead having open's namei execute with real credentials), our troubles would be over: in the absence of two operations, there can no longer be a race.

What prevents us from doing just that? Specifically, we have already paid the full price of re-implementing a large part of namei in user-mode (parsing the filepath, retrieving symbolic links along the way, and recursively following them; see Figures 6, 7, and 9) and shown it to be reasonable. So why not go the extra mile and enforce file permissions in user mode as well?

We argue that there is indeed no technical barrier to do that and thereby finally provide a *complete* user-mode emulation of namei. This is done by replacing the vulnerable call to access (along with the accompanying hardness amplification code

[7]For example, see the following internal kernel functions that implement the access system call within the specified operating systems (provided links point to the actual functions): sys_faccessat of Linux-2.6.23, at *http://lxr.linux.no/linux+v2.6.23/fs/open.c#L421*; kern_access of FreeBSD-7 at *http://fxr.watson.org/fxr/source/kern/vfs_syscalls.c?v=RELENG7#L1907*; caccess of OpenSolaris at *http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/syscall/access.c#62*; and access of Darwin-9.0 at *http://fxr.watson.org/fxr/source/bsd/vfs/vfs_syscalls.c?v=xnu-1228#L3489*.

of atom_race in Figure 8) with a direct check of each atom against the real user credentials. Doing so introduces some additional complexity, but in return the strengthening loops (and the now truly redundant access call) are no longer needed and can be omitted. Omitting the hardness amplification loop may have a positive effect on performance, as asymptotically it turns our $O(n \cdot K)$ algorithm into an $O(n)$ algorithm. This issue is further discussed in Section 6. The real benefit of this change, however, is that the security of our algorithm ceases to rely upon probabilistic arguments and becomes completely deterministic.

5.2.1 *Obtaining User Credentials.* The first thing one needs in order to directly enforce file permissions on a given user is to know this user's credentials. These are defined by POSIX to be

(1) the user id (uid),
(2) the user's group id (gid; sometimes denoted here as the *primary* group), and
(3) an additional array of *supplementary* group ids.

In Figure 10, we define the credentials structure to serve as an aggregator for these three fields. The get_cred function initializes this structure in a straightforward manner, as described next.

POSIX dictates that the getuid and getgid system calls always return the real credentials of the associated user, whereas geteuid and getegid always return the effective credentials. In other words, the two types of credentials coexist. The situation is fundamentally different for supplementary groups, as the kernel maintains only one instance of the associated array, and there is no notion of real vs. effective credentials.[8] Fortunately, in the context of a setuid program, this works to our advantage. When a user logs into a system, as part of the login process, the supplementary array is adequately populated with all the supplementary groups to which the user belongs. The array is than inherited across forks and execs, regardless of whether the execed program happens to be setuid or not. This means that the getgroups system call, which returns a copy of the supplementary array, is doing the right thing for our purposes, namely, it retrieves the credentials of the program's invoker, rather than its owner.

The only caveat we have to handle is that "it is implementation-defined whether getgroups also returns the effective group id in the array" [Man getgroups(2) 2004]. This means that, for a setuid program, one of the gids within the returned array might be the primary group of the owner of the program, which is obviously not a part of the credentials of any arbitrary user that just happens to invoke it. The solution is to filter out the effective gid by overwriting it with the primary gid of the user, as is done towards the end of get_cred. The possibility that the primary group would also appear in the supplementary array has no negative consequences, as will be discussed next.

5.2.2 *Enforcing User Credentials.* With the user credentials at our disposal, we go on to enforce them explicitly on each atom along the path. This is done with a new routine, termed atom_open, as listed in Figure 11. Recall that atom_open

[8]Indeed, POSIX explicitly states that "the setuid function [or any other set∗id system call for that matter] shall not affect the supplementary group list in any way." [Man setuid(2) 2004]

```
typedef struct credentials {
  uid_t  uid;
  gid_t  gid;
  gid_t *sup; // supplementary
  int    siz; // size of sup
} cred_t;



int
get_cred(cred_t *c,
         gid_t  a[],
         int    asiz)
{
  int j;
  gid_t egid = getegid();

  c->uid = getuid();
  c->gid = getgid();
  c->sup = a;

  DO_SYS( c->siz =
          getgroups(asiz,a) );

  // filter egid out
  for(j=0; j < c->siz; j++)
    if( c->sup[j] == egid )
      c->sup[j] = c->gid;

  return 0;
}
```

Fig. 10. *Assigning the credentials of the invoking user into* c *(*asiz *is the size of the array* a*, which is assumed to be big enough).*

```
bool find(gid_t gid, cred_t *c)
{
  int i, n = c->siz;
  for(i=0; i < n; i++)
    if( gid == c->sup[i] )
      return true;
  return false;
}

int
atom_open(char *atom, struct stat *s0, cred_t *c)
{
  int    fd, dir  = S_ISDIR(s0->st_mode);
  mode_t ok, mode = s0->st_mode;
  struct stat s;

  // 1) safe open: we *know* atom isn't a symlink
  DO_SYS( fd = open  (atom, O_RDONLY ) );
  DO_SYS(      fstat (fd  , &s       ) );
  DO_CHK(      DO_CMP(s0  , &s       ) );

  // 2) enforce credentials
  if( s.st_uid==c->uid )
    ok = mode & (dir ? S_IXUSR : S_IRUSR);
  else if( s.st_gid==c->gid || find(s.st_gid,c) )
    ok = mode & (dir ? S_IXGRP : S_IRGRP);
  else
    ok = mode & (dir ? S_IXOTH : S_IROTH);

  return (ok || c->uid==0) ? fd : (close(fd),-1);
}
```

Fig. 11. *By explicitly comparing the given user credentials against the atom's permissions, we are able to deterministically decide whether the associated user is allowed to access the atom or not.*

serves as a deterministic replacement to the older probabilistic atom_race. Thus, the prototypes of the two are almost identical. Specifically, the atom_open function gets the name of the soon-to-be-opened atom that was just lstated and verified as not a symbolic link. (The result of the latter lstat call was placed in s0.) However, in addition, atom_open also gets the credentials of the user on behalf of which it is about to do the open: The operation may succeed *only* if the user has adequate permissions, and will deterministically fail, otherwise.

The first part of the new algorithm opens the atom, fstats the resulting file descriptor, and makes sure that the outcome is identical to that of the lstat that took place just before the atom_open was invoked (s0). This idiom, of sandwiching open within a preceding lstat and a succeeding fstat that are verified to produce the same result, deterministically ensures that the lstated and opened files are one and the same. Thus, we know for a fact that the opened file is not a symlink, which along with the fact that the filepath itself is an atom, ensures no symlinks are in any way involved. We therefore conclude that a symlink attack is impossible, and

```
int access_open_det(char *f, cred_t *c)
{
  ...

  // everything is the same as in
  // the probabilistic column-
  // oriented solution (Figure 9)
  // except we're using atom_open
  // instead of atom_race...

  DO_SYS( fd = is_sym
    ? access_open_det(target,c)
    : atom_open(f,&s,c) ); // ...here!

  ...
}
```

```
void usage_example()
{
  ...

  // helper vars
  const int N = getgroups(0,NULL);
  gid_t arr[N];
  cred_t c;
  int fd;

  // safely access/open
  get_cred(&c, arr, N);
  fd = access_open_det(path, &c);

  ...
}
```

Fig. 12. *The initial column-oriented algorithm is made completely deterministic by merely adding a credentials argument and replacing* atom_race *with* atom_open. *(Compare with Figure 9.) The code snippet on the right exemplifies how to use the newer algorithm.*

that there is *no* TOCTTOU vulnerability within this sequence of operations.

