Multiprocessors May Reduce System Dependability under File-based Race Condition Attacks

Jinpeng Wei and Calton Pu
Georgia Institute of Technology
{weijp,calton}@cc.gatech.edu

Abstract

Attacks exploiting race conditions have been considered rare and "low risk". However, the increasing popularity of multiprocessors has changed this situation: instead of waiting for the victim process to be suspended to carry out an attack, the attacker can now run on a dedicated processor and actively seek attack opportunities. This change from fortuitous encountering to active exploiting may greatly increase the success probability of race condition attacks. This point is exemplified by studying the TOCTTOU (Time-of-Check-to-Time-of-Use) race condition attacks in this paper. We first propose a probabilistic model for predicting TOCTTOU attack success rate on both uniprocessors and multiprocessors. Then we confirm the applicability of this model by carrying out TOCTTOU attacks against two widely used utility programs: vi and gedit. The success probability of attacking vi increases from low single digit percentage on a uniprocessor to almost 100% on a multiprocessor. Similarly, the success rate of attacking gedit jumps from almost zero to 83%. These case studies suggest that our model captures the sharply increased risks, and hence the decreased dependability of our systems, represented by race condition attacks such as TOCTTOU on the next generation multiprocessors.

Keywords: Probabilistic Modeling, Race Condition

1. Introduction

Emerging multiprocessors such as SMP (Symmetric Multiprocessing) with multi-core processors expected to dominate the next generation PC and server markets. These multiprocessors offer significant performance and power consumption advantages, making them potentially more useful for secure systems. For example, additional processors can be dedicated to computationally intensive deep packet inspection in IDS, IPS (Intrusion Detection and Prevention), and anti-virus scanners [11]. However, the use of the additional processing power by attackers to exploit known or new vulnerabilities has received less attention. This paper demonstrates that a concrete class of exploits (file-based race condition called TOCTTOU) will see the success rate of attacks increase sharply from negligible to almost certainty.

TOCTTOU (Time-of-Check-to-Time-of-Use) is a security problem known for more than 30 years [1][2]. An illustrative example is *sendmail*, which used to check for a specific attribute of a mailbox file (e.g., it is not a symbolic link) before appending new messages. However, the checking and appending file system operations are not executed in an atomic transaction. Consequently, if an attacker (the mailbox owner) is able to replace his/her mailbox file with a symbolic link to /etc/passwd between the checking and appending steps by *sendmail*, then *sendmail* may be tricked into appending emails to /etc/passwd (assuming that *sendmail* runs as setuid root). If successful, an attack message containing a syntactically correct /etc/passwd entry would give the attacker root access. TOCTTOU vulnerabilities are widespread and cause serious consequences [24].

The check and use file system calls in the victim process of a TOCTTOU vulnerability are called *TOCTTOU pairs* [18][24]. The time between the two file system calls of a TOCTTOU pair is the *window of vulnerability* (or critical section) of the TOCTTOU vulnerability. To succeed, an attacker process must complete the attack steps within the window of vulnerability of the victim process. The success rate of a TOCTTOU attack thus depends on the scheduling events surrounding and during the window of vulnerability, making it a race condition between the victim and attacker processes. Some attempts have been made to slow down the victim and increase the probability of success, examples include: (1) using slow storage devices (e.g. floppy disks); (2) using

extremely long pathnames (e.g. file system mazes [3]); (3) using large files. This paper studies one method to make the attacker faster and reduce scheduling uncertainty by exploiting additional CPU resources available in multiprocessors.

This paper offers two technical contributions. The first is a probabilistic model for predicting TOCTTOU attack success rate, both for uniprocessors and multiprocessors. By comparing their different capabilities, the model shows that multiprocessors give an attacker more opportunities in winning the race. The second contribution is an experimental study and detailed event analysis of multiprocessor attacks on two recently found TOCTTOU vulnerabilities against popular applications: *vi* and *gedit*. Both attacks have very low success rate on uniprocessors and almost certain success on a multiprocessor (nearly 100% for *vi* and up to 83% for *gedit*). The *gedit* experiments demonstrate that when the vulnerability window is extremely small, the race condition moves to a lower level and the implementation of the attacker program becomes crucial. These analyses give a better understanding of the TOCTTOU attacks on multiprocessors. The main conclusion of the paper is the confirmation of sharply increased risks represented by TOCTTOU attacks.

The rest of this paper is organized as follows. Section 2 briefly introduces the TOCTTOU errors with *vi* and *gedit* which are the target of the attacks discussed in this paper. Section 3 introduces a probabilistic model for TOCTTOU attack success rate. Section 4 summarizes our previous TOCTTOU attack experiments on uniprocessors as a baseline for comparison. Section 5 describes TOCTTOU attacks against *vi* on a SMP. Section 6 discusses TOCTTOU attacks against *gedit* on both a SMP and a multi-core. Section 7 describes an implementation technique that leverages parallelism opportunities provided by multi-cores to significantly speedup the attack program. Section 8 summaries the related work and Section 9 concludes the paper.

