Concurrency Attacks

Junfeng Yang, Ang Cui, John Gallagher, Sal Stolfo, Simha Sethumadhavan
Columbia University

Abstract

Just as errors in sequential programs can lead to security exploits, errors in concurrent programs can lead to *concurrency attacks*. In this paper, we present an in-depth study of concurrency attacks and how they may affect existing defenses. Our study yields several interesting findings. For instance, we find that concurrency attacks can corrupt non-pointer data, such as user identifiers, which existing memory-safety defenses cannot handle. Inspired by our findings, we propose new defense directions and fixes to existing defenses.

1 Introduction

Two trends have caused concurrent programs to become pervasive and critical. The first is a hardware trend: the rise of multicore computing [7]. For years, sequential code enjoyed automatic speedup as computer architects steadily made single-core multiprocessors faster. Recently, however, power and wire-delay constraints [17] have forced microprocessors into multicore designs, and adding more cores does not automatically speed up sequential code.

The second trend is a software one: the coming storm of cloud computing [8]. More and more users are going online, requesting ever richer and more powerful—and thus computation and data intensive—services. These services, including many of those traditionally offered on desktops (*e.g.*, word processing), are now served from distributed "clouds" of servers to meet user demands for scalability, always-on availability, everywhere connectivity, and desirable consistency. To cope with this massive workload, practically all services employ concurrent programs to increase performance.

Unfortunately, despite our increasing reliance on concurrent programs, they remain much more difficult to write, test, and debug than sequential versions [18, 22]. This impediment has led to numerous subtle but serious *concurrency errors* in many widespread concurrent programs [19]. Even as tools for sequential programs are maturing [9, 11], concurrency errors are likely to become the dominant type of software errors in the near future.

Just as errors in sequential programs can lead to security exploits, concurrency errors can be similarly vulnerable and lead to *concurrency attacks*. Although initial evidence has shown that concurrency attacks are feasible [2], neither the research community nor industry have studied concurrency attacks to the same extent as sequential attacks [12, 20]. To better defend against concurrency attacks, it is key to thoroughly understand them. For instance, what are the characteristics of concurrency attacks? How do they work? How do they affect traditional sequential defense mechanisms?

In this paper, we present an in-depth study of concurrency attacks and how they may affect existing defenses. The raw data of our study consists of 46 potential attacks on concurrency errors from 23 real-world programs, including the Linux kernel, GNU Libc, and applications such as Apache and Chrome. For each attack, we investigate its attack path, analyzing the buggy interleaving of code, identifying data that is corrupted or inconsistently exposed, and categorizing the actual impact such as arbitrary code execution. We also investigate the effectiveness of existing defenses against these attacks.

Our study yields several interesting findings. For instance, we find that many concurrency errors compromise memory safety and can be exploited the same way as sequential errors. Thus, existing memory safety techniques, once made aware of concurrency, can defend against these attacks. However, some concurrency errors can corrupt non-pointer data, such as user identifiers or authentication flags, which existing memory-safety defenses cannot handle. We also analyze many existing defenses and identify common weaknesses caused by concurrency errors, and propose new defense directions and fixes to some of these weaknesses.

This paper makes three main contributions. First, to the best of our knowledge, we are the first to systematically study concurrency attacks and their implications for existing defenses. Second, the paper outlines several open research challenges in the defense of these attacks as well as new defense directions. We hope our initial work will motivate fellow researchers to work on addressing concurrency attacks. Lastly, we will release our study results including the automated attack scripts we construct as "concurrency attack specimens" to the research community.

This paper is organized as follows. We first describe our study methodology in (§2). We then describe our findings on concurrency attacks (§3) and existing defense techniques (§4). We then propose new defense directions

Figure 1: *Concurrency attack paths.*

(§5). Finally, we discuss related work (§6).