The second part of our algorithm answers the question of whether the open operation that we have just performed is permitted under the given credentials:

—The first step is to check if the user id within the credentials structure agrees with that of the file. If this is the case, we examine the permissions mode, to see if the file (directory) is readable (searchable) by our user. A positive answer means the operation is legal: the user is allowed to open the file. But equally so, a negative answer means the user is forbidden, even if the group ids happen to agree, or if the file happens to be readable (searchable) by all.

—If user ids disagree, we perform a similar check using the associated groups. Here, however, it is not mandatory for the user's primary group to be equal to that of the file, as in the case of a mismatch, we also consult the supplementary array to see if the file's group is supplementary to the user.

—Finally, if both user and group ids disagree, we check whether the file is readable (searchable) by all. But even if this too fails, the function will nevertheless return a valid file descriptor if its caller is root (zero uid). This is the correct thing to do, as root is allowed to open any file.

—In case none of the above holds, we conclude that the user does not have adequate permissions to open the file. We therefore close the descriptor and return -1 to indicate failure.

We are now ready to implement the deterministic solution, which is shown in Figure 12. This algorithm is almost identical to our column-oriented probabilistic solution, with the sole difference that the current solution accepts an additional credentials argument to be passed along to atom_open, which serves as the deterministic replacement of atom_race. Being completely deterministic, the new algorithm finally ends the arms race.

6. OVERHEAD

The improved safety we are aiming to achieve does not come without a price, as every algorithm we have described entails some overhead penalty. In this section we attempt to quantify this overhead. For better understanding of the tradeoffs involved, we review all solutions suggested so far (here and in the related literature, whether robust or flawed). Specifically, we evaluate five access/open algorithms:

naive	The original vulnerable access/open idiom ("if access then open"). This serves as the baseline for comparison when assessing the overhead incurred by the other algorithms.
row	Employs "row-oriented" harness amplification by executing the naive algorithm for K times and verifying that access always succeeds and open always returns the same object. (Presented in [Dean and Hu 2004] and found vulnerable to the maze attack in [Borisov et al. 2005].)
unixdom	Forks a child that drops all extra privileges, performs the open, and returns the resulting opened file descriptor to the parent through a Unix-domain socket. (Suggested and ruled out as non-portable in [Dean and Hu 2004]; deterministically safe.)
col	Employs "column-oriented" hardness amplification on a per-atom basis, by resolving the path in user-mode and conducting a race for each atom. (Section 4; probabilistically safe.)
determ	Resolves the path and enforces credentials in user-mode. (Section 5; deterministically safe.)

Table I summarizes the main features of all of the above.

Algorithm	Portable	Deterministic	Immune to maze attack
naive	√		
row	√		
col	√		√
unixdom		√	√
determ	√	√	√

Table I. *Algorithms' main features.*

Recall that the running time of an algorithm that operates on a filepath is inevitably a function of n — the number of its components — as each atom must be individually resolved. Therefore, when assessing the algorithm's overhead, the question is which n to use? We look for the answer in a recent five-year study by Agrawal et al., which analyzed the yearly metadata snapshots of more than 60,000 Windows PC filesystem in a large corporation [2007]. Their findings regarding the distribution of filepath lengths are reproduced in Figure 13 and suggest that the relevant range for n is 1–12, as nearly 100% of the files fall within this range.

We therefore ran the following experiment in order to measure the overhead: Each of the five access/open algorithms was sequentially executed 100,000 times and repeatedly applied to a filepath of the length n ($n = 1, 2, ..., 12$). The same filepath was used throughout the measurement, which means its components were typically cached. For each algorithm/n pair, we measured the time it takes the corresponding

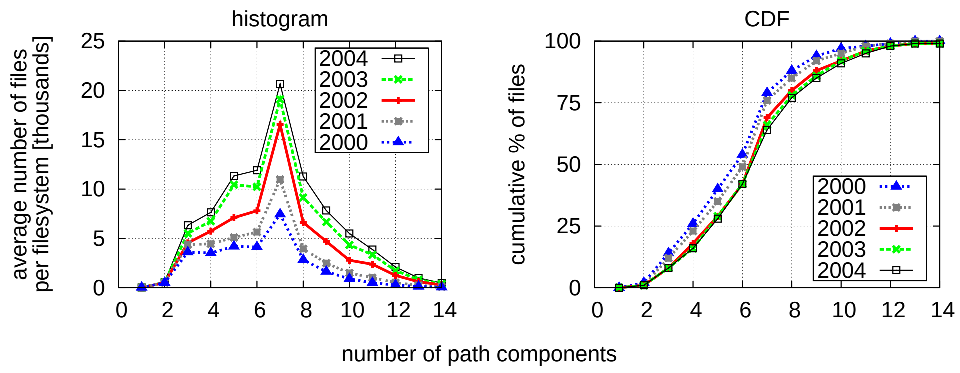

Fig. 13. *Yearly histogram and cumulative distribution function (CDF) of filepath lengths. Originally published in [Agrawal et al. 2007]; reproduced with permission.*

microbenchmark to complete and computed the associated per-operation average; these are the numbers we report here. The K values we used for benchmarking the *row* and *col* algorithms were $1, 2, ..., 10$. Measurements of *col* and *determ* included the time it takes to save and restore the current working directory before and after each invocation of the algorithm, respectively. Likewise, the measurement of *determ* included the time it takes to execute get_cred before each invocation. We ran the microbenchmark on five different platforms, as listed in Table II.

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

Table II. *Platforms used for the experimental evaluation.*

In Figure 14 we present the per-algorithm average execution times that were obtained on the fastest machine, as a function of the filepath length n. Clearly (and unsurprisingly), the execution time and n are linearly proportional.

The difference between *naive* and *unixdom* reflects the time it takes to fork a child; otherwise, the slopes of the associated curves are similar. While both *row* and *col* visit $n \cdot K$ inodes upon each invocation and are therefore both $O(n \cdot K)$, the latter significantly slower. The reason is that *col* does more per-inode work, a fact that manifests itself mostly in the larger number of kernel entries (system calls) it induces. Indeed, *row* invokes $O(K)$ system calls ($O(1)$ for each strengthening round), whereas *col* invokes $O(1)$ system calls for each atom within each round, amounting to a total of $O(n \cdot K)$. Due to this performance penalty, $col_{K=8}$ becomes slower than even *unixdom* with an n as little as four components.

The *determ* algorithm fixes this drawback of *col* by replacing the hardness amplification loop with a direct check of the user credentials. This eliminates the K factor from the execution time and transforms *determ* to be an $O(n)$ algorithm.