2. Background: TOCTTOU Vulnerabilities in Unix-Style File Systems

Recently, several new TOCTTOU vulnerabilities have been found in often-used utility programs such as vi, rpm, emacs and gedit [24]. In this section, we describe the TOCTTOU vulnerabilities with vi and gedit, which are the target of attacks presented in this paper. Each vulnerability is associated with a TOCTTOU pair (e.g., <open, chown>), where the first (check) call is used to establish some invariant about a file object (e.g. the file exists), and the second (use) call is an operation on that same file assuming that the invariant is still valid.

```
while ((fd = mch_open((char *)wfname, …)
……
chown((char*)wfname, st_old.st_uid, st_old.st_gid);
```

Figure 1: *vi* 6.1 vulnerability (fileio.c)

```
1  while (!finish){
2    if (stat(wfname, &stbuf) == 0){
3      if ((stbuf.st_uid == 0) && (stbuf.st_gid == 0))
4        {
5          unlink(wfname);
6          symlink("/etc/passwd", wfname);
7          finish = 1;
8        }
9    }
10 }
```

Figure 2: A program to attack *vi*

2.1. The *vi* Vulnerability and Attack Scheme

The Unix "visual editor" *vi* is a widely used text editor in many UNIX-style environments. For example, Red Hat Linux distribution includes *vi* 6.1. We found that if *vi* is run by root to edit a file owned by a normal user, then the normal user may become the owner of sensitive files such as /etc/passwd. The problem can be summarized as follows. When *vi* saves the file (*wfname*) being edited, it first renames the original file to a backup, then creates a new file under the original name (*wfname* in Figure 1). The new file is closed after all the content in the edit buffer has been written to it. Because this new file is created by root (*vi* runs as root), its initial user is set to root, so *vi* needs to change its owner back to the original user (the normal user). This forms a **<open**, **chown>** window of vulnerability every time *vi* saves the file (Figure 1). During this window, if the normal user (also the attacker) could replace *wfname* with a symbolic link to /etc/passwd, *vi* can be tricked into changing the owner of /etc/passwd to the normal user. A typical attack of this vulnerability is to constantly check the ownership of file *wfname*, and replace *wfname* when its owner becomes root (Figure 2).

2.2. The *gedit* Vulnerability and Attack Scheme

gedit [10] is a text editor for the GNOME desktop environment. We find that *gedit* 2.8.3 (the current distribution in Debian and Redhat Linux) has a

<rename, chown> TOCTTOU vulnerability (See Figure 3). This happens when *gedit* is run by root to edit a file (*real_filename*) owned by a normal user (also the attacker), and *gedit* saves the file. What happens is *gedit* first saves the current buffer content to a temporary scratch file (*temp_filename*), then renames the scratch file to the original file *real_filename* (after backing up the original file properly). Because the scratch file is created by root, the owner of the just saved file (*real_filename*) is root, so *gedit* needs to change its owner back to the original user. This forms a **<rename, chown>** vulnerability window. An attack (Figure 4) against this vulnerability is essentially the same as the attack against *vi* in Section 2.1.

```
if (rename (temp_filename, real_filename) != 0){
… }
chmod (real_filename, st.st_mode);
chown (real_filename, st.st_uid, st.st_gid);
```

Figure 3: gedit 2.8.3 TOCTTOU vulnerability (gedit-document.c)

```
1  while (!finish){
2    if (stat(real_filename, &stbuf) == 0){
3      if ((stbuf.st_uid == 0) && (stbuf.st_gid == 0))
4        {
5            unlink(real_filename);
6            symlink("/etc/passwd", real_filename);
7            finish = 1;
8        }
9    }
10 }
```

Figure 4: gedit attack program version 1

2.3. Discussion

From the description above, we can see that a successful attack against *vi* and *gedit* requires the following preconditions: (1) The attacker has an account on the system. (2) The system administrator edits a file belonging to the attacker. (3) The system administrator makes the mistake of logging in as 'root' instead of the attacker's uid. (4) The attacker makes a reasonable guess about which editor the administrator will use. Such a list of preconditions seems to suggest that a TOCTTOU attack can not easily succeed. However, there are many kinds of TOCTTOU vulnerabilities (e.g., 224 for Linux), and depending on how the victim program is implemented, some TOCTTOU vulnerabilities are much easier to attack than those discussed here [15]. Interested readers are referred to [18] and [24] for more information. The point of this paper is that once these preconditions are satisfied, the attacker can succeed much easier on a multiprocessor than on a uniprocessor.

3. A Probabilistic Model for Predicting TOCTTOU Attack Success Rate

3.1. The Basic General Model

A TOCTTOU attack succeeds when the attacker is able to modify the mapping from file name to disk block within the vulnerability window. In order to succeed, the attacker must first find the vulnerability window, and then change the file mapping. Therefore, our model divides the attacker program into two parts: (1) a detection part that finds the beginning of the vulnerability window, and (2) an attack part that modifies the file mapping.

One of the critical issues is whether the victim is suspended within the vulnerability window, since the suspension increases substantially the success rate. Based on the law of total probability, the attack success rate:

P(attack succeeds) = P(victim suspended) * P(attack succeeds | victim suspended) + P(victim not suspended) * P(attack succeeds | victim not suspended)

In order for the attack to succeed, the attacker program must be scheduled within the vulnerability window and the attack must finish within the vulnerability window, so

P(attack succeeds | victim suspended) = P(attack scheduled ● attack finished | victim suspended)
= P(attack scheduled | victim suspended) * P(attack finished | victim suspended)

We can derive P(attack succeeds | victim not suspended) in a similar way and get the refined probability in Equation 1.

