Concurrency Attacks

Junfeng Yang, Ang Cui, Sal Stolfo, Simha Sethumadhavan
{*junfeng, ang, sal, simha*}*cs.columbia.edu*
Department of Computer Science
Columbia University

Abstract

Just as errors in sequential programs can lead to security exploits, errors in concurrent programs can lead to *concurrency attacks*. Questions such as whether these attacks are real and what characteristics they have remain largely unknown. In this paper, we present a preliminary study of concurrency attacks and the security implications of real concurrency errors. Our study yields several interesting findings. For instance, we observe that the exploitability of a concurrency error depends on the duration of the timing window within which the error may occur. We further observe that attackers can increase this window through carefully crafted inputs. We also find that four out of five commonly used sequential defense mechanisms become unsafe when applied to concurrent programs. Based on our findings, we propose new defense directions and fixes to existing defenses.

1 Introduction

Concurrent programs have become pervasive and critical because of the move to multicore hardware and deployment of large-scale distributed systems. Yet these programs remain much more difficult to write, test, and debug than sequential versions [23, 31]. This impediment has led to numerous, subtle and serious errors in concurrent programs [25]. Just as errors in sequential programs can lead to security exploits, concurrency errors may be vulnerable and lead to *concurrency attacks* which allow attackers to steal information, escalate privileges, inject code, *etc*. To defend against these attacks, we need to better understand concurrency errors and how they can or have been exploited. Prior work [25] has studied properties of many concurrency errors, but it focused on general concurrency errors, not the *exploitable* ones, which our study reveals to have different properties.

In this paper, we present a study of concurrency attacks and the security implications of concurrency errors. We focus on four questions:

§ Do concurrency attacks occur in the wild? In theory any bug—concurrent or sequential—may be exploited to compromise security but we want to know if real-world concurrency bugs have been exploited in practice. If concurrency attacks are not practicable we need not worry about them. (Section 2).

§ What factors make concurrency errors easy to exploit? If attackers have to jump through hoops to exploit any concurrency error, they will likely go after other low-hanging errors. (Section 3).

§ Can we leverage these factors to improve the effectiveness of existing concurrency error detection techniques? (Section 4).

§ How do concurrency attacks weaken existing defenses such as taint tracking and intrusion detection and how can we fix them? (Section 5).

Our study yields several interesting findings. We find that concurrency attacks are indeed real as we find plenty of exploitable concurrency errors in the CVE database [3]. Interestingly, few of these errors have appeared in prior studies or race detection literature, suggesting that the research communities may not be aware their existence or impact. We observe that how exploitable a concurrency error is highly depends on the size of the timing window within which the error may occur which we call the *vulnerability window*. Further, an attacker can expand this window through carefully crafted inputs. Our study shows that many common mechanisms in existing defenses will not work against concurrency attacks. We propose fixes to some of these weaknesses and also propose new defense directions.

We hope that our initial work on concurrency attacks will further stress the importance of ongoing work on better programming languages and specifications [12, 20], runtime systems [9, 10, 16, 17, 24], and tools [18, 22, 26, 36, 38] for concurrent programs. In addition, we hope that it will raise awareness of concurrency attacks and motivate fellow researchers to work on preventing them. The raw data including URLs to the concurrency errors and sometimes their exploits and our detailed categorization of the errors studied are available at `http://systems.cs.columbia.edu/projects/concurrency/`.

2 Concurrency Attacks Are Real

To construct concurrency attacks, we initially tried exploiting concurrency errors in existing benchmarks [21, 25]. Unfortunately, quite a few of these errors cannot be triggered without manually injecting `sleep()` calls. Moreover, many, such as the errors in SPLASH2, are practically harmless from a security perspective. The worst ones tend to cause only program crashes, not the security exploits we want.

We then turned to the race section of the CVE database [3], which fortunately lists many exploitable concurrency errors besides the familiar file system Time-of-Check-to-Time-of-Use (TOCTOU) races [27, 32, 33, 35]. We also examined the bug databases of popular software. From these sources, we collected concurrency errors that are exploitable and have detailed description, such as a well-written report of the error, sample exploit code, or a source patch. We then carefully inspected these materials to understand the cause of the errors and how they can be exploited. Although collecting these errors is not difficult, understanding, categorizing, and sometimes reproducing them absorbs most of our efforts.