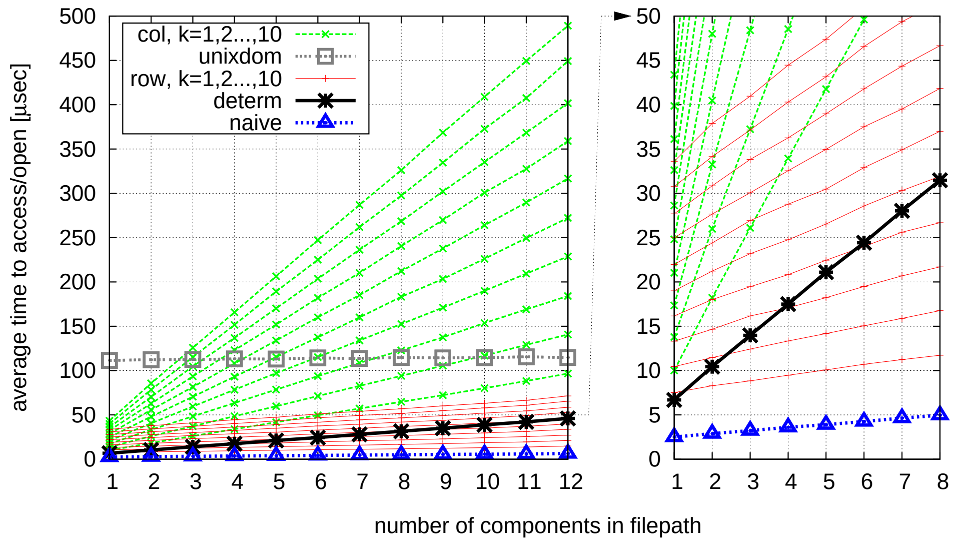

Fig. 14. *Overhead as measured on the Intel Core 2 Duo machine.*

And indeed, we can see that *determ* is faster than what $col_{K=0}$ would have been, had we chosen to evaluate it. Consequently, *determ* is significantly more efficient than *unixdom*, and is even faster than the faulty $row_{K=7}$ (that was suggested in [Dean and Hu 2004]) across the entire domain of interest.

The fact that the connection between the overhead and n is linear means we can model this connection by applying linear regression. This provides us with a superior way to accurately quantify and compactly present the results. Note that we use $n-1$ instead of n when we do the modeling.[9] Hence, given a slope A and a y-intercept B, the time to access/open a filepath with length n is modeled as $A \cdot (n-1) + B$. As a first step, Figure 15 plots the modeled slope and y-intercept of all the *row* and *col* curves from Figure 14 (a total of 20 curves).

Focusing on $col_{K=10}$ as an example, we see a y-intercept and slope of 44.4 and 40.5 μsec, respectively. Thus, the respective time to access/open a n=1 filepath (e.g., `/tmp`) is 44.4 μsec. Likewise, with n=6 (e.g., `/tmp/2/3/4/5/6`), the execution time is $40.4 + (n-1) \cdot 44.5 \approx 247$ μsec, coinciding with the associated empirical date shown in Figure 14. The linear connection apparent from Figure 15 reflects the fact that each strengthening round adds a constant amount of work. This allows us to express the slope and y-intercept of *row* and *col* as linear functions of K.

The full results of modeling the overhead are presented in Table III. The first part attests the accuracy of applying the linear model, with R^2 (coefficient of determination) values typically very close to 1, indicating a tight fit.[10] The slope of *determ* is roughly 10× bigger than that of *naive*, whereas *unixdom*'s slope is similar.

[9] A zero n relates to the root directory that gets special treatment by the *col* algorithm (immediately opened without strengthening rounds; see Figure 9), making it unsuitable for the model.
[10] For *row* and *col*, each R^2 table entry is the minimum across a set of eleven R^2 values: ten that are the result of applying the model to the individual Ks, and another from reapplying the model to these ten values, when expressing the parameters as a function of K.

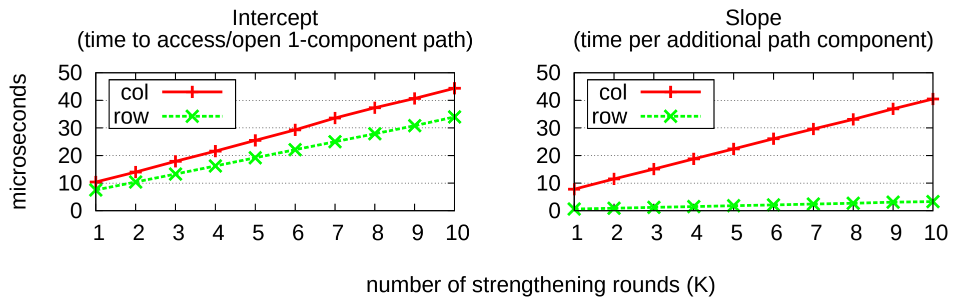

Fig. 15. *Applying linear regression to all the row and col curves shown in Figure 14 reveals a higher-level linear connection.*

This reflects the asymptotic penalty for when n goes to infinity. However, within the n range that is of interest, the performance is dominated by the y-intercepts: These are 2–3 times bigger for *determ* relative to the baseline, but are 1–2 orders of magnitude bigger for *unixdom*, making it slower in practice. Further comparing the parameters of *col* and *determ* supports our observation that the slope and y-intercept of *determ* are typically smaller than that of the $K = 0$ versions of *col*.

The final part of Table III presents a representative bottom line in the form of a typical slowdown penalty induced by each algorithm. We choose to use $K = 8$, as this is the minimal value found to be immune to the hypothetical know-all attack for a long enough period (see Appendix B). We use $n = 6$, as this is the median value of the filepath length distribution (see the point at which the CDF curves intersect the horizontal grid line associated with 50% in Figure 13). Under this setting, the deterministic solution is 3.3–5.8 times slower than the insecure idiom it is intended to replace. We note that our deterministic solution is simultaneously faster and more secure than the Dean-Hu K-race proposal ($row_{K=7}$).

7. GENERALIZING

So far, the focus of the discussion has been the access/open TOCTTOU problem. However, the applicability of our solution is broader. This is analogous to the maze attack, which originally targeted the access/open race only, but turned out to be effective against all check-use pairs for which filepaths can be manipulated by the attacker: indeed, mazes can be used to slow down and synchronize with such pairs, and create the ideal conditions for the attack to prevail. Likewise, user-mode path resolution is a generic way to prevent this from happening, by executing the pairs "atomically". The problem with this view, however, is that it would be counterproductive and unreasonable to expect programmers to come up with a slightly different path-resolution procedure for every legitimate check-use scenario. Luckily, this is unnecessary. We will show that our algorithm can be generalized to provide a generic way to secure a check/use sequence of filesystem operations.