2 Study Methodology

2.1 Attacks

Our study consists of 46 potential concurrency attacks. To collect this data, we examined three sources of vulnerabilities: (1) the Common Vulnerabilities and Exposures (CVE) database [2], (2) real concurrency errors extensively studied in previous work [19], and (3) the bug databases of real open-source software. From these sources, we included only concurrency errors that are (potentially) 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 root cause of the errors and how they may be exploited. The raw data of the errors is available online [2].

The errors in our dataset range across five main OS environments, including Windows, MacOS X, Linux, iPhone OS, and CISCO IOS. These errors are from a diverse set of 23 real-world programs, including kernels such as the Linux kernel, 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.

When studying the concurrency attacks, we classify them based on their *attack paths*, which have the form "*interleaving* → *data* → *impact*." Specifically, we define a taxonomy as in Figure 1 with the following semantics:

- *Interleaving:* the origin of a concurrency attack, namely, the buggy interleaving of code execution that makes the attack possible. For instance, the interleaving may violate *atomicity* (multiple memory accesses must occur atomically) or *execution order* (multiple memory accesses must occur in a fixed order) constraints.

- *Data:* the intermediate effects of a concurrency attack, namely, what data the attack corrupts.[1] There may be one or more data steps in a concurrency attack. For instance, once a pointer is corrupted, a `store` instruction via this pointer can further corrupt other program data.

- *Impact:* the results of a concurrency attack, such as denial of service via program crashes, code injection, or privilege escalation. An attack may have more than one results: for instance, once an attacker can run arbitrary code, she can easily escalate her privilege set.

A previous study [19] investigated interleaving patterns in depth. Our dataset differs from the dataset used in the previous study in three ways: (1) some errors in our dataset come from programs communicating by passing messages; (2) some errors are in the kernel or user-space libraries; and (3) our focus is on exploits not just any bugs. Nonetheless, we find that the patterns of the interleavings in our dataset match those previously observed. Thus, we do not discuss the interleaving patterns further.

Our key questions of the attack study are therefore

- Which data is corrupted or inconsistently read, pointer data or scalar data (*i.e.*, non-pointer data)?
- How are the concurrency errors exploited?

2.2 Defense Techniques

An important goal of our study is to analyze the efficacy of existing defense techniques against concurrency attacks. These techniques operate at different steps along an attack path, some of which operate at more than one steps. For instance, memory safety techniques (*e.g.*, [21]), StackGuard [14], and PointGuard [15] prevent memory corruption at the data step. Address Space Randomization (ASR) [10] makes it harder to exploit memory corruptions. Anomaly detection techniques (*e.g.*, [16]) look for abnormal system behaviors along all steps of an attack path.

Our key questions of the defense analysis are

1. What defense techniques still work and what are weakened by concurrency attacks?
2. How do we strengthen the weakened defense techniques?

When doing this analysis, we consider conceptual attacks because the real concurrency attacks in our dataset target specific vulnerabilities in programs, not defense techniques, presumably because few defense techniques have gained popularity.

2.3 Caveat

The main caveat to keep in mind with our results is that, as in previous empirical studies, our dataset does not contain the universe of all concurrency attacks, nor is it a uniform sample of the universal set. Thus, it is unknown whether our dataset is representative. We attempt to compensate for this issue by collecting concurrency attacks from multiple sources and from various real-world programs that spread across the kernel, libraries, and different types of user-space applications.

[1] Sometimes an attack does not literally corrupt data; instead, it causes some code to read inconsistent data. For brevity, we broadly define these effects as corrupting data.

Findings	Implications
A majority (24 out of 46) of the concurrency attacks corrupt pointer data.	Existing memory safety tools, once made aware of concurrency, may be able to prevent concurrency attacks that corrupt pointer data.
9 concurrency attacks *directly* corrupt scalar data, such as user identifiers, without compromising memory safety.	Few existing defenses handle attacks that directly corrupt scalar data.
Many existing defenses become unsafe in the face of concurrency errors	These defenses must consider concurrent execution.
The exploitability of a concurrency error highly depends on the duration of its *vulnerable window* (*i.e.,* the timing window within which the concurrency error may occur).	New defense techniques may reduce the exploitability of concurrency errors by reducing the duration of the vulnerable window.