In Equation 1, all the events are under the context of the victim vulnerability window. e.g. 'attack finished' means 'attack finished within the vulnerability window'.

P(attack succeeds) = P(victim suspended) * P(attack scheduled | victim suspended) * P(attack finished | victim suspended)
+ P(victim not suspended) * P(attack scheduled | victim not suspended) * P(attack finished | victim not suspended)

Equation 1: The probability of a successful TOCTTOU attack

3.2. Attack Success Rate on a Uniprocessor

On a uniprocessor, P(attack scheduled | victim not suspended) = 0 since it is impossible to schedule the attacker when the victim is running. Therefore on a

uniprocessor the second part of Equation 1 contributes nothing to the success rate. E.g., P(attack succeeds) = P(victim suspended) * P(attack scheduled | victim suspended) * P(attack finished | victim suspended).

Several observations can be made about P(attack succeeds) on a uniprocessor:

- P(attack succeeds) ≤ P(victim suspended). The probability that the victim is suspended within its vulnerability window gives an upper bound for the attack success rate. If the victim is always suspended (e.g. *rpm* in [24]), the attacker can achieve a success rate as high as 100%. In contrast, if the victim is rarely suspended (e.g. *gedit* in Section 2.2), the attack success rate can be near zero.
- P(attack scheduled | victim suspended) is the probability that the attacker process gets scheduled when the victim relinquishes CPU. This value depends on several factors such as the readiness of the attacker, the system load (if round-robin scheduling is used), or the priority of the attacker (if priority-based scheduling is used). Typically in a lightly loaded environment this value can be nearly 100% if the attacker program uses an infinite loop actively looking for the exploit opportunity.
- P(attack finished | victim suspended) is the probability that the attacker successfully modifies the file mapping while the victim is suspended. Since there is only one CPU, as long as the attack part is not interrupted, this probability can be 100%. Typically this is the case because modifying the file mapping requires very short processing time and needs not block on I/O.

Based on the above analysis, the attack success rate is mainly determined by P(victim suspended) on a uniprocessor system, and the implementation of the attack part is relatively less critical.

3.3. Attack Success Rate on Multiprocessors

On multiprocessors, the attacker can run on a different processor than the victim when the victim is running within its vulnerability window. This makes the second part of Equation 1 non-zero, i.e., P(attack scheduled | victim not suspended) > 0. This fact increases the success rate of TOCTTOU attacks on multiprocessors as compared to uniprocessors. If P(victim suspended) is relatively large, then the success rate on multiprocessors may not increase significantly. However, if P(victim suspended) is very small (approaching 0), then P(victim not suspended) approaches 1, and the gain due to the second part of P(attack succeeds) may become very significant.

Therefore for an attacker, the benefit of having multiprocessors is maximized when the victim is rarely suspended in the vulnerability window. An analysis of the second part of Equation 1 shows that:

- P(attack scheduled | victim not suspended) is similar to P(attack scheduled | victim suspended) discussed in Section 3.2. The conclusion is that it can be as high as 100%.
- P(attack finished | victim not suspended) is the probability that the attack is finished within the vulnerability window. Since the victim is running concurrently with the attacker, the result of the attack depends on the relative speed of the attacker and the victim, a more detailed analysis is needed (next Section).

3.4. Probabilistic Analysis of P(attack finished | victim not suspended)

In order to predict P(attack finished | victim not suspended) in more detail, we analyze the race condition at different levels: the first level is CPU, which is the main contention in uniprocessor attacks; the next level is file object, because the file system already has a synchronization mechanism to regulate shared accesses. In Unix-style file systems, the modifications to an inode are synchronized by a semaphore. Since the operations of the victim and the attacker on the shared file modify the same inode, they both need to acquire the same semaphore. In this case, the race is reduced to the competition for the semaphore and we can model the success rate of the attack in the following way.

In this model, we assume that the attacker runs in a tight loop (the detection part), waiting for the vulnerability window of the victim to appear. Let D be the time consumed by each iteration of detection part, and let t_1 be the earliest start time for a successful detection and t_2 be the latest start time for a successful detection followed by a successful attack (e.g. the attacker acquires the semaphore first). t_1 and t_2 are determined by the victim process. Some observations can be made as follow (Figure 5):

A successful attack starts with a successful detection as its precondition. This successful detection may start as early as t_1 (Figure 5, case (a)), and as late as $t_1 + D$ (Figure 5, case (f)). Then the interval $[t_1, t_1 + D)$ is our sample space. Out of this interval $[t_1, t_1 + D)$, if the detection is started before t_2, the attack succeeds

(Figure 5, cases (a) through (c)); otherwise the attack fails (Figure 5, cases (d) through (f), because the attack is launched too late). Let's assume a uniform distribution for the start time of the detection part, the success rate is thus $\frac{t_2 - t_1}{D}$.

In Figure 5 we assume that $t_2 \in [t_1, t_1 + D)$. Two other cases are:

- If $t_2 < t_1$, then the success rate is 0;
- If $t_2 \geq t_1 + D$, then the success rate is 1.

