Portably Preventing File Race Attacks with User-Mode Path Resolution

Dan Tsafrir — *IBM Research*

Tomer Hertz — *Microsoft Research*

David Wagner — *UC Berkeley*

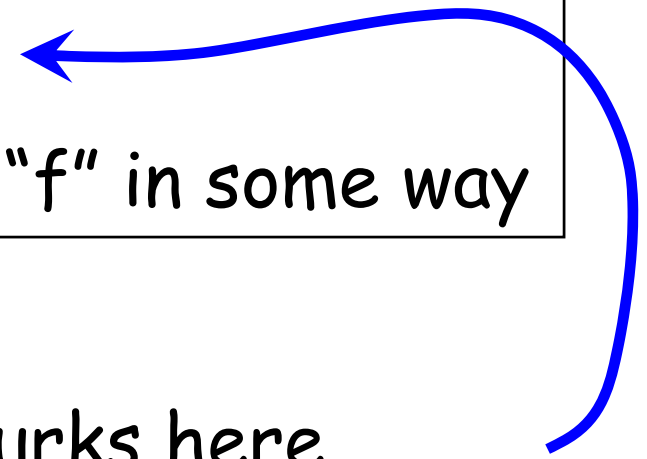
Dilma Da Silva — *IBM Research*
dilmasilva@us.ibm.com

TOCTTOU
(Time Of Check To Time Of Use)

- Given a "check use" pair of file operations:

1) **Check** something about filename "f"

2) Based on the result, **use** "f" in some way



- A TOCTTOU race condition lurks here

Example 1: *garbage collector*

root

readdir (*/tmp*)
lstat (*/tmp/etc*)
readdir (*/tmp/etc*)
lstat (*/tmp/etc/passwd*)
unlink (*/tmp/etc/passwd*)

attacker

mkdir (*/tmp/etc*)
creat (*/tmp/etc/passwd*)

rename (*/tmp/etc*, */tmp/x*)
symlink (*/etc*, */tmp/etc*)

Example 2: *mail server*

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

Example 3: *setuid*

root

```
if ( access (fname) == 0 )   {
      fd = open (fname)
      read( fd, …          ) …
}
```

attacker

```
unlink (                fname )
 symlink ( secret_file , fname )
```

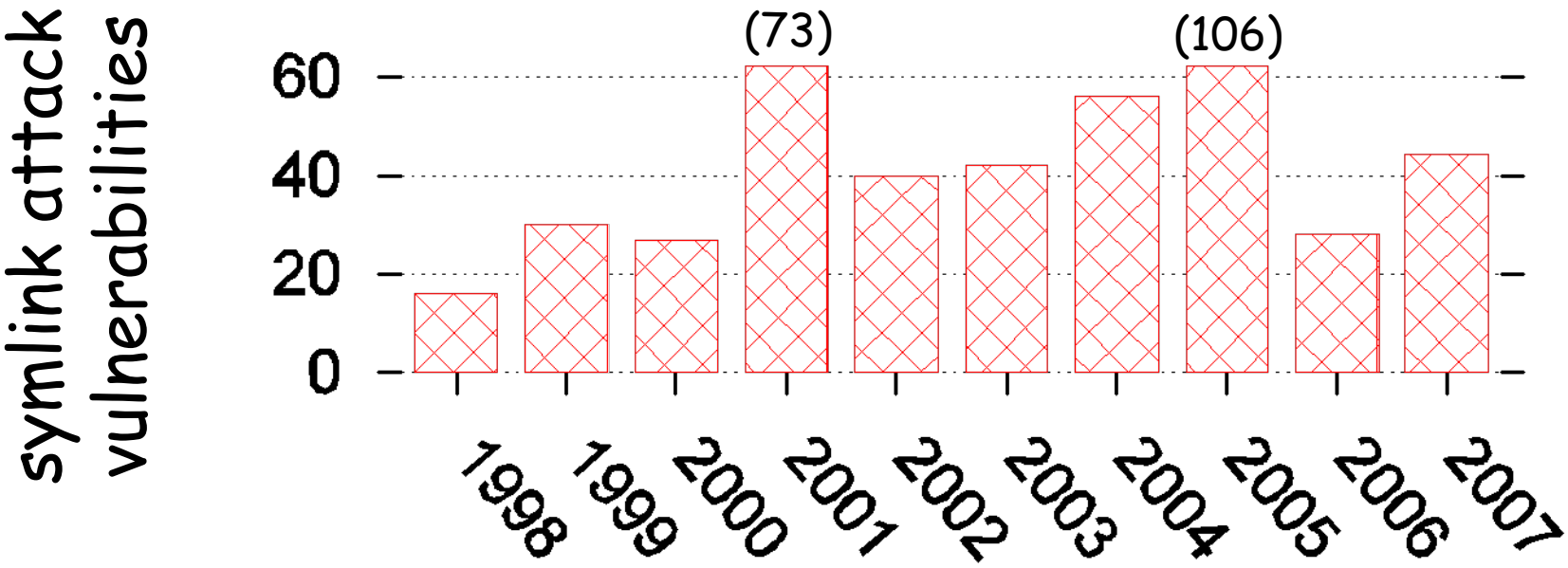
access() manual:

*"The access system call is a potential security hole due to race conditions and **should never be used**."*

Existing Solutions

- Static detection
  - Bishop 1995; Viega et al. 2000; Chess 2002; Chen & Wagner 2002; Schwartz et al. 2005;
- Dynamic detection
  - Ko & Redmond 2001; Goyal et al. 2003; Lhee & Chapin 2005; Joshi et al. 2005; Wei & Pu 2005; Aggarwal & Jalote 2006
- Dynamic prevention
  - Cowen et al. 2001; Tsyrklevich & Yee 2003; Park et al. 2004; Uppuluri et al. 2005; Wei & Pu 2006
- New API
  - Schumuck & Wylie 1991; Maziéres & Kasshoek 1997; Wright et al. 2007

Vulnerabilities are widespread



per-year data from the NVD
(National Vulnerability Database)

The problem: No solution for existing systems !

- Static detection
  - Finds races, doesn't fix them
- Prevention & new APIs
  - Not prevalent
- But once a race is found…
  - What should the programmer do?
- Much harder to solve than, say, buffer overflow
  - Even for experts

Previously suggested solutions for *access-open* race

1. Switch to "real" identity before *open*
   => Not portable [see *"Setuid Demystified", Usenix Security 2001*]
2. Do *open* + *fstat* to check ownership
   => Bug
3. Use Unix-domain socket to pass open fd
   => Not portable X 2
4. Use hardness amplification
   => Discussed next…