7.1 Privileged Programs that are Not Setuid

The first step towards a generalization is making the algorithm usable by programs that are not necessarily setuid. This is needed because ordinary privileged programs are equally likely to suffer from TOCTTOU vulnerabilities. (For example,

Alg.	Duo Linux2.6	AMD Linux2.6	Pentium Linux2.4	PowerPC AIX5.3	Sparc Solaris8
a. R^2					
naive	0.999	0.982	0.987	1.000	0.919
determ	0.999	0.998	0.998	0.999	0.999
unixdom	0.908	0.495	0.927	0.919	0.394
row	0.996	0.962	0.987	1.000	0.970
col	1.000	0.992	0.999	0.999	1.000
b. y-intercept					
naive	2.5	3.5	8.8	11.2	19.5
determ	7.3	7.4	24.3	26.0	46.7
unixdom	111.8	193.3	504.7	561.6	3373.2
row	2.9K+4.6	3.8K+5.0	10.3K+15.5	11.9K+18.9	24.2K+34.3
col	3.9K+6.3	4.8K+8.7	14.6K+23.2	17.2K+27.2	34.0K+47.5
c. Slope					
naive	0.3	0.4	2.2	3.5	2.6
determ	3.5	3.7	13.1	13.9	23.3
unixdom	0.3	0.3	2.7	2.2	1.7
row	0.3K+0.3	0.4K+0.5	2.2K+2.1	3.5K+3.5	2.6K+2.5
col	3.6K+4.2	4.1K+4.4	13.6K+16.2	14.2K+17.7	24.9K+29.1
d. Slowdown (n=6)					
naive	1.0	1.0	1.0	1.0	1.0
determ	5.8	4.6	4.6	3.3	5.4
unixdom	26.9	34.6	26.9	20.0	112.4
row (K=8)	9.8	9.4	10.2	9.4	11.7
col (K=8)	48.0	42.2	39.6	28.6	48.7

Table III. *Modeling the overhead. The y-intercept (B) embodies the price of opening a n=1 filepath. The slope (A) reflects the cumulative penalty for each additional atom. The overall execution time is therefore $B + (n-1) \cdot A$ microseconds.*

the garbage collector and the mail server in Figure 2 are both non-setuid.) Fortunately, a reexamination of the relevant code (Figures 10–12) reveals that our objective is already achieved, as the identity issue is now fully decoupled from the algorithm through the use of the credentials structure. Specifically, the algorithm never consults any global environment settings regarding the identity in effect, but rather, exclusively uses the given credentials for this purpose.

The issue we do need to address is how to discover the required credentials. The get_cred function (Figure 10) may not be suitable for this purpose, as it is tailored for setuid programs: a non-setuid process that invokes get_cred would only get its own credentials. The solution, though, is straightforward. Figure 16 presents a get_cred-equivalent that can be used by any process, whether setuid or not.

Given a user name as an input argument, we can easily retrieve the corresponding uid and primary gid through getpwnam. The harder part is finding the associated

```
int get_cred_hard(cred_t *c,
                  char   *user,
                  gid_t   arr[])
{
  int i;
  int n=0;
  struct group *g;
  struct passwd *p = getpwnam(user);

  DO_CHK( p != NULL );

  // open group db
  setgrent();

  // iterate through group db
  for(g=getgrent(); g; g=getgrent())
    for(i=0; g->gr_mem[i]; i++)
      if(strcmp(g->gr_mem[i],user)==0)
        arr[n++] = g->gr_gid;

  // close group db
  endgrent();

  c->uid = p->pw_uid;
  c->gid = p->pw_gid;
  c->sup = arr;
  c->siz = n;

  return 0;
}
```

Fig. 16. *Discovering the credentials of an arbitrary user, regardless of setuid capabilities. (Compare with Figure 10;* arr *is assumed big enough.)*

```
enum {CHK_R=1, CHK_W=2, CHK_X=4};
bool chk_perm(struct stat *s,
              cred_t      *c,
              int          perm)
{
  mode_t r, w, x;
  r = w = x = s->st_mode;

  if( c->uid == 0 ) { // root
    return true;
  }
  else if( s->st_uid == c->uid ) {
    if( perm & CHK_R ) r &= S_IRUSR;
    if( perm & CHK_W ) w &= S_IWUSR;
    if( perm & CHK_X ) x &= S_IXUSR;
  }
  else if( s->st_gid == c->gid ||
           find(s->st_gid,c) ) {
    if( perm & CHK_R ) r &= S_IRGRP;
    if( perm & CHK_W ) w &= S_IWGRP;
    if( perm & CHK_X ) x &= S_IXGRP;
  }
  else {
    if( perm & CHK_R ) r &= S_IROTH;
    if( perm & CHK_W ) w &= S_IWOTH;
    if( perm & CHK_X ) x &= S_IXOTH;
  }

  return perm && r && w && x;
}
```

Fig. 17. *Check whether the given credentials convey the ability to access the file represented by the* stat *structure; required access permissions (*CHK *flags) are bitwise-or'd within* perm.

supplementary groups, as it mandates an exhaustive traversal through the entire group database. While doing this work can be time consuming, it appears there is no other portable solution. On the other hand, supplementary information is not always required (as is the case with the mail server and the garbage collector from Figure 2), allowing for a faster implementation. Note, however, that the difficulty of obtaining supplementary group information is not specific to our approach, as any process that wishes to retrieve such information regarding users different than its invoker faces the same difficulty. For example, utilities like `id` and `groups` [GNU Coreutils 2007] are typically implemented similarly to Figure 16.

The bottom line is that encapsulating the identity within the credentials structure and decoupling it from the core path resolution algorithm transformed the mail-server issue (Figure 2(b)) and the setuid issue (Figure 2(c)) into the same problem of executing some operation on some user's behalf (this is the classic "confused deputy" problem [Hardy 1988]). Our solution supports exactly this primitive: the programmer specifies the filesystem operation and the credentials of the user on whose behalf the operation is to be executed.

7.2 Opening the Terminal Point

Given a valid filepath p, its *terminal point* t is defined to be the last atom found after all symbolic links within p have been followed. All components prior to t are, by definition, directories or symlinks. Thus, the components of p can be divided into three categories: symlinks, directories, and a terminal point. The path resolution process (Figure 12) handles the first kind (symlinks) by recursively following them, whereas the other two (directories and t) are consumed by atom_open from Figure 11. The latter routine enforces the given credentials as follows: it checks that (1) directory atoms are searchable (executable), and that (2) other atoms are readable. (But as was just explained, these "other atoms" exclusively refer to the terminal point.) In both cases, if the credentials check is successful, the atom is opened for *reading* and the corresponding file descriptor is returned.

Notice that atom_open is problematic in two respects. First, it does not allow the caller to specify flags to be passed to the open system call. This needlessly constrains users who may want to open the terminal point for reading (O_RDONLY), writing (O_WRONLY), or both (O_RDWR), and may also want to supply additional open flags (like O_APPEND, O_NONBLOCK, O_NOCTTY, etc). The second problem is that t might be a directory, in which case our algorithm contains a security bug, as it checks if t is searchable, but opens it for reading.[11] The problem only applies to the terminal point (and not to prior directories that also seemingly suffer from the same symptom) because t's file descriptor is returned to the caller, whereas the other descriptors are used internally (for fchdiring) and are immediately closed.

The chk_perm function, listed in Figure 17, provides a way to overcome these flaws. Depending on the bitwise-or'd flags within its perm argument, chk_perm checks whether the given credentials allow to read (CHK_R), write (CHK_W), or execute (CHK_X) the current atom, as represented by the given stat structure.