These errors range across four main OS environments, including Windows, MacOS X, Linux, and Apple iOS. These errors are from a diverse set of 23 real-world programs, including kernels such as the Linux, system libraries such as GNU Libc, and user-space programs such as KDE, Apache, and Chrome. We hope this diversity increases the coverage and value of our dataset.

In the remaining of this section, we present five examples of exploitable concurrency errors.

Linux. Figure 1 shows an example concurrency error that corrupts pointer data in the Linux kernel. This violation is quite serious: a working exploit of this violation enables a local user to gain root access or execute arbitrary code within ring 0 [6, 29]. Specifically, this violation occurs as follows. To load a shared library in ELF format, a process issues system call `uselib()`, which subsequently calls function `load_elf_binary()` (Figure 1). This function correctly holds the semaphore `mmap_sem` the first time it modifies the current process's memory map structures (line 2–4). However, when it modifies these data structures the second time by calling `do_brk()` (line 7), it does not hold the right semaphore. Thus, another thread in the same process may be modifying the memory map

```
1 : load_elf_library(...) {
2 :     down_write(&current->mm->mmap_sem);
3 :     error = do_map(...); // CORRECT
4 :     up_write(&current->mm->mmap_sem);
5 :     ...
6 :     if(bss > len)
7 :         do_brk(...);
8 : }
9 : do_brk(...) {
10:     struct mm_struct * mm = current->mm;
11:     ...
12:     vma = kmem_cache_alloc(...);
13:     ... // initialize vma
14:     // ERROR! link vma to possibly stale mm!
15:     vma_link(mm, vma, ...); // link vma onto mm
16: }
```

Figure 1: *Linux kernel memory map corruption.*

```
1 : __nptl_setxid (struct xid_command *cmdp)
2 : {
3 :     lll_lock (stack_cache_lock);
4 :     // signal all threads on list to set user id.
5 :     // a thread is represented as a stack
6 :     list_for_each (runp, &stack_used)
7 :     {
8 :         struct pthread *t = list_entry (runp, struct pthread, list);
9 :         if (t == self)
10:             continue;
11:         setxid_signal_thread (cmdp, t);
12:     }
13:     lll_unlock (stack_cache_lock);
14:     // ERROR: does not wait for other threads to acknowledge
15: }
16: allocate_stack(...) { // called when a new thread is created
17:     lll_lock (stack_cache_lock);
18:     list_add (&pd->list, &stack_used);
19:     lll_unlock (stack_cache_lock);
20: }
```

Figure 2: *Glibc `setuid` race.*

structures concurrently while this `do_brk()` call is running, causing kernel memory corruption.

Glibc. Figure 2 shows a concurrency error that corrupts the user identities, and allows privilege escalation attacks [4]. This bug is caused by Glibc's default thread library, `nptl`, not handling `setuid()` atomically. In Linux, each kernel thread has its own set of user identities (user ID, effective user ID, etc). However, POSIX standards require that all other threads in the same process have identical user identities. Thus, when one thread calls `setuid()`, `nptl` has to ensure that all threads in the current process call `setuid()`. It does so using function `__nptl_setxid()` in Figure 2, which iterates through a list of all threads and signals each thread to call `setuid()` (line 6–12). However, this function releases the lock `stack_cache_lock` protecting the thread list, before it waits for all threads to finish setting their identifiers. A new thread may be created, and still have the old

```
1 : bool FastCopy (MonoArray *src, MonoArray* dest, int length){
2 :    // Checks that the type of dst[i] derive from src[i]
3 :    for (i = 0; i < length; ++i)
4 :       if(!safe_cast(type_of(src[i]), type_of(dest[i])))
5 :          return FALSE;
6 :
7 : //ERROR: another thread might run
8 : // dst[0] = object with incompatible type;
9 :
10:    // directly copy the bytes with memcpy()
11:    for (i = 0; i < length; ++i)
12:       memcpy(dest[i], src[i], size_of(ObjPtr));
13:    return TRUE;
14: }
```

Figure 3: *Moonlight fast array copy race.*

user identifiers. Since `setuid()` is often called to drop privileges, a thread skipping `setuid()` can thus result in privilege escalation.