Hardness amplification

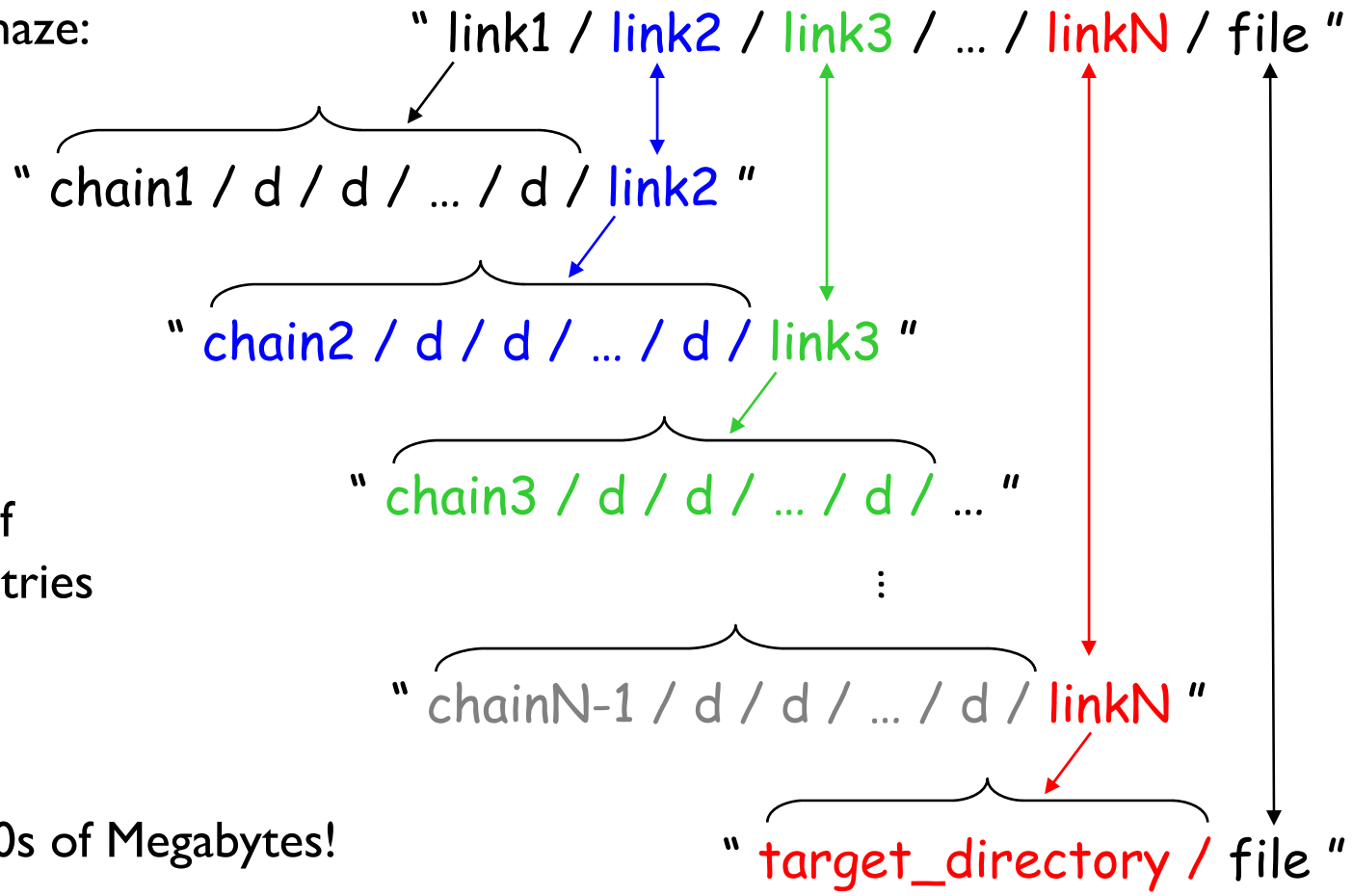
[Dean & Hu, Usenix Security 2004]


```
#define  SYS ( call )   if( (call) == -1 )   return -1

int access_open ( char * fname )  {

p       { SYS(          access ( fname, R_OK      )  );
          SYS(  fd1 =   open   ( fname, O_RONLY   )  );
          SYS(          fstat  ( fd1  , &s1       )  );

p^2K    for ( i = 1 ... K ) {
            { SYS(          access ( fname, R_OK      )  );
              SYS(  fd2 =   open   ( fname, O_RONLY   )  );
              SYS(          fstat  ( fd2  , &s2       )  );
            }
            SYS(          close  ( fd2              )  );
            CHK(          CMP    ( &s1  , &s2       )  );
        }
        return fd1;
}
```

Defeating the *K*-race

[Borisov et al., Usenix Security 2005]

- Filesystem maze:
- Composed of 10,000s of directory entries
- Requires 100s of Megabytes!



Defeating the *K*-race

[Borisov et al., Usenix Security 2005]

- Maze takes a long time to traverse
  - Often results in going to disk
- Path traversal updates symlink access time
  - Attacker can poll symlink access time and figure out what the defender is doing
- The attack (tricking victim to open 'secret')
  - Just before access() set target file to be public
  - Just before open() set target file to be 'secret'

Defeating the *K*-race

[Borisov et al., Usenix Security 2005]

Maze attack:

Prepare K+1 mazes that point to a public file

Prepare K+1 mazes that point to a private file

for(i = 0 … K)