Let $L = t_2 - t_1$, and we get:

$$\text{The success rate} = \begin{cases} 0, if\ (L < 0) \\ L/D, if\ (0 \leq L < D) \\ 1, if\ (L \geq D) \end{cases} \quad (1)$$

In formula (1), L measures the laxity of the successful attacks, which is a characterization of the victim: the larger L, the more vulnerable the victim. D is a characterization of the detection part of the attacker: the smaller D, the faster the attacker, and the higher success rate. So L/D gives a very useful measurement of the relative speed of the victim and the attacker.

It should be noted that L and D in formula (1) are not strictly constant, because the executions of the victim as well as the attacker are interleaved with other events (e.g. kernel timers) in the system. That is, the running environment imposes variance on these parameters. So formula (1) only offers a statistical guidance about the attack success rate.

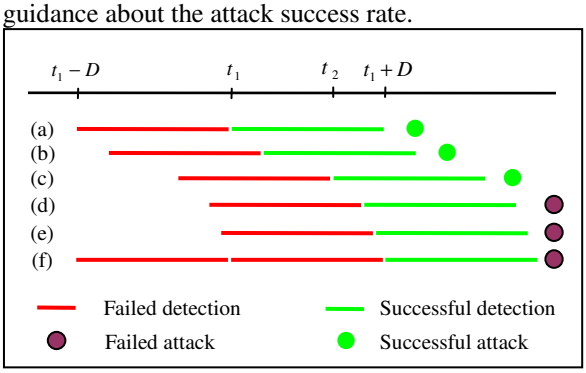

Figure 5: Different attack scheduling on a multiprocessor

4. Baseline Measurements of TOCTTOU Attacks on Uniprocessors

For comparison purposes, in this section we summarize the measured success rates of *vi* and *gedit* TOCTTOU attacks on uniprocessors from [24].

4.1. *vi* Attack Experiments on Uniprocessors

Since the *vi* vulnerability window includes the writing of a whole file, the size of the window naturally depends on the file size. The measured success rates for file sizes ranging from 20KB to 10MB are the following:

- When the file size is small (from 100KB to 1MB), there is a rough correlation between attack success rate and file size, as shown in Figure 6. However, the correlation disappears for larger file sizes (e.g., between 2MB to 3MB), showing that file size alone does not determine the success rate completely.
- Besides file size, we studied other factors (e.g., I/O operation, CPU slicing, and preemption by higher priority kernel threads) that corroborate the non-deterministic nature of TOCTTOU attacks on a uniprocessor [24].

From Figure 6 we can see that for normal file sizes (Using *vi* to edit a 2MB text file is considered rare in real life), the success rate can be as low as 1.5% and as high as 18%. Furthermore, when the file size approaches 0, the success rate also approaches 0.

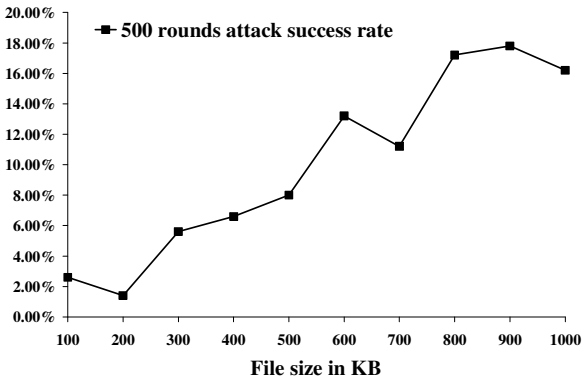

Figure 6: Success rate of attacking vi (small files) on a uniprocessor

4.2. *gedit* Attack Experiment on Uniprocessors

The experiments in which a TOCTTOU attack was carried out against the *gedit* vulnerability saw no successes. This is because the *gedit* vulnerability window (Figure 3) does not include the writing of the new file as in *vi*, so it is much shorter and bears no relationship to the file size. These factors reduced the success rate for *gedit* attacks to essentially zero on a uniprocessor.

5. vi Attack Experiments on SMP

We repeated the *vi* attack experiments described in Section 4.1 on a SMP machine (2 Intel Xeon 1.7GHz CPUs, 512MB main memory, and 18.2GB SCSI disk with ext3 file system).

First we tried different file sizes ranging from 20KB to 1MB with a stepping size of 20KB, and observed the success rate of 100% for all file sizes. This confirms the probabilistic predictions in Section 3.3 and shows that a multiprocessor greatly increases the attacker's chance of success compared to a uniprocessor (Figure 6 in Section 4.1). We did a detailed event analysis to confirm the attacker and victim processes ran on separate CPUs during the vulnerability window. We also eliminated the possibility that the attack success is due to the victim being blocked on I/O operations (which would have made the attack easier). Consequently, we conclude that the attack success is due to the length of *vi* vulnerability window being much larger than the time it takes the attacker to finish the attack steps (file name redirection).