Although any combination of the three CHK flags can be supplied, our use is more constrained. Specifically, perm can be either CHK_X (for all atoms other than t), or the return value of the convert function (for t itself). The latter function is shown in Figure 18. It gets standard open flags as given by the user. According to the specification of the open system call, the supplied flags must include exactly one of O_RDONLY, O_WRONLY, and O_RDWR; the function respectively converts them to CHK_R, CHK_W, or their or'd combination. (Zero is returned if all three flags are absent; this is correctly considered as an error by chk_perm in Figure 17.)

Having defined a separate routine to check atom permissions, we remove this functionality from atom_open, as is shown in Figure 19.

7.3 Generic Check/Use Utility

So far, we have seemingly focused on only one "use" activity: to *open* a filepath. But if this activity can be safely accomplished, it can be leveraged to overcome many TOCTTOU vulnerabilities, because the binding between file descriptors and the corresponding file objects is immutable and therefore inherently invulnerable

[11]This is wrong because a searchable directory allows users to open a file within the directory only if they know that file's name, whereas a readable directory allows users to list the names of the files within the directory (regardless of whether they can be opened or not). Hence, opening a searchable/unreadable directory for reading reveals information that should not be accessible.

```
int convert(int flags)
{
  if( (flags & O_RDONLY) == O_RDONLY )
    return CHK_R;

  if( (flags & O_WRONLY) == O_WRONLY )
    return CHK_W;

  if( (flags & O_RDWR  ) == O_RDWR   )
    return CHK_R | CHK_W;

  return 0;
}
```

Fig. 18. *Convert the user-supplied open flags to an encoding that is suitable for the use of the chk_perm function (Figure 17).*

```
int atom_open(char        *atom,
              struct stat *s0,
              int          flags)
{
  int    fd;
  struct stat s1;

  DO_SYS( fd = open  (atom, flags) );
  DO_SYS(      fstat (fd  , &s1  ) );
  DO_CHK(      DO_CMP(s0  , &s1  ) );

  return fd;
}
```

Fig. 19. *Securely open an atom, assuming the credentials are checked elsewhere. (Compare with Figure 11.)*

to TOCTTOU attacks. Consequently, once a valid file descriptor is safely opened and available, the programmer can securely use the wealth of system calls that operate on file descriptors (fchown, fchmod, fchdir, fstat, ftruncate, etc.), instead of their respective insecure TOCTTOU-prone counterparts (chown, chmod, chdir, stat, truncate etc.) that operate on file names.

Similarly to the observation regarding the "use" operation, so far, we have focused on only one "check" activity: of user credentials against file permissions. But like the ability to safely open a file, this too has broad applicability and covers a wide range of TOCTTOU flaws, as discussed in Section 7.1.

Nevertheless, for greater generality we seek to support other check/use operations. We attain this goal with the check_use function, shown in Figure 20, which serves as the final incarnation of our user-mode path resolution algorithm. Once again, the changes we have made relative to the previous version (Figure 12) mostly affect the prototype of the function (four additional parameters) and the "? :" construct (replaced by the if-else marked with 2.1 and 2.2). The only other change is that we now maintain a new Boolean variable called term that, together with the new cont parameter, help identify the terminal point in order to solve the problems raised in Section 7.2. For reasons to be discussed shortly, the check_use function is not used directly, but rather, through the following macro that sets cont to true:

```
#define CHECK_USE(fname,c,flags,tr,dat) check_use(fname,c,flags,tr,dat,true/*cont*/)
```

Note that, by definition, the terminal point is the last atom of the original filepath as provided by the programmer, unless this last atom is a symlink, in which case the terminal point would be the last atom of the symlink's target. Of course this statement is recursive, as the last atom of the said target might also be a symlink. Accordingly, throughout the recursion, a true value of cont indicates that the fname parameter is either (1) the original filepath as provided by the programmer, or (2) the target of its last atom, if this atom is a symlink. (Once again, this is a recursive statement.) When cont is true, then term would also be true if the last atom of the filepath is reached. Thus, when term is true, then either the current atom is the terminal point, or the current atom is a symlink pointing to the terminal point.

Having explained the roles of cont and term, it is now easy to understand the

```
typedef int (*trans_t)
   (char        *atom,
    struct stat *s,
    int          fd,
    void        *dat,
    bool         terminal_point);

int check_use(
  char *fname, // filepath to
               // check/use

  cred_t *c,   // with these
               // credentials

  int flags,   // open-flags for
               // terminal point

  trans_t tr,  // transaction to
               // apply to each
               // atom

  void *dat,   // passed to 'tr'

  bool cont)   // may contain
               // terminal point?
{
  int fd=-1;
  char *suffix, target[PATH_MAX];
  struct stat s;
  bool is_sym, term;

  // 1- If fname is absolute
  if( *fname == '/' ) {
    DO_SYS( chdir("/") );

  // continued on right
```

```
    do { ++fname; } while( *fname == '/' );
    if( *fname == '\0' )
      return open("/",O_RDONLY);
  }

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

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

    if( is_sym ) {  /* 2.1 */
      DO_SYS( tr(fname,&s,-1,dat,false)       );
      DO_SYS( fd =
        check_use(target,c,flags,tr,dat,term) );
    }
    else {          /* 2.2 */
      int prm = term ? convert(flags) : CHK_X;
      int flg = term ? flags : O_RDONLY;
      DO_CHK( chk_perm(&s,c,prm)           );
      DO_SYS( fd = atom_open(fname,&s,flg) );
      DO_SYS( tr(fname,&s,fd,dat,term)     );
    }

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

  return fd;
}
```

Fig. 20. *Adding a user-supplied function to be applied to each atom gives programmers fine-grained control on the path resolution process. Adding a flags parameter allows user to decide how the terminal point would be opened. (Compare with Figure 12.)*

new code in Figure 20. Branch 2.1 simply recurses into symlink targets, correctly identifying whether or not a target points to the terminal point. Branch 2.2 handles non-symlink atoms. For these, if term is true, then the terminal point is reached, in which case the new flags parameter is used to (1) check the permissions, and to (2) open the terminal point. If term is false, then the current atom is a non-terminal directory. We therefore check it is searchable (CHK_X) and open it for reading (O_RDONLY). Notice that this approach fixes the problems discussed in Section 7.2, namely, it allows the caller to specify open flags for the terminal point and it securely handles the case where the terminal point is a directory.

The remaining changes pertain to the trans_t identifier (top left of Figure 20), which defines the prototype of a user-supplied function (the "transaction") that is applied to every atom along the path. Its arguments are the name of the current

```
int collect_garbage(char *atom, struct stat *s, int fd, void *dat, bool term_point)
{
  time_t cutoff = time(0) - 72*3600;

  if( S_ISLNK(s->st_mode)  ) return -1;
  if( S_ISDIR(s->st_mode)  ) return 0;
  if( s->st_atime < cutoff ) return unlink(atom);

  return 0;
}
```

Fig. 21. *A user supplied "transaction" that safely implements the /tmp garbage collector from Figure 2(a).*

atom, the associated stat structure and file descriptor (-1 if the atom is a symlink), some arbitrary user-supplied data, and a Boolean indicating whether the given atom is the terminal point or not. The return value is treated by check_use like that of a system call, namely, -1 indicates failure and aborts the algorithm.