1) Link "link1" to "chain1" of i^{th} public maze
   Poll atime

2) Link "link1" to "chain1" of i^{th} private maze
   Poll atime

Consequences

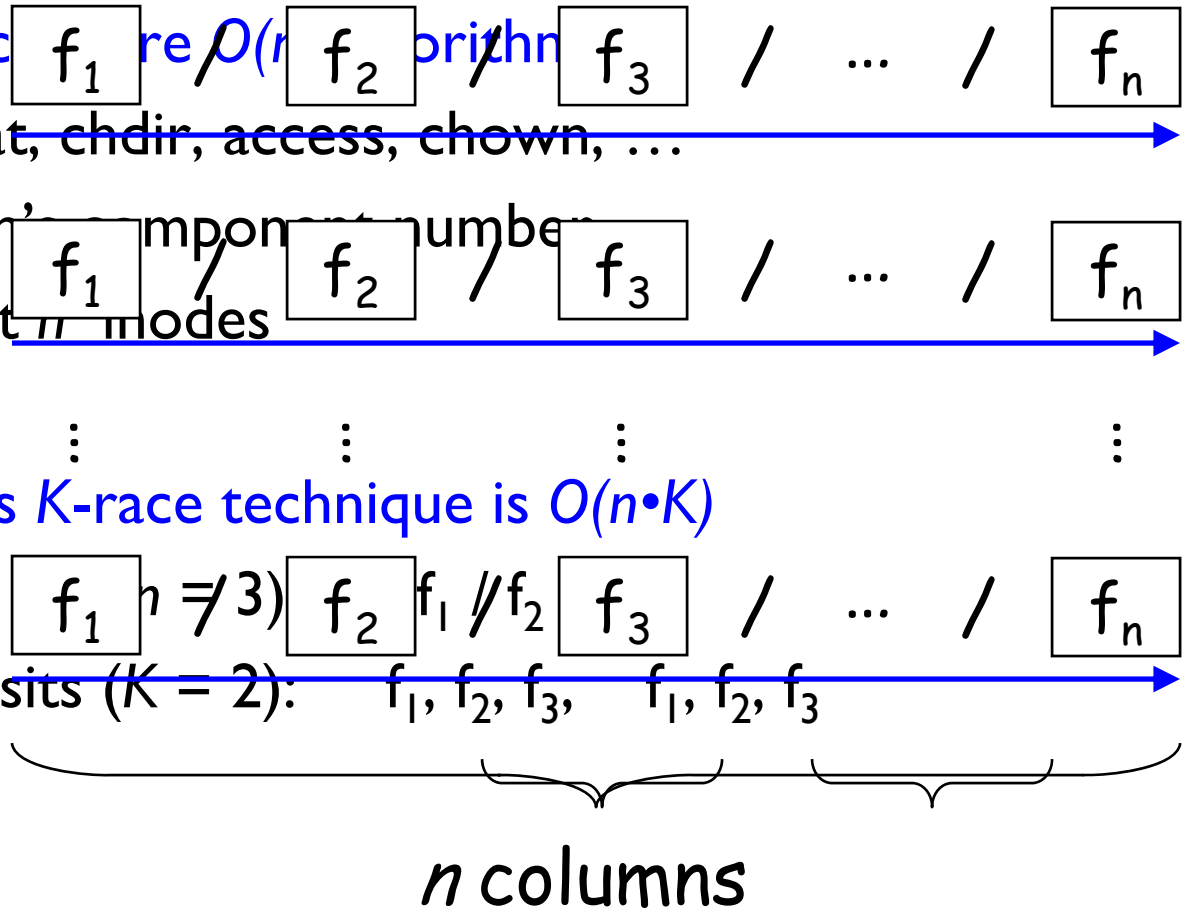
- It was previously believed that

> [Wei and Pu, FAST 2005]:
>
> "*TOCTTOU vulnerabilities are hard to exploit, because they [...] relay on whether the attacking code is executed within the usually narrow window of vulnerability (on the orders of milliseconds).*"