Moonlight. Figure 3 shows an atomicity error [7] that allows an attacker to silently violate type safety in Moonlight, a Silverlight browser plugin implementation of the Mono open-source .NET framework. To speed up the array copying process, the `FastCopy()` method first checks that the types of the destination element and the source element are compatible (line 3–5) and, if so, performs a fast element-wise `memcpy()` instead of a slow copy implemented as CLR instructions. However, the type check and the copy are not implemented as one atomic step, allowing an attacker to change the destination array after the type check, compromising type safety. For instance, the attacker can create a new type with the same field layout, except that all fields in this new type are `public`, thus gaining access to the private fields in the original object.

MSIE. Another example is the MSIE R6025 exploit [2] which allows an attacker to launch a code injection attack to Microsoft Internet Explorer (IE) through a malicious webpage. Specifically, when IE opens the malicious page in multiple windows, the javascript code in the page calls the `appendChild()` method to append a DHTML element of one window to an element of another. A race in `appendChild()` can corrupt a function pointer in the heap. To reliably exploit this function pointer corruption, the attacker sprays the heap by repeatedly invoking the DHTML `createComments()` function, before calling `appendChild()`.

iOS. Our study also reveals *physical proximity attacks*, a unique class of attacks carried out in human-time. Such attacks typically exploit concurrency errors in the user interface (UI) logic. There have been several demonstrated vulnerabilities in the UI logic of Apple's iOS that allow attackers to bypass the passcode protection screen by executing a timed sequence of physical actions. Consider the latest vulnerability in iOS version

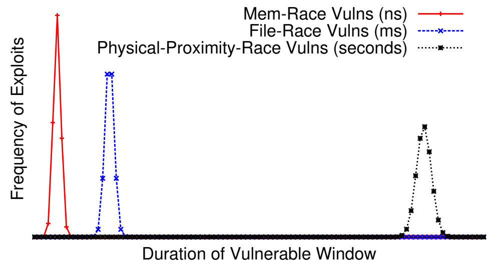

Figure 4: Our study suggests a likely *tri-modal* distribution of the duration of the vulnerable window for all concurrency attacks. Intuitively, this distribution can be broken into at least three distinguishable ranges, corresponding to concurrency errors culminating in *memory*, *file*, and *physical proximity* based exploit.

4. When presented with a passcode screen, an attacker can hit the "Emergency Call" button, enter a malformed phone number such as "###", and then quickly hit the screen lock button to bypass the passcode screen. Several other *physical proximity attacks* which exploit UI race conditions have been identified [1, 5, 8].

3 Observations of Concurrency Attacks

Vulnerable window duration heavily affects exploitability. In our analysis, we find that the exploitability of a concurrency error heavily depends on the duration of its *vulnerable window*—the timing window in which the concurrency error may occur. Figure 4 shows a tri-modal distribution of the vulnerable window duration suggested by our study.

Out of the 46 concurrency errors we studied, 3 allow physical proximity attacks. These errors have vulnerable windows measured in human time. Exploiting them is easy because attackers simply need to manually trigger a sequence of UI events. Of the 46 errors we studied, 13 allow file system TOCTOU attacks. These races have vulnerability windows measured in quanta of disk access time. This relatively large vulnerable window duration makes file races also easy to exploit: attackers typically re-run a command a few times (possibly using a shell script). Majority of the studied errors allow memory data to be corrupted or inconsistently exposed. The vulnerable windows of these errors are measured in quanta of memory access time. These errors are harder to exploit than the previous two classes of errors because attackers have to make the offending events occur within small timing windows. In addition, hardware cache leases or CPU time slices are often larger than these small windows, masking the errors.

Nonetheless, our study shows that the third class of errors can also be exploited using two styles of attacks. First, an attacker can retry many times to increase the probability of success. The MSIE error described in the previous section falls into this category, whose exploit repeatedly triggers the racing `appendChild()` calls in different threads. However, an excessive number of retires is easy to detect using for example anomaly detectors, so we expect this style of attack may not be as dangerous as the second style of attack, where attackers can use carefully crafted input to enlarge the vulnerability window. For instance, the exploit of the moonlight error in Figure 3 enlarges the vulnerability window by copying a large array, increasing the number of iterations of the type check loop (line 3–5). As another example, the exploit of the Linux error in Figure 1 enlarges the vulnerable window by triggering blocking operations such as disk access. Specifically, `do_brk` calls `kmem_cache_alloc` to allocate memory. In normal case when there is free memory, `kmem_cache_alloc` returns immediately, and the vulnerability window (line 10–15) is small. However, the exploit of this error allocates a large amount of memory to drive the system into low memory state, so that the call to `kmem_cache_alloc` has to swap used memory to disk to make room for this new allocation request. The vulnerable window thus lasts as long as a disk access, making it highly likely to corrupt the memory map.