Table 1: *Summary of Findings.*

2.4 Summary of Key Findings

We summarize the key findings from our study in Table 1. In the next two sections (§3 and §4), we provide more detailed explanations, as well as real examples, of these findings.

3 Findings on Concurrency Attacks

In this section, we present our findings on concurrency attacks, focusing on the data step (§3.1) and the impact step (§3.2) along the attack paths.

3.1 Data

In our dataset, 13 out of 46 concurrency errors lead to file corruption, and the other 35 errors corrupt program data. Since previous work has studied file system Time-of-Check-to-Time-of-Use (TOCTOU) races in detail, in this paper we focus on program data.

We further distinguish program data into *pointer data* and *scalar data*. We define pointer data as data that, once corrupted, compromises memory safety. For instance, regular pointers or array indexes are pointer data. We define scalar data as all other program data. We distinguish these two types of data because pointer data corruption can largely be exploited the same way for both concurrency and sequential attacks, whereas scalar data corruption is unique to concurrency attacks.

Of the 35 concurrency errors that corrupt program data, 24 involve pointer data and the other 9 involve scalar data. Figure 2 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 [3]. 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 2). 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

```
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:     vma = kmem_cache_alloc(...);
11:     ... // initialize vma
12:     // ERROR! modify mmap without holding mmap_sem
13:     vma_link(mm, vma, ...); // link vma onto mm
14: }
15:
```

Figure 2: *Linux kernel memory map corruption.*

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

Figure 3 shows a concurrency error that corrupts scalar data, in particular user identities, and allows privilege escalation attacks. 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 3, 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 user identifiers. Since

```
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 3: *Glibc* `setuid` *race.*