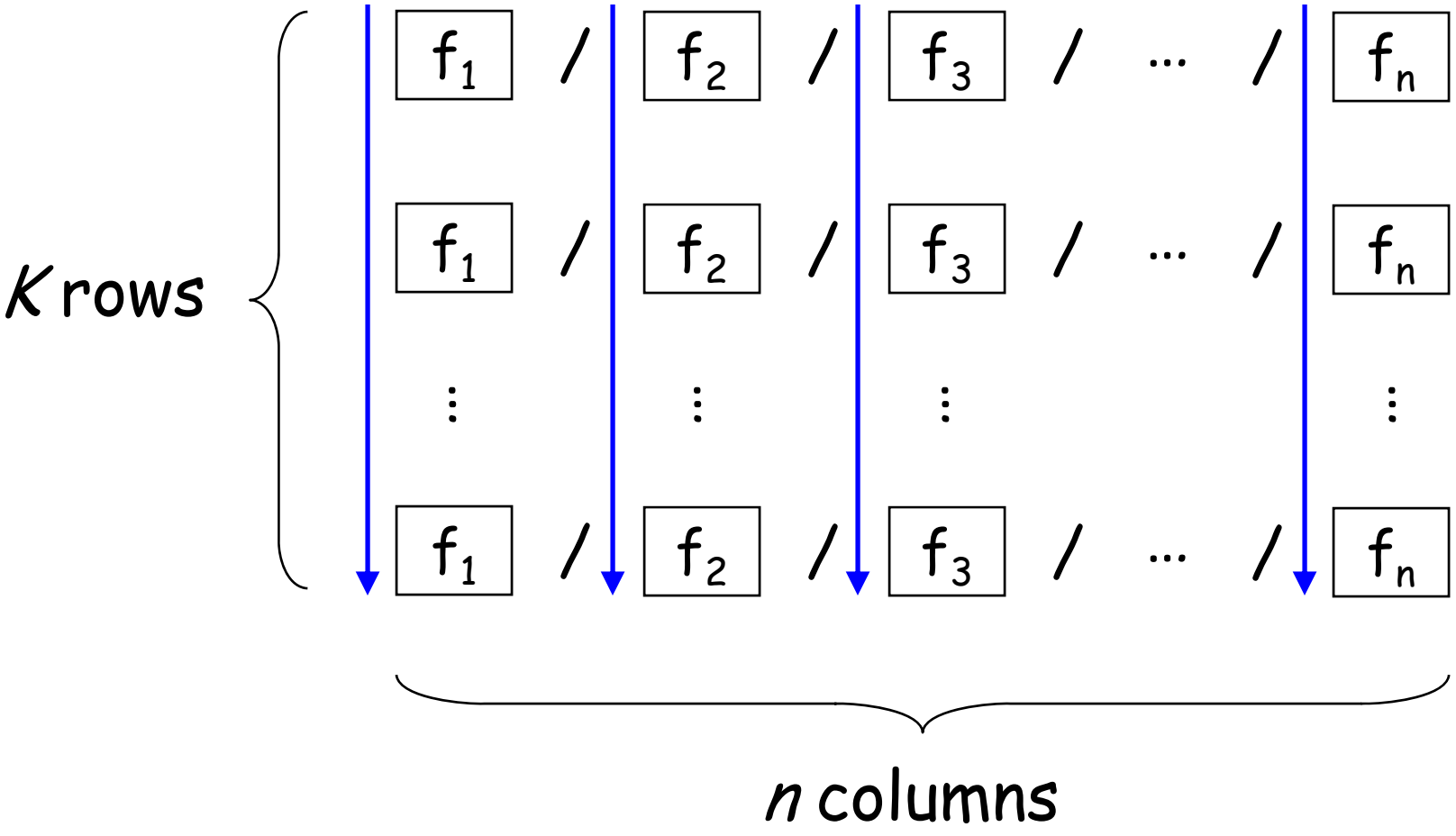
- This is no longer the case…
  - The maze attack always wins ($p \approx 1$)
  - And is generic!

Row-oriented traversal

- Filepath sysc re *O(r* orithm
  - open, stat, chdir, access, chown, …
  - *n* = path's component number
  - Must visit *n* inodes
- Dean & Hu's *K*-race technique is *O(n•K)*
  - n = 3) f1 // f2
  - *K*-race visits (*K* = 2): $f_1, f_2, f_3, f_1, f_2, f_3$



Column-oriented traversal



Column-oriented traversal


```
int access_open( char * fname ) {

    if( fname is absolute )   chdir ( "/" ) +  make relative

    foreach atom in fname do  // atoms of "x/y" are "x" and "y"

        if( is symlink )  SYS(  fd = access_open( atom's target ) )
        else              SYS(  fd = atom_race   ( atom, & s      ) )

        if( not last )    SYS(  fchdir (fd) ;  close (fd)          )
        else              break

    return fd
}
```

How safe is it?