Concurrency errors in API methods are particularly prone to concurrency attacks. The reasons are two fold. First, an API, such as the system call interface or the Silverlight runtime interface, often coincides with a protection boundary. That is, the application code often cannot access sensitive data directly. Instead, it has to do so through the API methods. To corrupt this sensitive data, an attacker has to exploit the errors in the API methods. The Linux and the moonlight errors are two examples illustrating this point. Second, an API is typically provided to support third-party, potentially untrusted programs. Leveraging this support, attackers can carefully craft malicious code of her choice to run on top of and *programmatically* exploit the buggy API. The Linux, the moonlight, and the MSIE errors are examples illustrating this point; their exploits were carefully crafted to retry an attack or force events to occur in a dangerous temporal order.

Concurrency attacks are more than just TOCTOU attacks. Our goal is to bring attention to general concurrency attacks that target errors in concurrent programs. These attacks are much broader than the TOCTOU attacks studied by previous work [27, 32, 33, 35]. The reasons are threefold.

First, the TOCTOU attacks in previous work target primarily the file system interface. This interface allows users to check file permissions and use file data, but does not directly support transactions that make the check and the use atomic. An attacker may thus exploit this limitation to gain illegal file accesses. In contrast, the general concurrency attacks we study may target many different programming interfaces such as a language runtime interface or the load/store memory interface, corrupt not just file data but general shared program data, and lead to effects more serious than illegal file accesses.

Second, the TOCTOU races in previous work exhibit one specific interleaving pattern: atomicity violations where the check and the use is not atomic. In contrast, the general concurrency errors we study may be simple read-write or write-write races, or execution order violations [25] where a set of accesses is supposed to occur in a fixed order, but no synchronizations enforce the order.

Third, as a natural fallout of the first two reasons, techniques proposed by previous TOCTOU work are too specific to detect or prevent general concurrency attacks. For instance, while launching a TOCTOU attack requires concurrent executions, the vulnerable program may be purely *sequential*, so TOCTOU detectors (*e.g.*, [35]) may not need to reason about concurrency at all. Similarly, TOCTOU detectors may mediate all file system calls without high runtime overhead (*e.g.*, [33]), but it would be prohibitive to mediate all load or store instructions to detect memory races.

4 Implications on Detection Techniques

It is unfortunate that existing concurrency-error detection techniques have not reached the maturity of sequential tools. Dynamic detectors are not good for detecting security vulnerabilities because they tend to cover only the executions or code run. Static detectors tend to give many false positives, burying the true errors.

Fortunately, leveraging the observations we made in the previous section, we can improve the effectiveness of these detection techniques. One idea is to prioritize detection towards the API methods at protection boundaries. These API methods must correctly protect sensitive data in face of abuses from arbitrarily malicious programs. In addition, errors in the API methods may have particularly bad impact as they may be used by a wide range of programs. A related idea is to prioritize detection toward sensitive data, such as user identities (corrupted by the glibc error), function pointers (MSIE), type data (Moonlight), process memory map (Linux).

Another idea is to rank the error reports of static detectors based on the vulnerable window duration, so that developers can inspect the errors that are more dangerous, *i.e.*, easier to exploit, first. For instance, if a vulnerable window of code may block, such as issuing as a disk or network I/O, or may loop an input-dependent number of iterations, then the corresponding error should be ranked high. Identifying code that may block is relatively straightforward: we can annotate the leaf oper-

```
// thread t1                          thread t2
taint[x] = taint[bad];
                                 taint[x] = taint[good];
                                 x = good;
x = bad;
```

Figure 5: *Data race renders taint tracking unsafe.*

ations that may block, then flag any function that may transitively call these operations as blocking. To identify input-dependent loop bounds, we may use taint analysis or symbolic execution to track where user inputs flow.