```
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: }
15:
```

Figure 4: *Moonlight fast array copy race.*

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

Figure 4 shows an atomicity error which allows an attacker to silently violate type safety in Moonlight, the Mono implementation of the Silverlight browser plugin. To speed up an array copying process, the `FastCopy()` method in the Mono CLR 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.

3.2 Impact

Once a concurrency error corrupts pointer data and compromises memory safety, an attacker can exploit the corruption by leveraging the same techniques for sequential errors, such as launching denial of service (DoS) by crashing the program, injecting code, or escalating privileges. One example is the kernel memory map corruption in Figure 2. Another example is the MSIE R6025 exploit [1] 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()`.

A concurrency error can also corrupt scalar data without compromising memory safety. At a basic level, scalar data corruption compromises data integrity. Worse, by exploiting scalar data corruption, an attacker can often launch privilege escalation attacks. All the studied concurrency errors of this kind, including the two examples (Figure 4 and Figure 3) shown in previous section, enable privilege escalation attacks.

In our analysis, we also 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. The concurrency errors presented so far have vulnerable windows measured in quanta of memory access time. The moonlight error in Figure 4 is particularly dangerous because an attacker can enlarge the vulnerability window of the error by copying a large array and increasing the number of iterations of the type check loop (line 3–5). The file system TOCTOU races often have vulnerability windows measured in quanta of disk access time. Our study also reveals *physical proximate attacks*, a unique class of attacks carried out in human-time. Such attacks typically exploit concurrency errors in the user interface (UI) logic.

For instance, 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 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 race-condition vulnerabilities within GUI's have been identified [4–6].

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

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

The apparent tri-modal distribution of the duration of *vulnerable window* of analyzed attacks suggests that this feature may be used to identify the general type of concurrency vulnerability. Furthermore, since such attacks tend to be highly time-sensitive, the *expected* duration of the vulnerable window within any region of a vulnerable program may be used to improve the efficacy of randomization-based defenses against these attacks.

4 Findings on Defense Techniques

As discussed in previous section, many concurrency errors can be exploited in the same ways as sequential errors. It is thus key to understand (1) which defense techniques are 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 from the rich research literature. Specifically, we extract five common mechanisms that underlie many defense techniques such as memory safety tools, taint trackers, and intrusion detection systems. For each mechanism, we analyze how it 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 4). These techniques, if unaware of concurrency, are prone to 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.

Static analysis. Static bug detectors have been very effective at finding sequential errors. In addition, techniques such as memory safety enforcement often rely on sophisticated static analysis to identify places where bounds checking is unnecessary. However, static analysis for concurrent programs tend to be very inaccurate. Thus, it is unlikely we will get precise static bug detectors or other analysis tools for concurrent programs.

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 [16] too noisy. In other words, we lack simple and accurate models for the behaviors of concurrent programs. (Content-based anomaly detection techniques 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, four out of the six mechanisms discussed above are weakened by concurrency. Although fixing static analysis or 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 the 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.

5 New Defense Directions

Our study of concurrency attacks and existing defenses inspire us to look for new, effective defense techniques. The reasons are two-fold. First, for concurrency attacks corrupting scalar data, we have few or no effective defense techniques. Second, based on our analysis of the wide spectrum of the exploit types of concurrency errors, we posit that it is unlikely that a single mechanism can defend against all types of concurrency attacks.

If we know the program location and cause of an exploitable concurrency error, we can use techniques such as LOOM [23] to fix these known vulnerabilities. How-

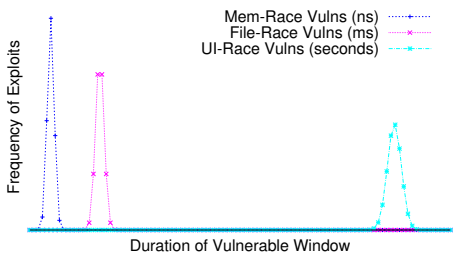

Figure 6: Our study suggests a likely *tri-modal* distribution of the duration of the vulnerable window for all concurrency attacks.

ever, deployed concurrent programs often contain many unknown concurrency errors because it is extremely difficult to write correct concurrent programs or check them. Thus, we focus on defense mechanisms which do not require *a priori* knowledge of the existence of concurrency errors.

Randomization techniques can often 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 we can develop similar randomization techniques to defend against unknown concurrency attacks. Specifically, we can randomize timing to hinder an attacker from exploiting predicted duration and timing of the vulnerability window of a concurrency error.

Figure 6 shows a likely *tri-modal* distribution of the duration of the vulnerability windows for all concurrency errors analyzed in our study. Intuitively, this distribution can be broken into at least three distinguishable ranges, corresponding to concurrency errors culminating in *memory*, *file*, and *physical proximate* based exploit.

Two challenging research questions arise. 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?

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.

Chen *et al.* studied *non-control data* attacks [12], the attacks that do not compromise control flow integrity. However, Chen's study focused on sequential errors, and the proposed non-control data attacks still originated from memory corruption attacks. In contrast, our study focuses on concurrency errors, and attacks corrupting scalar data do not cause memory corruption at all.

Recently, Lu *et al.* studied many concurrency errors from real software such as MySQL and Apache. 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.

References

[1] Msie javaprxy.dll com object exploit. `http://www.exploit-db.com/exploits/1079/`.

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

[3] Cve-2006-4814. `http://www.cvedetails.com/cve/CVE-2006-4814`.

[4] Cve-2008-0034. `http://www.cvedetails.com/cve/CVE-2008-0034/`, .

[5] Cve-2010-1754. `http://www.cvedetails.com/cve/CVE-2010-1754/`, .

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

[7] A. Agarwal and M. Levy. The kill rule for multicore. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 750–753, 2007.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[9] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

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

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, Dec. 2008.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium*, pages 177–192, 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 7th conference on USENIX Security Symposium - Volume 7*, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1267549.1267554`.

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

[16] 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*, 1996.

[17] M. Horowitz. Scaling, power and the future of cmos. In *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, page 23, 2007.

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

[19] 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)*, Mar. 2008.

[20] H. Meer. Memory Corruption Attacks: The (Almost) Complete History... In *In BlackHat USA*, 2010.

[21] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 128–139, 2002.

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

[23] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.