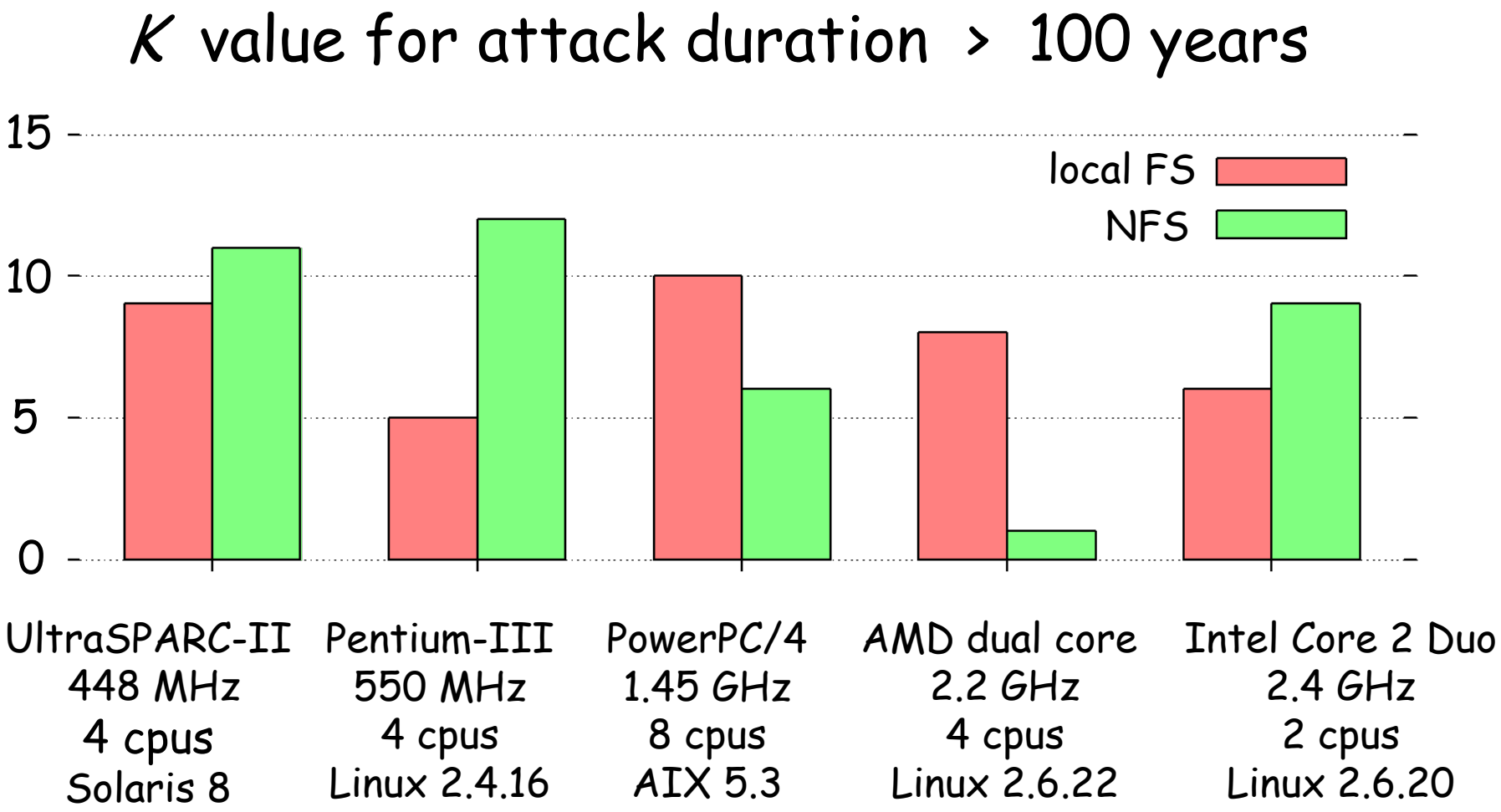
- Obviously, maze attack fails
- But maybe someday somebody will do better?
- We seek a stronger result, with the help of a hypothetical "know all" attack:

Exposed defender: for(i = 1 … 10^6)

```
s = LSTAT   ; lstat      ( f , & s1 )
s = ACCESS  ; access     ( f )
s = OPEN    ; fd = open  ( f )
s = FSTAT   ; fstat      ( fd, & s2)
s = CLOSE   ; close      ( fd )
```

```
if( ! syscalls failed          &&
    ! symlink( s1 )            &&
    s1.inode == s2.inode       &&
    s1.inode == secret_ino )
  losses++
```

How safe is it?