5 Implications on Defense Techniques

Security researchers have developed many defenses that prevent security exploits at runtime. However, they tend to assume only sequential programs. We thus want to understand (1) which defense techniques are still effective against concurrency attacks and (2) for those that are ineffective, how to fix them.

In this section, we attempt to answer these questions by analyzing a plethora of defense techniques [11, 14, 15, 19] from the research literature. Instead of describing how each of these defenses is weakened, we first extract five common mechanisms that underlie many of these defense tools such as memory safety tools, taint trackers, and intrusion detection systems. We then analyze how each mechanism is affected by concurrency.

Metadata tracking. Techniques such as taint tracking or memory safety enforcement track program data with metadata, such as taint tags or array bounds. If the tracked program has a data race, the race may manifest on the metadata owned by the defense technique, rendering it unsafe. Figure 5 illustrates this problem using a contrived example. The original code has a race on variable `x`: thread t1 assigns a tainted `bad` value to `x` and thread t2 assigns a untainted `good` value to `x`. The interleaving in the figure can cause the taint tag of `x` to be inconsistent with the value of `x`. That is, at the end of the execution, the tag of `x` indicates that `x` is untainted, but the value of `x` is `bad`.

Software checks. Many techniques rely on software checks to validate untrusted data. For instance, a taint tracker checks that a piece of data is untainted before using it in a dangerous operation; a memory safety tool checks that a pointer is within bounds before deferencing it; and a type checker ensures type safety (such the fast copy type check in Figure 3). These techniques, if unaware of concurrency, are prone to general TOCTOU attacks if the check and the use are not made atomic against concurrently running code. Software checks on stack data are typically not affected by concurrency errors because stack data is rarely shared.

Anomaly detection. Typical anomaly detection systems work by learning normal program behaviors, then detect deviations from the learned behaviors. Complications arise at both steps for concurrency attacks. For instance, if an anomaly detector learns behaviors only with respect to a single thread in a multithreaded system, it may miss anomalies involving multiple threads. On the flip side, if the anomaly detector models behaviors of all threads, the model may become overly complex and noisy. For instance, multiple threads may issue concurrent system calls, making the n-gram model [19] too noisy. In other words, we lack simple and accurate models for the behaviors of concurrent programs. (Content-based anomaly detection techniques [30] may still work.)

Hardware checks. Some techniques rely on hardware checks. For instance, several defense techniques prevent code injection attacks by marking pages non-executable via the NX bit. These techniques should work in concurrent models because the check is performed atomically by the hardware at the time of use.

Randomization. Address Space Randomization or instruction set randomization work by hindering the impact step. They should be equally effective for both concurrency and sequential attacks.

To summarize, three out of the five mechanisms discussed above are weakened by concurrency. Although fixing anomaly detection for concurrent programs may be difficult, fixing metadata tracking and software checks appear viable using standard approaches. For instance, a defense technique can use locks to enforce atomicity; it can also make a local copy of a piece of shared data, then perform the check and the use on the local data for atomicity. However, these fixes may introduce high performance overhead, and how to make them practical remains an open research challenge.

6 Related Work

Since we have discussed related work on attacks and defenses throughout this paper, this section focuses on related empirical studies of software errors and attacks. Previous work studied a large number of operating system errors [13]. The study focuses on sequential errors detected by an automated static analysis tool. Recently, Lu *et al.* studied many concurrency errors from real software such as MySQL and Apache [25]. Their analysis focuses on interleaving and memory access characteristics of concurrency errors, whereas ours focuses on the security, exploit, and defense aspects of the concurrency errors. Jalbert *et al.* created the RADBench concurrency error suite and proposed an approach to make them easy to reproduce [21].

Watson presented a specific concurrency attack against system call interposition [34]. Sender and Vidergar presented a toy example of concurrency attacks in web applications in Blackhat '08 [28]. These studies are not based on real concurrency errors; nor did they analyze broadly the detection and defense implications of the concurrency attacks.

7 Discussion

In this paper we catalogued concurrency attacks in the wild and presented their characteristics. We studied 46 different types of exploits and categorized them based on the duration of the vulnerabilities. We also observed that the risk of concurrency attacks is proportional to the duration of the vulnerability window, and further that attackers may be able to dilate the vulnerability windows to facilitate attack.