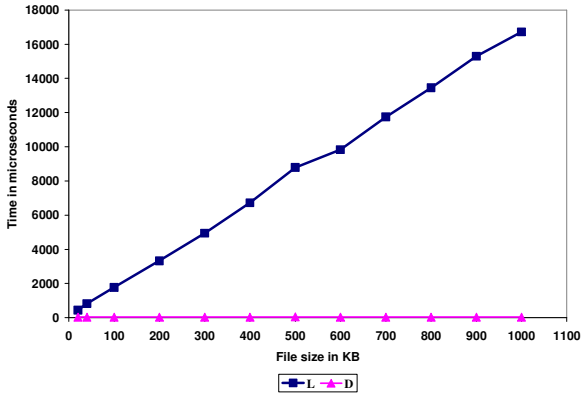

Figure 7: The L and D values for vi SMP attack experiments

Figure 7 shows the L and D values (Section 3.4) for the *vi* attack experiments that we conducted on the SMP. We can see that L >> D when the file is large (e.g.1MB); and the difference (L – D) decreases as the file size decreases. But (L – D) is always positive, even when the file size becomes very small. Therefore we can say with almost certainty that for *vi* attack experiments, L > D. By formula (1) we know that the success rate of *vi* attacks is almost 100% all the time.

One thing to notice from Figure 7 is that as the file size approaches 0, the difference (L – D) also approaches 0. Is it possible that L becomes smaller than D? Then according to formula (1) the attack success rate will be smaller than 100%.

To see this we run the experiment again with the smallest files (only 1 byte each). And the success rate we get is around 96%. Again we did a detailed event analysis of this experiment. We measure the average L and D values and put them in Table 1. We can see that although L > D in these attacks, they have become very close. If we consider the fact that the values for L and D are not strictly constant due to the environmental influence, we realize that whether L > D all the time becomes questionable when they are close enough (When L >> D the inaccuracy introduced by the environment does not change the relationship). This helps to explain why the success rate can not be 100% when the file contains only 1 byte.

Another point is that so far we actually treat P(attack finished | victim not suspended) in Section 3.4 as the sole basis for predicting the success rate, which is not always accurate (Equation 1). The justification is that when the *vi* vulnerability window is large enough, the effect of other factors in Equation 1 is negligible. For example, P(attack scheduled | victim not suspended) < 100% in general which means that the attacker may not be scheduled during sometime in the vulnerability window. However, if the vulnerability window is very large, the attacker is still within it when he/she is scheduled eventually. That is, the temporary suspension does not affect the result of the attack. However, when the vulnerability window becomes small enough (e.g. L and D become close enough), the suspension may cause the attacker to miss the vulnerability window. In such a case the attack fails, thus the suspension changes the attack result.

In several of the failed 1-byte *vi* experiments, we find that some other processes prevents the attacker from being scheduled on another CPU during the *vi* vulnerability window.

This analysis tells us that although using a multiprocessor can greatly increase the attack's chance of success, the success is still not guaranteed: the attack is still influenced by other environmental factors such as kernel activities and system load. However, 96% is more than enough for an attacker.

Table 1: The average L and D values (in microseconds) for *vi* SMP attack experiments (file size = 1 byte)

	Average	Stdev
L	61.6	3.78
D	41.1	2.73

6. *gedit* Attack Experiments on Multiprocessors

6.1. *gedit* SMP Attack Event Analysis

As mentioned in Section 4.2, our attack experiments against *gedit* on uniprocessors saw no successes. However, when we try this attack on a SMP (the same machine as in Section 5), we get roughly 83%, a surprisingly high success rate. A detailed event analysis is thus conducted to understand this result.

For the *gedit* attack, we have observed that if the attacker's **unlink** is invoked before *gedit*'s **chmod** (Figure 3 and Figure 4), then attack succeeds. This is because these two system calls compete for the same semaphore, so if **unlink** wins, **chmod** as well as the following **chown** will be delayed. As a result the attacker's **unlink** and **symlink** can have enough time to finish before *gedit*'s **chown**. On the other hand, if **unlink** loses, **unlink** and the following **symlink** of the attacker will be delayed, so the attack will fail. So there is an interesting cascading effect in *gedit* attack experiment. Therefore, for *gedit* attacks, t_1 is somewhere within the execution of **rename** (the attacker does not need to wait until the end of **rename** to see that *real_filename* has been created), D is the interval between the start of **stat** and the start of **unlink**. Let t_3 be the start of **chmod**, then $t_2 = t_3 - D$, and $L = t_2 - t_1 = t_3 - D - t_1$. We experimentally get the L and D values as in Table 2.

Table 2: L and D values for *gedit* attacks on a SMP (in microseconds)

	Average	Stdev
L	11.6	3.89
D	32.7	2.83

The calculation of L here is not accurate because the estimation of t_1 is not accurate. Currently t_1 is established as the earliest observed start time of **stat** which indicates a vulnerability window. So it may not be optimal. An earlier (thus smaller) t_1 will result in a larger L. So the success rate indicated by Table 2 (35%) may be overly conservative compared to the observed success rate.

An important contributing factor to L is the computation time between the end of **rename** and the start of **chmod**. The average length of this computation is 43 microseconds. As we will see in Section 6.2, this factor is very important for the high success rate of *gedit* attack on the SMP.