Papers

- Previous slides - conference version
  - "Portably solving file TOCTTOU races with hardness amplification"
  - In *USENIX Conference on File and Storage Technologies (FAST)*
  - Feb 2008
- Following slides - journal version
  - "Portably preventing file race attacks with user-mode path resolution"
  - Submitted (TISSEC)

Must it be probabilistic?


```
int access_open( char * fname ) {

    if( fname is absolute )   chdir ( "/" ) + make relative

    foreach atom in fname do  // atoms of "x/y" are "x" and "y"

        if( is symlink )  SYS(  fd = access_open( atom's target ) )
        else              SYS(  fd = atom_race   ( atom, & s      ) )

        if( not last )    SYS(  fchdir (fd) ; close (fd)           )
        else              break

    return fd
}
```

Must it be probabilistic?

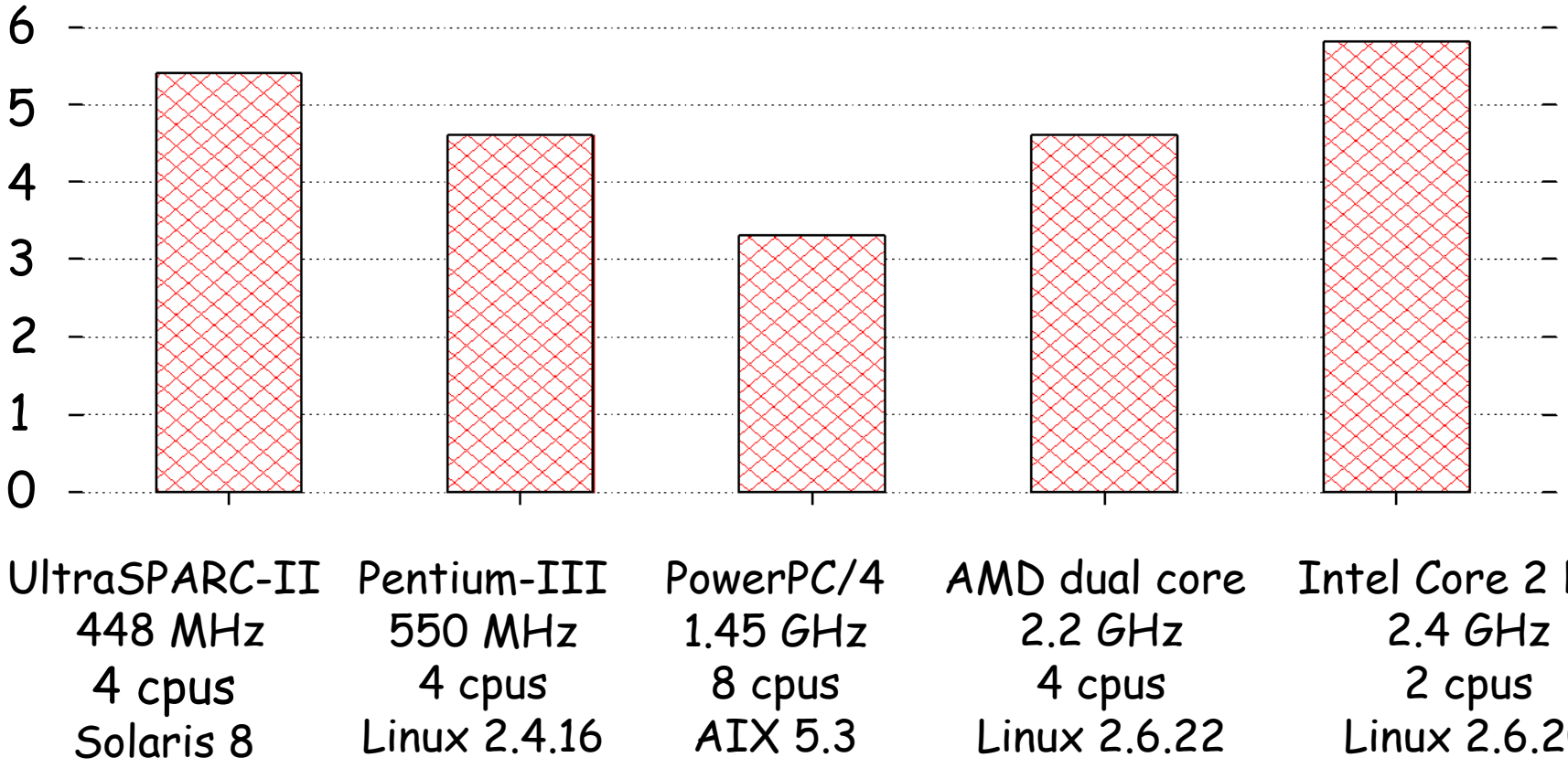

```
struct credentials {
        uid_t     uid;
        gid_t     gid;
        gid_t    *supplementary;
        int       size;   // of supplementary array
};

int access_open( char * fname ) {struct credentials * c

    if( fname is absolute )   chdir ( "/" )  ,  make relative

    foreach atom in fname do  // atoms of "x/y" are "x" and "y"

        if( is symlink )  SYS(  fd = access_open( atom's target ) )
        else              SYS(  fd = atom_open   ( atom, &s, c   ) )
        ...
```