Our study of concurrency attacks and existing defenses inspire us to look for new, effective defense techniques. The reasons are three-fold. First, we note that some existing defense techniques such as taint tracking may fail to work in the presence of concurrency errors. Second, there are very few effective defense techniques for concurrency attacks that corrupt scalar data. Finally, based on our analysis of the wide spectrum of the concurrency-error exploits, a single mechanism is unlikely to defend against all types of concurrency attacks.

Consequently, two challenging research questions arise from our study. First, can we develop defense mechanisms which can mitigate all concurrency errors regardless of vulnerability window duration? Second, given an arbitrary program, can we identify, with some confidence, the most likely type of concurrency vulnerability to exist in a region of the program, assuming that a vulnerability does exist?

An important requirement is that defense mechanisms against concurrency attacks should not require *a priori* knowledge of the existence of particular concurrency errors. Traditionally, randomization techniques have been used to successfully mitigate unknown errors. For instance, address space randomization and instruction set randomization are often the "universal last resort" to mitigate many traditional sequential attacks. We believe that timing randomization techniques may be able to defend against unknown concurrency attacks.

Acknowledgement

We thank Madan Musuvathi and the anonymous reviewers for providing many helpful comments. John Gallagher helped in the initial stages of this work [37]. This work was supported in part by AFRL FA8650-11-C-7190 (DARPA MRC), FA8650-10-C-7024, and FA8750-10-2-0253 (DARPA CRASH); NSF CNS-1117805, CNS-1054906 (CAREER), CNS-1012633, and CNS-0905246; ONR N00014-12-1-0166; and a Sloan fellowship. This paper's opinions, findings, conclusions, and recommendations are those of the authors and do not necessarily reflect the views of the US Government.

References

[1] CVE-2010-1754. `http://www.cvedetails.com/cve/CVE-2010-1754`.

[2] MSIE javaprxy.dll COM object exploit. `http://www.exploit-db.com/exploits/1079`.

[3] Common vulnerabilities and exposures database. `http://cvedetails.com`.

[4] RHBA-2009:1634-1. `http://rhn.redhat.com/errata/RHBA-2009-1634.html`.

[5] CVE-2008-0034. `http://www.cvedetails.com/cve/CVE-2008-0034`.

[6] CVE-2004-1235. `http://www.cvedetails.com/cve/CVE-2004-1235`.

[7] CVE-2011-0990. `http://www.cvedetails.com/cve/CVE-2011-0990`.

[8] CVE-2010-0923. `http://www.cvedetails.com/cve/CVE-2010-0923`.

[9] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, 2010.

[10] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: Safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, 2009.

[11] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '05)*, pages 519–538, 2005.

[13] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Nov. 2001.

[14] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Symposium*, 1998.

[15] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[16] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[17] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[19] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (SP '96)*, 1996.

[20] ISO. C++0x Standards, ISO/IEC 14882:2011.

[21] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. Radbench: a concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism (HOTPAR '11)*, 2011.

[22] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI '11)*, 2011.

[23] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[24] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011.

[25] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, 2008.

[26] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, 2007.

[27] M. Payer and T. R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the Eighth International Conference on Virtual Execution Environments (VEE '12)*, pages 215–226, 2012.

[28] S. Sender and A. Vidergar. Concurrency attacks in web applications. Blackhat '08.

[29] P. Starzetz. uselib() privilege elevation. `http://www.isec.pl/vulnerabilities/isec-0021-uselib.txt`.

[30] S. J. Stolfo, F. Apap, E. Eskin, K. Heller, S. Hershkop, A. Honig, and K. Svore. A comparative evaluation of two algorithms for windows registry anomaly detection. *J. Comput. Secur.*, 13:659–693, July 2005.

[31] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[32] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *Sixth USENIX conference on File and Storage Technologies(FAST '08)*, 2008.

[33] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[34] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX workshop on Offensive Technologies (WOOT '07)*, 2007.

[35] J. Wei and C. Pu. TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study. In *Fourth USENIX conference on File and Storage Technologies(FAST '05)*, 2005.

[36] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and precise analysis of parallel programs through schedule specialization. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, 2012.

[37] J. Yang, A. Cui, J. Gallagher, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. Technical Report CUCS-028-11, Columbia University.

[38] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 251–264, 2011.