There is another contributing factor. Usually when *gedit*'s **chmod** is blocked, the Linux kernel will try to schedule something else to run (e.g. internal kernel events such as soft IRQs, kernel timers and tasklets), which further lengthens **gedit** vulnerability window (but this contributes just a little to the delay compared with that due to the semaphore).

6.2. *gedit* Multicore Attack Experiment

6.2.1. Attack one

We repeat the *gedit* attack (Figure 4) on a multi-core (Dell Precision 380 with 2 Intel Pentium D 3.2 GHz dual-core and Hyper-Threading CPUs, 4GB main memory, and 80GB SCSI disk with ext3 file system). We get very different result: now we see almost no success in the same attack experiment. The main change in the situation is that the victim spends much less time between **rename** and **chmod** (3 microseconds vs. 43 microseconds), so **chmod** happens before **unlink** of the attacker, but in the SMP experiment (Section 6.1) situation is the opposite.

Figure 8 shows the important system events during one failed attack on the multi-core. The upper bar corresponds to the execution of *gedit* (**rename**, **chmod**, **chown**) and the lower bar corresponds to that of the attacker (**stat**, **unlink**, **symlink**). Notice that the gap (the computation) between **rename** and **chmod** of *gedit* is only 3 microseconds, but the gap between **stat** and **unlink** of the attacker is 17 microseconds. It is because of this relatively larger gap that the attacker's **unlink** is called later than the victim's **chmod**. Actually we can see that **unlink** is called later than **chown** and as a result **unlink** has to wait on the semaphore during its execution. The 17 microsecond gap of the attacker includes 11 microseconds of computation and 6 microseconds of system trap processing (page fault). Speaking in terms of D, these 17 microseconds are counted so D is around 22. On the other hand L is around $3 - D = -19$, so according to formula (1) the attack success rate is probably 0. Putting this in another way, the victim is now much faster than the attacker, so it is very difficult for the attacker to win the race.

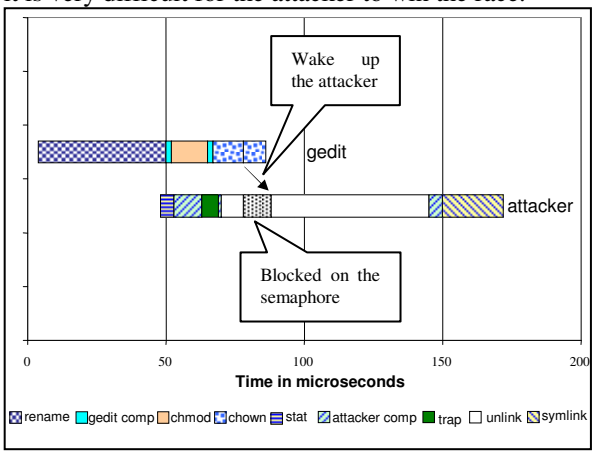

Figure 8: Failed *gedit* attack (program 1) on a multi-core

6.2.2. Attack Two

We think that the 17 microsecond gap in Figure 8 is mainly responsible for the low success rate. If we could reduce the length of this gap then the situation may change. A source code analysis tells us that before the vulnerability window the true branch of statement 3 in Figure 4 (statements 5 to 7) is never taken. Once the vulnerability window starts, the true branch of statement 3 is taken, and then statement 5 (**unlink**) is about to be executed. Right at this point the attacker program encounters a trap (page fault). We figure out that this effect is due to the memory management for shared libraries in Linux. Specifically, in Linux all system calls are through *libc*, which is a dynamic library shared among user-level applications. To save physical memory, Linux kernel keeps only one copy of *libc* in physical memory, and its virtual memory mechanism maps the pages of this copy to the address space of an application on demand. For example, the physical page containing the wrapper for **unlink** is mapped into an application's address space when this application first invokes **unlink**. This mapping is preceded by a trap (page fault) and the corresponding handler routine carries out the mapping. This is exactly what happens in Figure 4, where **unlink** is first invoked when the true branch of statement 3 is taken. As a consequence, if we intentionally invoke **unlink** (and **symlink** although it seems to be on the same page as **unlink**) before the true branch of statement 3 is taken, we may remove the trap (page fault).

```
1   while (!finish){  /* argv[1] holds real_filename */
2     if (stat(argv[1], &stbuf) == 0){
3       if ((stbuf.st_uid == 0) && (stbuf.st_gid == 0))
4         {
5           fname = argv[1];
6           finish = 1;
7         }
8       else
9           fname = dummy;
10
11      unlink(fname);
12      symlink("/etc/passwd", fname);
13    }//if stat(argv[1] ..
14  }//while
```

Figure 9: *gedit* attack program version 2

So we re-implement the attacker program as shown in Figure 9. Now **unlink** and **symlink** are called no matter the vulnerability window appears or not. The only trick is to switch in the correct file name when it does appear.

Then we perform the *gedit* attack experiment again using the program in Figure 9. And we begin to see many successes!

We plot the important system events during one successful *gedit* attack in Figure 10, similar to Figure 8. We can see that now the gap between **stat** and **unlink** of the attacker has decreased to 2 microseconds: the trap has disappeared. On the other hand, the gap between **rename** and **chmod** of *gedit* is 2 microseconds. So the attacker has a very narrow chance of winning the race. In this particular case, the attacker wins because his/her **stat** starts well before the end of **rename**, so he/she identifies the vulnerability window at the first moment, and invokes **unlink** ahead of **chmod**. Has the attacker been 2 microseconds later, the attack would fail.