The fact that the "transaction" is applied to each atom along the path conveys the programmer fine-grained control over the path resolution process. With this ability it is possible to e.g., forbid the algorithm from following symbolic links ("check"), and by so doing, to categorically prevent symlink attacks. Likewise, the user-supplied function enables arbitrary "use" operations.

The end result is that e.g., securely implementing the /tmp garbage collector from Figure 2(a) becomes a trivial task, as exemplified in Figure 21: The prototype of the collect_garbage function agrees with that of the trans_t type, and so can be handed to the check_use algorithm. To ensure all deleted files are under the /tmp directory as required, the function refuses to follow symbolic links (first "if"), but otherwise allows the algorithm to descend into real subdirectories (second "if"). Upon reaching the path's terminal point, it **check**s the access time of the file (third "if"); if this occurred three days ago or before, it **use**s the supplied atom name to "collect" the corresponding file by unlinking it.

Note that it does not matter whether the last (unlinked) file was placed there by an attacker (e.g., symlink or hardlink to some sensitive file), as the outcome would merely be that some link created by an attacker is deleted by the garbage collector, a fact that does not affect the target file in any way.

Finally, note that our example "transaction" does not handle ".." atoms, which would have allowed attackers to escape /tmp, had they were able to use it directly. However, recall that the "transaction", as well as check_use itself, are merely helper functions that allow some higher-level privileged algorithm to avoid TOCTTOU races: the algorithm must still use them correctly. Indeed, when the garbage collection script traverses /tmp, the question of ".." atoms is simply irrelevant, as they are not generated by the traversal process. (A defensive programmer may nevertheless verify that the atom argument in Figure 21 is different than "..".)

7.4 Limitations

Multithreading. Our algorithm is inappropriate for multithreaded applications if in addition to the thread that performs the path resolution there exists another thread that requires the current working directory to remain unchanged. The problem occurs because the working directory is shared by all the threads, and so the said requirement would be violated by our usage of fchdir in Figure 20. (This is similar to the problem of multiple threads within a setuid program that share the "current identity", so a set∗uid system call invoked by one thread affects all the others.) We note, however, that the new openat system call [Man openat(2) 2006] will eliminate the problem. Specifically, openat opens filenames relative to a given directory file descriptor instead of relatively to the current working directory. And modifying our algorithm to use openat instead of fchdir would be trivial. The new system call is already implemented in Linux and Solaris, and is proposed for inclusion in the next revision of POSIX [Josey 2006].

File Creation and Truncation. The check_use function allows the caller to specify flags to be used when the terminal point is opened. This includes all the standard open flags, with two exceptions. The first is O_CREAT, which is used to create new files. The corresponding TOCTTOU problem is typically associated with the use of temporary files [Cowan et al. 2001]: a program comes up with a new candidate name f, checks that f is not already in use, and if so, creates it by opening f with O_CREAT. This procedure can be exploited by e.g. linking f to some sensitive file immediately after the check. Since the check relates to a file that does not yet exist and hence has no file descriptor, our algorithm has no way to prevent the attack. Also, a file created with O_CREAT would be created with the wrong owner.

The same vulnerability also applies to other system calls that create new directory entries in the filesystem: the rename, link, and creat system calls. The only solution we are aware of is to fork, drop privilege in the child, and (in the case of creat or O_CREAT) pass the resulting file descriptor from the child to the parent. As discussed in Section 2.3, this solution creates portability problems.

The second problematic flag of open is O_TRUNC, which is used to truncate files. The problem arises due to the open of the terminal point in Figure 19, which would truncate the file regardless of whether the subsequent calls to fstat and DO_CHK fail. The solution is to open the file without O_TRUNC and, upon success, to ftruncate the resulting file descriptor.

Executables. As noted earlier, files can be opened for reading, writing, or both. But unfortunately, they cannot be "opened for execution". Accordingly, there is no standard fexec system call, which means it is impossible to execute a program through a file descriptor. Thus, since execution is done strictly through file names, our algorithm is powerless to prevent the associated TOCTTOU problem.

7.5 Implementation Notes

For the sake of brevity and simplicity, we allowed ourselves to avoid handling several minor technical details, which should be addressed in a real implementation.

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

Additional details and corner cases that should be handled are (1) making sure that paths that end with a slash (e.g. `/x/y/`) are successfully opened only if the terminal point (`y`) is a directory, (2) setting errno to EACCES when DO_CMP or

DO_CHK fail, (3) closing already opened file descriptors upon error, e.g., when fstat fails in Figure 19, (4) handling the case where the root directory "/" is not readable/searchable by all, and finally, (5) saving and restoring the working directory before and after the invocation of the algorithm (as was done in Section 6).

8. CONCLUSIONS

The POSIX filesystem API is broken: its semantics inherently promote TOCTTOU races between check/use operations and make privileged applications vulnerable to malicious attacks. Existing solutions can help locate these problems, but otherwise relate to future non-prevalent systems. As a result, programmers must individually come up with solutions to numerous variants of a hard and elusive problem, which poses a challenge even for experts. We propose to alleviate the situation by providing programmers with a standard generic abstraction that effectively binds a check-use pair into a single pseudo-atomic transaction. We show that this goal can be obtained in a portable manner without changing the kernel, by trading off some performance and emulating the kernel's path resolution algorithm in user-mode.

A generic check-use facility like the one we propose, if made a standard library function, would serve three purposes. First, it will make these operations secure on existing platforms. Second, it will encapsulate vulnerable check-use pairs, so that when more efficient alternatives become prevalent (e.g. transactional filesystems) the internal implementation can be collectively replaced. Finally, the inclusion of a check-use function in the standard API will serve an educational purpose, as new programmers get familiar with the API and through it become aware of the TOCTTOU problem.

ACKNOWLEDGMENTS

We thank Nitin Agrawal for providing the data regarding the filepath length distribution. The first author would also like to thank Nikita Borisov, Drew Dean, Alan Hu, Ethan Miller, Wietse Venema, and Erez Zadok for providing valuable and much appreciated feedback on earlier versions of this manuscript. David Wagner was supported by the National Science Foundation (grant NSF CCF-0424422).

REFERENCES

Aggarwal, A. and Jalote, P. 2006. Monitoring the security health of software systems. In *17th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE '06)*. 146–158.

Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. 2007. A five-year study of file-system metadata. In *5th USENIX Conf. on File & Storage Technologies (FAST '07)*. 31–45.

Ashcraft, K. and Engler, D. 2002. Using programmer-written compiler extensions to catch security holes. In *IEEE Symp. on Security and Privacy (S&P '02)*. 143.

Bishop, M. 1995. Race conditions, files, and security flaws; or the tortoise and the hare *Redux*. Tech. Rep. CSE-95-8, University of California at Davis. SEP.

Bishop, M. and Dilger, M. 1996. Checking for race conditions in file accesses. *Computing Systems 9,* 2 (SPRING), 131–152.