A deterministic solution


```
int atom_open( char * atom, struct stat * s, struct credentials * c )
{
    SYS(  fd = open  (atom )  );   // we did lstat (atom, &s) before
    SYS(       fstat (fd,&s2)  );  // and doing fstat(fd,&s2) after
    CHK(        CMP   (s, &s2)  );  // => it's a hard-link atom

    if       ( c->uid == 0       )   return fd;  // root
    else if ( s->uid == c->uid )   return fd if s->mode permits user;
    else if ( s->gid == c->gid )   return fd if s->mode permits group;
    else if ( s->gid in  c->sup )  return fd if s->mode permits group;
    else                           return fd if s->mode permits others;

    close( fd);
    return -1;
}
```

Overhead

Slowdown relative to naive access/open



Generalizing

- Not just setuid ("check")
  - Credentials structure decouples identity
  - "Deputy" is no longer confused…
  - Exactly same solution to access-open & mail-server
- Not just open ("use")
  - fd/inode mapping is *immutable* => invulnerable
  - Once fd is safely opened, can use *fchown, fchmod, ftruncate, fchdir, fstat, …* (instead of *chown, chmod, truncate, chdir, stat…*)
- Other check / use operations?

Generalizing


```
typedef int (* transaction_t) (char *          atom,
                               struct stat *   s,
                               int             fd);
int check_use (
    char *                fname, //  filepath to check/use
    struct credentials *  c,     //  with these credentials
    int                   flags, //  for when open()ing last atom
    transaction_t         tr);   //  applied to each atom along 'fname'


int collect_garbage (char * atom, struct stat * s, int fd)  {
    if ( S_ISLNK (s)                        )  return -1;
    if ( S_IDDIR (s)                        )  return  0;
    if ( s->atime  >  time(0) - 72*3600 )  return unlink (atom);
    return 0;
}
```

Limitations

- File creation
  - Race typically associated with temp files
- File execution
  - Can't open file "for execution" (only read/write)
  - No standard fexec
- Multithreading
  - Due to fchdir
  - But openat(2) will solve this problem

Conclusions

- POSIX filesystem API is broken
  - Semantics inherently promote TOCTTOU races
- Existing solutions can only locate races
  - But otherwise relate to non-prevalent systems
  - Programmers are on their own
- We propose user-mode path resolution
  - Effectively binds check/use pairs in a generic way
  - Efficiency/safety tradeoff becomes explicit
  - Pairs encapsulated, new programmers educated

Thanks !

BACKUP SLIDES

How safe is it?

- Expected time T_K until K consecutive rounds are lost:

$$\begin{cases} t = \text{avg. time to finish one round} \\ p = \text{probability to lose one round} \\ T_K = t \cdot p^{-K} \end{cases}$$

- Measure t & p under "ideal" attack conditions:
  - SMPs / CMPs only (some older & slower)
  - Multiple attackers, different busy-wait periods
  - Small memory, recursive bg grep-s, huge dir