Notice that during this attack the running time of **stat** has been lengthened to 26 microseconds (typically it needs 4 microseconds), probably due to some other more complicated race condition (For example the contention for directory entries along the path name). We are not quite clear about the reason but this does not change the applicability of formula (1) because now we have a much earlier t_1 (27 microseconds into **rename**), which makes a L value of at least 1 microseconds.

This experience tells us that on multiprocessors the implementation of the attacker program can be very critical in determining the attack success rate, especially when the vulnerability window is very narrow.

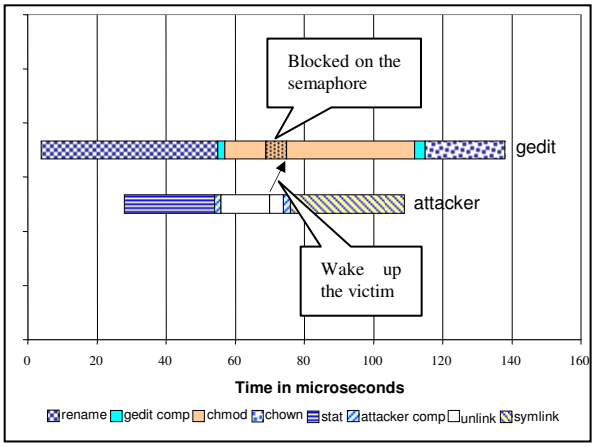

Figure 10: Successful *gedit* attack (program 2) on a multi-core

7. Pipelining Attacker Program

The multi-core *gedit* experiment highlights the importance of the implementation of the attacker program. Concretely, we found that among the three

steps of the attack (**stat**, **unlink**, **symlink**), **unlink** is the most time-consuming. A closer look into the file system source code shows that actually **symlink** needs not wait on the completion of **unlink**. Instead **symlink** can begin once the inode has been detached from the directory by **unlink**, which happens relatively early. (The main part of **unlink** is spent physically truncating the file.) This observation shows that on a multiprocessor, the attacker can distribute its attack steps to multiple CPUs to speed up the attack part and increase its success rate.

To confirm this hypothesis, we implemented a multithreaded *gedit* attack program with two threads: the first thread carries out the **stat**, **unlink** steps and the second thread carries out the **symlink** step asynchronously. Figure 11 shows the effect of parallelizing the attack program for three different file sizes. For each file size (e.g. 500KB), there are three bars: the first two bars correspond to the execution of the two threads in a parallelized attack program, and the third bar corresponds to the execution of the normal sequential attack program. In the parallelized attack, **symlink** can finish (and so does the attack) well before the end of **unlink.** This is in contrast to the sequential attack, where **symlink** has to wait until **unlink** finishes. The comparison between the end times of **symlink** shows that leveraging on the parallelism provided by a multiprocessor can greatly reduce the amount of time needed for a successful attack. This is especially important when the vulnerability window is very narrow so the attacker needs to be very fast. This experiment shows one feasible way of doing it.

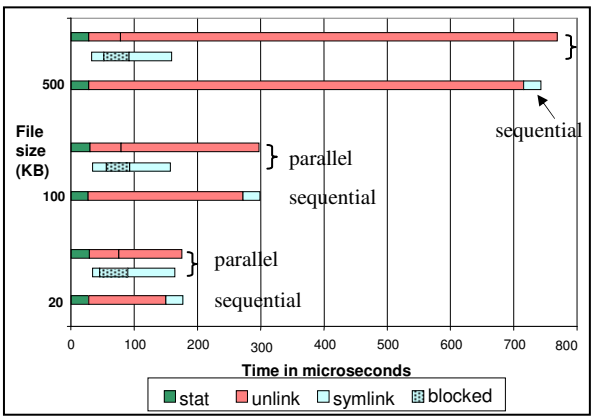

Figure 11: The effect of parallelizing the attack program

8. Related Work

TOCTTOU is one example of race condition problem. In general, every shared resource has the potential for such problems [23]. Percival [17] shows that shared access to memory caches in Hyper-Threading technology allows a malicious thread to steal RSA keys. Similar attacks have also been reported on AES [16]. While carrying out such attacks do not rely on multiprocessors, it would be interesting to see if they become easier on multiprocessors.

Timing attacks have long been used to infer secret keys in cryptosystems [4][13][21]. This kind of attacks share a common attribute with TOCTTOU attacks - both try to infer something about the victim. The difference between them is that the former only read (steal) information from the victim to violate its *confidentiality* but the latter modify the information used by the victim to violate its *integrity*.

TOCTTOU vulnerabilities can be detected in two ways: static analysis or dynamic analysis. The first approach analyzes the application source code to find TOCTTOU pairs. One such tool is MOPS [5] which uses model checking and is able to find 41 TOCTTOU bugs in an entire Linux distribution [20]. Other potentially useful techniques include compiler extensions [8][9]. The main difficulty with these static tools is high false positive rate. The second approach to detect TOCTTOU vulnerabilities is dynamic monitoring and analysis. These tools can be further classified into dynamic online detection tools such as [14] and [19] and post mortem analysis tools such as [12] and [24]. Compared to static analysis, dynamic analysis has lower false positive rate, but it suffers from false negatives because the search space is incomplete.