Borisov, N., Johnson, R., Sastry, N., and Wagner, D. 2005. Fixing races for fun and profit: How to abuse atime. In *14th USENIX Security Symp.* 303–314.

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

CERT Coordination Center. 1993. CERT Advisory CA-1993-17 xterm Logging Vulnerability. URL http://www.cert.org/advisories/CA-1993-17.html. (Accessed Jun 2007).

Chen, H. and Wagner, D. 2002. MOPS: An infrastructure for examining security properties of software. In *ACM Conf. on Comput. & Communi. Security (CCS '02)*. 235–244.

Chen, H., Wagner, D., and Dean, D. 2002. Setuid demystified. In *11th USENIX Security Symp.* 171–190.

Chess, B. 2002. Improving computer security using extended static checking. In *IEEE Symp. on Security and Privacy (S&P '02)*. 160.

Cowan, C., Beattie, S., Wright, C., and Kroah-Hartman, G. 2001. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *10th USENIX Security Symp.* 165–172.

Dean, D. 2007. Solving TOCTTOU races. Private email communication.

Dean, D. and Hu, A. J. 2004. Fixing races for fun and profit: How to use *access(2)*. In *13th USENIX Security Symp.* 195–206.

Engler, D. and Ashcraft, K. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *19th ACM Symp. on Operating Syst. Principles (SOSP '03)*. 237–252.

Engler, D., Chelf, B., Chou, A., and Hallem, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *4th USENIX Symp. on Operating Syst. Design & Impl. (OSDI '00)*. 1.

Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th ACM Symp. on Operating Syst. Principles (SOSP '01)*. 57–72.

GNU Coreutils 2007. lib/getugroups.c. Source code of the GNU Core Utilities Library, URL http://ftp.gnu.org/gnu/coreutils/coreutils-6.9.tar.gz. (Accessed Feb 2008).

Goyal, B., Sitaraman, S., and Venkatesan, S. 2003. A unified approach to detect binding based race condition attacks. In *Int'l Workshop on Cryptology & Network Security (CANS '03)*.

Hardy, N. 1988. The confused deputy (or why capabilities might have been invented). *ACM Operating Syst. Review (OSR) 22,* 4, 36–38.

Hu, A. J. 2005. On-line publication list. URL http://www.cs.ubc.ca/spider/ajh/pub-list.html. (Accessed Jan 2008).

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

Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. 2005. Detecting past and present intrusions through vulnerability-specific predicates. In *20th ACM Symp. on Operating Syst. Principles (SOSP '05)*. 91–104.

Ko, C. and Redmond, T. 2002. Noninterference and intrusion detection. In *IEEE Symp. on Security and Privacy (S&P '02)*. 177–187.

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

Man access(2) 2001. The FreeBSD system calls manual. URL http://www.freebsd.org/cgi/man.cgi?query=access. (Accessed Jan 2008).

Man getgroups(2) 2004. The open group base specifications issue 6, IEEE Std 1003.1, 2004 edition. URL http://www.opengroup.org/onlinepubs/000095399/functions/getgroups.html. (Accessed Jan 2008).

Man namei(1) 1990. Shell utility manual. URL http://linuxcommand.org/man_pages/namei1.html. (Accessed Jan 2008).

Man openat(2) 2006. Linux programmer's manual. URL http://www.kernel.org/doc/man-pages/online/pages/man2/openat.2.html. (Accessed Jan 2008).

Man setuid(2) 2004. The open group base specifications issue 6, IEEE Std 1003.1, 2004 edition. URL http://www.opengroup.org/onlinepubs/000095399/functions/setuid.html. (Accessed Jan 2008).

Maziéres, D. and Kaashoek, F. 1997. Secure applications need flexible operating systems. In *6th IEEE Workshop on Hot Topics in Operating Syst. (HOTOS '97)*. 56–61.

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

NVD 2008. National vulnerability database. URL http://nvd.nist.gov/. (Accessed Jan 2008).

Park, J., Lee, G., Lee, S., and Kim, D.-K. 2004. RPS: An extension of reference monitor to prevent race-attacks. In *5th Advances in Multimedia Information Processing (PCM '04)*. 556–563. Lect. Notes Comput. Sci. vol. 3331.

Pu, C. and Wei, J. 2006. A methodical defense against TOCTTOU attacks: The EDGI approach. In *IEEE Int'l Symp. on Secure Software Engineering (ISSSE '06)*.

Schmuck, F. and Wylie, J. 1991. Experience with transactions in QuickSilver. In *13th ACM Symp. on Operating Syst. Principles (SOSP '91)*. 239–253.

Schwarz, B., Chen, H., Wagner, D., Lin, J., Tu, W., Morrison, G., and West, J. 2005. Model checking an entire Linux distribution for security violations. In *Ann. Comput. Security Applications Conf. (ACSAC '05)*. IEEE, 13–22.

Sirainen, T. 2002–2004. 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. (Accessed Dec 2007).

Southwick, R. S. 1990. misc/utils/namei.c. Source code of the util-linux library, URL ftp://ftp.kernel.org/pub/linux/utils/util-linux/util-linux-2.12r.tar.gz. (Accessed Jan 2008).

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

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

Torvalds, L. 2008. The Linux kernel source code version 2.6.23.14. URL http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.14.tar.gz. (Accessed Jan 2008).

Tsafrir, D., Hertz, T., Wagner, D., and Da-Silva, D. 2008. Portably solving file TOCTTOU races with hardness amplification. In *6th USENIX Conf. on File & Storage Technologies (FAST '08)*.

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

Uppuluri, P., Joshi, U., and Ray, A. 2005. Preventing race condition attacks on file-systems. In *ACM Symp. on Applied Comput. (SAC '05)*. 346–353.

US-CERT 2005. United States computer emergency readiness team: Vulnerability notes database. URL http://www.kb.cert.org/vuls. (Accessed Jan 2008).

Viega, J., Bloch, J., Kohno, Y., and McGraw, G. 2000. ITS4: A static vulnerability scanner for C and C++ code. In *Ann. Comput. Security Applications Conf. (ACSAC '00)*. IEEE, 257–267.

Wei, J. and Pu, C. 2005. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *4th USENIX Conf. on File & Storage Technologies (FAST '05)*. 155–167.

Wei, J. and Pu, C. 2007. Multiprocessors may reduce system dependability under file-based race condition attacks. In *37th IEEE/IFIP Ann. Int'l Conf. on Dependable Syst. & Networks (DSN '07)*.

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

Yao, A. C. 1982. Theory and applications of trapdoor functions. In *23th IEEE Symp. on Foundations of Computer Science*. 80–91.

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

APPENDIX

A. CRAFTING A FULLY-SYNCHRONIZED ATTACK

In Section 4 we have developed an algorithm that utilizes hardness amplification to make access/open probabilistically safe. By design, this algorithm is immune to the filesystem maze attack. Regrettably, this does not prove that our algorithm is safe against any possible future attack. It's still possible that somebody someday would come up with some clever way to regain synchronicity and overcome our probabilistic defense. We therefore attempt to obtain a 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, makes these capabilities significantly more powerful and precise, and measures the probability attackers have to win a single round in light of the new approach. We then study: Do file TOCTTOU races still pose a problem in the face of a column-oriented traversal? And if so, to what extent?

