Understanding and Detecting Concurrency Attacks

Paper #82

The University of Hong Kong

@cs.hku.hk

Abstract

Just like sequential bugs lead to attacks, concurrency bugs also lead to concurrency attacks. There are various tools working on concurrency bug detection, diagnosis, and correction. Unfortunately, existing tools could not efficiently detect and help understand these attacks. Compared with concurrency bugs, concurrency attacks are more severe since they may have already been exploited by attackers, though concurrency bugs have been fixed. This paper finds two challenges for detecting concurrency attacks and presents a general model with a practical framework for understanding and detecting concurrency bugs – OWL.

Our study on 10 widely used programs reveals 26 concurrency attacks with broad threats (e.g., OS privilege escalation), and we built scripts to successfully exploit 10 attacks. Our study further reveals that, only extremely small portion of inputs and thread interleaving (or schedules) can trigger these attacks, and existing concurrency bug detectors work poorly because they lack help to identify the vulnerable inputs and schedules.

Our key insight is that the reports in existing detectors have implied moderate hints on what inputs and schedules will be likely to lead to attacks and what will not (e.g., benign bug reports). With this insight, OWL extracts hints from the reports with static analysis, augments existing detectors by pruning out the benign inputs and schedules, directs detectors with its own runtime vulnerability verifiers to work on the remaining likely vulnerable inputs and schedules, and finally give possible inputs based on fuzzer triggering concurrency attacks by exploiting the concurrency bug.

Evaluation shows that OWL reduced 94.3% reports caused by benign inputs or schedules and detected 7 known concurrency attacks. OWL also detected 3 previously unknown concurrency attacks, including a use-after-free attack in SSDB confirmed as CVE-2016-1000324, an integer overflow, HTML integrity violation in Apache and three new MySQL data races confirmed with bug ID 84064, 84122, 84241. All OWL source code, exploit scripts, and results are available at https://github.com/ruigulala/ConAnalysis.

1. Introduction

Multi-threaded program is hard to be correct. Concurrency bugs are common in modern multi-threaded programs including atomic violation, order violation, and others. Extant work well explores interleaving that causes concurrency bugs, and efficiently detects explicit concurrency bugs that direct to severe consequences such as execution order violation, wrong output and program crash [46, 50, 68, 78, 86, 87].

Recent studies[66, 83] show rise of concerns about *concurrency attacks*. By triggering concurrency bugs, hackers may leverage the corrupted memory to conduct attacks including privilege escalations[8, 10], hijacking code execution[9], bypassing security checks[4–6], and breaking database integrity[66]. These vulnerabilities are often hidden in huge amount of concurrency bugs and implicit in program behaviors. Concurrency attacks are much more insidious and harmful than concurrency bugs. A privilege escalation attack may cause no outrageous effect, but possess a permanent security hole hiding in the system. Also, despite the subtle inputs that induce concurrency bugs, concurrency attacks may need other crafted inputs for exploitation of the vulnerabilities.

Unfortunately, although great progress has been made, there does not exist a general model, as well as practical and automatic tools for understanding and detecting concurrency attacks. Even some concurrency bugs has been detected and reported, professional may still miss the potential vulnerabilities caused by the bug. For instance, *apache-25520*[2] has been reported over years and well studied by researchers[47]. We are the first to exploit a new heap overflow attack leveraging on this bug and break the HTML integrity. This is dangerous because attacks may have been already exploited in wild. Hence, knowledge and detection of concurrency attacks is of crucial importance.

We studied 26 concurrency attacks and find two major challenges for detecting concurrency attacks. First, it's hard to distinguish the exact concurrency bug report which can conduct attack exploitation. Concurrency attacks are hidden in huge amount concurrency bug reports. For instance, a popular data race detector TSAN generates 1123 race reports running `MySQL`'s benchmark. However, 98% reports are benign race and only 2 of rest data races are vulnerable and finally conduct 2 concurrency attack. Current concurrency bug detection tools are not designed to analyze whether a concurrency bug is vulnerable and exploitable. Hence developers need lots of efforts distinguishing and analyzing the reports given by concurrency bug detection tools. In practice, this is not compatible with today's software development.

Second, extant work ignores indicating extra inputs to conduct concurrency attacks. A concurrency bug may become much more vulnerable when attackers 1.craft inputs to trigger the bug; 2.employee other inputs running to construct their attacks. In CVE-2017-7533, we first leverage two crafted inputs running on two threads to trigger a data race and construct kernel heap overflow. We also require another inputs running on *victim thread* to lay the target structure on the same heap. By corrupting the target structure, we finally achieve arbitrary code execution and get a root shell. Automatically indicating the inputs that construct concurrency attacks would be of vital helpful for developers to better understand the latent vulnerabilities.

To address the two challenges, we present a general model(§3.2) for understanding concurrency attacks. The model breaks down concurrency attacks into three stages: bug happening, bug-to-attack propagation, and attack happening. In this model, a concurrency bugs is triggered by bug-induced inputs. Then corrupted memory propagates through control flow and data flow of program execution. When corrupted memory propagates to vulnerable sites, a concurrency attack may be exploitable. This procedure addresses the first challenge. If we can provide hints for verified concurrency bug happening, and bug-to-attack propagation analysis, then developers can receive early warnings on concurrency attacks.