The high success rate of exploiting TOCTTOU vulnerabilities calls for effective defense against such attacks. Various technical remedies have been suggested, including setting proper file/directory permissions, randomizing file names, replacing mktemp() with mkstemp(), and using a strict umask to protect temporary directories. However, none of these fixes can be considered a comprehensive solution for TOCTTOU vulnerabilities.

There have been specialized mechanisms such as RaceGuard [6] and a probabilistic approach [7] which protect particular TOCTTOU pairs. Pseudo-transaction [22] is a more generic mechanism to protect some classes of TOCTTOU vulnerabilities. We have proposed a complete defense against TOCTTOU attacks called EDGI (Event Driven Guarding if Invariants) [18]. The details of EDGI are out of the scope of this paper.

9. Conclusion

TOCTTOU (Time-of-Check-to-Time-of-Use) is a file-based race condition that can cause serious

consequences. However, traditionally TOCTTOU vulnerabilities have been considered "low risk" because the success rate of exploits appears to be low and results non-deterministic. This paper shows that in multiprocessor environments, the uncertainty due to scheduling is greatly reduced for an attacker sitting on a dedicated CPU; as a result some TOCTTOU attacks can have very high success rates. Thus TOCTTOU attacks on multiprocessors are practical security threats.

The first contribution of this paper is a probabilistic model for TOCTTOU attack success rate. It predicts the probability of success of a TOCTTOU attack. It provides a basic guideline for modeling TOCTTOU attacks and performing experiments, showing higher success rates on a multiprocessor compared to a uniprocessor. This model can be applied to many race condition attacks, not just TOCTTOU.

The second contribution of this paper is a set of attack experiments against two concrete and well known applications: *vi* and *gedit*. The *vi* experiments show that even for the smallest files involved in the vulnerability window, the attacker can achieve nearly 100% success rate on a multiprocessor, compared to low single digit percentages on uniprocessors. The *gedit* experiments demonstrate that when the vulnerability window is extremely small, the race moves to a lower level and the implementation of the attacker program becomes very important. The *gedit* experiments show a success rate of up to 83% compared to essentially zero on uniprocessors. These experiments corroborate our probabilistic model.

Our main conclusion is that an attacker can exploit the parallelism provided by multiprocessors to achieve more effective and more efficient attacks. More generally, our model and experiments show that multiprocessors can potentially reduce overall system dependability, so we should re-evaluate the risks of known vulnerabilities and effectiveness of security mechanisms in multiprocessor environments.

10. Acknowledgement

This work was partially supported by NSF/CISE IIS and CNS divisions through grants CCR-0121643, IDM-0242397 and ITR-0219902. We also thank the anonymous reviewers for their insightful comments.

11. References

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

[2] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. Computing Systems, 9(2):131–152, Spring 1996.

[3] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing Races for Fun and Profit: How to Abuse atime. USENIX Security Symposium, 2005.

[4] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. USENIX Security Symposium, 2003.

[5] Hao Chen, David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), November 2002.

[6] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. USENIX Security Symposium, 2001.

[7] Drew Dean and Alan J. Hu. Fixing Races for Fun and Profit: How to use access(2). USENIX Security Symposium, 2004.

[8] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. Operating Systems Design and Implementation (OSDI), 2000.

[9] Dawson Engler, Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. ACM Symposium on Operating Systems Principles, 2003.

[10] http://www.gnome.org/projects/gedit/

[11] Amer Haider. Multi-Core Microprocessor Architecture for Network Services and Applications. http://www.commsdesign.com/design_corner/showArticle.jhtml?articleID=57703590

[12] Calvin Ko, George Fink, Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. Proceedings of the 10th Annual Computer Security Applications Conference, page 134-144.

[13] P. Kocher. Cryptanalysis of Diffie-Hellman, RSA, DSS, and other cryptosystems using timing attacks. In Advances in cryptology, CRYPTO'95, pages 171–183, 1995.

[14] K. Lhee and S. J. Chapin, Detection of File-Based Race Conditions, Intl. Journal of Information Security, 2005.

[15] http://xforce.iss.net/xforce/xfdb/8652

[16] Dag Arne Osvik, Adi Shamir, Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. Proceedings of RSA Conference 2006, Cryptographer's Track (CT-RSA).

[17] Colin Percival. Cache Missing for Fun and Profit. BSDCan 2005.

[18] Calton Pu, Jinpeng Wei. A Methodical Defense against TOCTTOU Attacks: The EDGI Approach. International Symposium on Secure Software Engineering (ISSSE '06).

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[20] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model Checking An Entire Linux Distribution for Security Violations. Annual Computer Security Applications Conference, December 6, 2005.

[21] Dawn Song, David Wagner, Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. USENIX Security Symposium, 2001.
[22] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. USENIX Security Symposium, 2003.
[23] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. Proceedings of the IEEE, 63(9): 1278-1308, September 1975.
[24] Jinpeng Wei, Calton Pu. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05), San Francisco, CA, December 2005.