A.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 22. Essentially, it does all of the defense steps that are listed in Figure 8 (which performs the hardness amplification), 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 filepath that 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.

A.2 Synchronized Attacker

We now go on to review the attacker's code, as given in Figure 23. 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

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

void exposed_defender(ino_t private)
{
  struct stat s1, s2;
  int fd;
  char *f = "target";

  sleep(1); // grace period for attacker

  while( true ) {

    sysfail = false;

    // x is the shared variable
    *x=LSTAT ; DO_SYS(   lstat (f , &s1     ));
    *x=ACCESS; DO_SYS(   access(f , R_OK    ));
    *x=OPEN  ; DO_SYS(fd=open  (f , O_RDONLY));
    *x=FSTAT ; DO_SYS(   fstat (fd, &s2     ));
    *x=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++;
  }
}
```

Fig. 22. *The defender publicizes the operations about to be performed through a shared variable "x" 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( *x != LSTAT )
       ;

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

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

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

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

Fig. 23. *The attacker achieves synchronicity by repeatedly polling the shared variable.*

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.

B. EVALUATING THE PROBABILISTIC SOLUTION

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.)

B.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 listed in Table II.

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 (Section 3.2). Like Dean and Hu [2004] and Borisov et al. [2005] before us, we use a local filesystem 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 greps during attacks in accordance to the suggestion by Borisov et al. [2005], to launch attacks from within a huge directory that contains tens of thousands of files in accordance to Maziéres and Kaashoek's suggestion [1997], and to simultaneously run several exposed-defenders on the same machine. We found that none of these techniques had a significant effect on the results.

Conversely, Wei and Pu [2007] 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. This technique turned out to be rather successful (from the attackers' perspective) and is therefore explicitly addressed below.

B.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 24 shows the per-machine probability (expressed as a percentage) for multiple simultaneous synchronized attackers to win a single round. This is plotted as a function of the number of attackers, where each point represents one of the aforementioned 2,904 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. The middle of Figure 24 plots the values of t and shows that they too can be rather high with top values typically at tens of milliseconds, and surprisingly, 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 24, 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. Of course, increasing the time t to complete a round in turn contributes, to some extent, to making the total attack time B_k larger.

Figure 25 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 [2004]) 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.

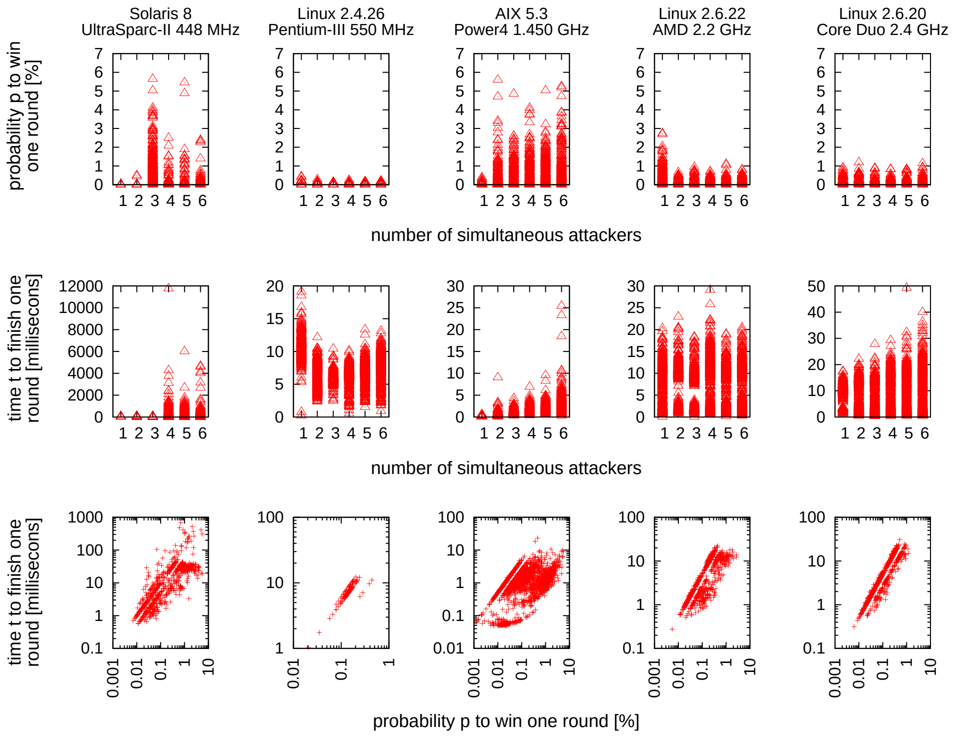

Fig. 24. *The probability p for a synchronized-attacker to win a single round within the loop executed by the exposed-defender (top), the time t it takes an exposed-defender to complete a single round (middle), and the connection between the two (bottom).*

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."* [Dean and Hu 2004]

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. Table IV compares between minimal B_k values devised when running the attack on local and 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).

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 filesystem (see also Figure 25) and becoming the most vulnerable with NFS. Conversely,

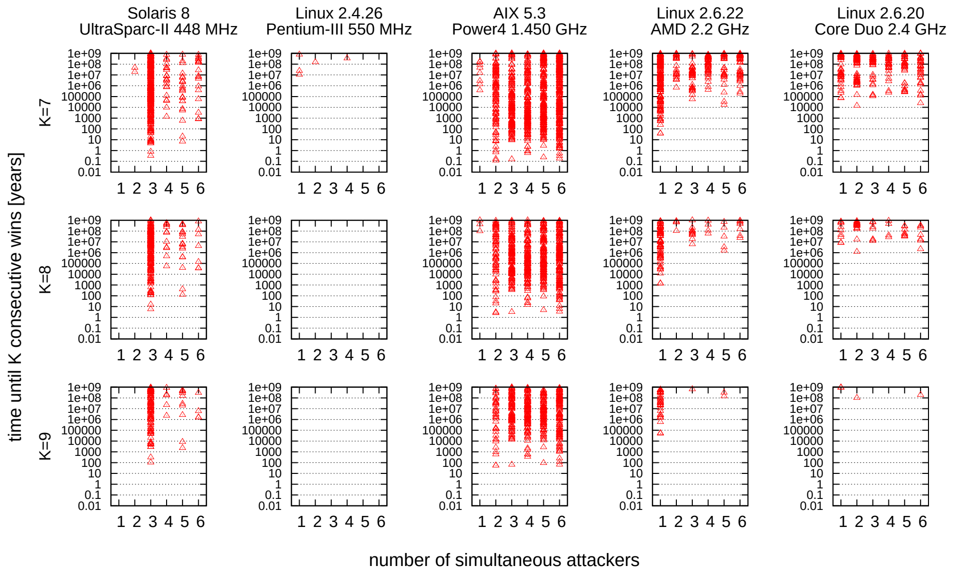

Fig. 25. *The expected runtime of an exposed-defender loop until k consecutive rounds are won by the attacker (B_k), for k values of 7 (top), 8 (middle), and 9 (bottom).*

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

Table IV. *Minimal B_k values, in years.*

with the Power4 machine, it is 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 proposed 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.