In special case, during data flow propagation, a buffer overflow may happen and additional threads' memory can be corrupted. In traditional sequential attacks, attackers can leverage buffer overflow to construct attacks like code hijacking, ... []. In concurrency situation, attackers can still conduct buffer overflow attack. However, indicating the other inputs is key for the second challenge. In sum, this model covers 26 concurrency attacks we studied.

Leveraging the model, we designed a practical, scalable and inter-procedural concurrency attack detection framework(§3.3), XXX. The framework contain two phases. The first phase is *concurrency analyzer* to detect concurrency attacks, which augment current concurrency bug detectors. We provide two extra hints for explore concurrency attacks. One hint is the benign schedules. The benign reports caused by adhoc synchronizations have already implied how these synchronizations act and how they work out schedules. Therefore, we can use static analysis to extract these synchronizations from the reports, automatically annotate these synchronizations in a program, then we can greatly prune out these benign schedules and their reports. The other hint is the bug-to-attack propagations, which imply vulnerable inputs. Our study found that most vulnerable races are already included in the race detectors reports (§**??**), and concurrency attacks sites are often explicit in program code (§**??**). Therefore, we can perform static analysis on only the data and control flow propagations between the bug reports and the potential attack sites, then we can collect relevant call stacks and branch statements as the potentially vulnerable input hints.

[TODO]The second phase is *concurrency fuzzer?* to provide another hint to indicate extra inputs for constructing a concurrency attack. The key goal of this phase is to find the input playing the victim role of concurrency attack. For instance, in CVE-2017-7533, we take a input that act as victim. And construct a kernel heap overflow to corrupt the victim structure's memory. By corrupting the memory, we successfully detected a arbitrary code execution vulnerability. Usually, the victim structure appearing on the heap is allocated with same memory allocation function. Hence we leverage this assumption, and indicate the victim inputs.

We implemented XXX on Linux, supporting both user space and kernel space concurrency attack detection. XXX adopts several race detectors including TSAN, VALGRIND for user space and KTSAN, SKI for kernel space. We evaluated XXX on 6 diverse, widely used programs, including `Apache`, `Chrome`, `Libsafe`, `Linux`, `MySQL`, and `SSDB`. XXX s benign schedule hints and runtime verifiers reduced 94.3% of the race reports, and it did not miss the evaluated concurrency attacks. With the greatly reduced reports, XXX s vulnerable input hints helped us identify subtle vulnerable inputs, leading to the detection of 7 known concurrency attacks as well as 4 previous unknown, severe ones in `SSDB` and `Apache`. The analysis performance of XXX was reasonable for in-house testing.

This paper makes two major contributions:

1. **A general model for understanding concurrency attacks.** This model explains most concurrency attacks in wild and providing two major direction for detecting concurrency attacks.
2. **A practical concurrency attack detection framework and its implementation, XXX.** XXX can easily employ existing concurrent bug detectors and vulnerability analyzer to improve the accuracy and ??? of detection.

The rest of this paper is structured as follows. §2 introduces the background of concurrency attacks. §3 gives an overview on the concurrency attack model and architecture of XXX. §4 describes the design of XXX. §5 states the implementation of XXX. In §6, we discuss our limitations and future work. We evaluated XXX and show results in §7. We talk about related work in §8 and make a conclusion in §9.

2. Background

2.1 Concurrency Bug

In multi-threaded program, concurrency bugs (unsynchronized memory access) is common and caused great loss in real world[12, 42]. Data race is a significant leading factor of concurrency bug and occurred when two threads access the same memory piece concurrently and at least one access is write[30, 59, 85]. Data race may cause order vilotion, atomicity violation and other concurrency bugs. Data race detectors has been mature in industry[50, 68] and readily detect most data races occurred. However, a previous study [47] shows that data races reported by data race detectors do not necessarily cause a concurrency bug. Many data races are benign race and cause no factors, e.g.while-flags. Also, our observation shows there are benign schedules in race reports. For instance, developers use semaphore-like adhoc synchronizations, where one thread is busy waiting on a shared variable until another thread sets this variable to be “true”. This type of adhoc synchronizations couldnt be recognized by TSAN or SKI and caused many false positives. In our framework XXX, we firstly reduce these benign race reports and then do further concurrency attack detection.

2.2 Concurrency Attack

Extant studies [66, 83] show rise of concurrency attacks. We conclude two main features for concurrency attacks comparing to concurrency bugs. First, concurrency attacks make much more severe threats: concurrency attacks can corrupt critical memory and cause four types of severe security consequences, including privilege escalations[8, 11], malicious code injections [9], bypassing security authentications[4–6] , and breaking database integrity[66].

Second, concurrency bugs and attacks can often be easily triggered by crafted program inputs, and consequences may become much more severe when attacks involves extra inputs. Existing concurrency bug detection tools are not designed to indicate whether a concurrency bug is vulnerable and exploitable.

2.3 Bug-to-Attack Propagation

In traditional sequential context, attacks are triggered by certain inputs. Program path and execution order is often determined once input is given. In concurrency context, attacks are triggered by both input and interleaving[1, 83]. There has been tools to analyze sequential attacks. They often trace the input and do program path analysis. For instance, mayhem[23] is the first to propose detecting sequential attacks and automatically generating exploit scripts, which has made great progress. It employees a hybrid approach of concrete and symbolic execution and achieves automatically exploiting vulnerabilities.

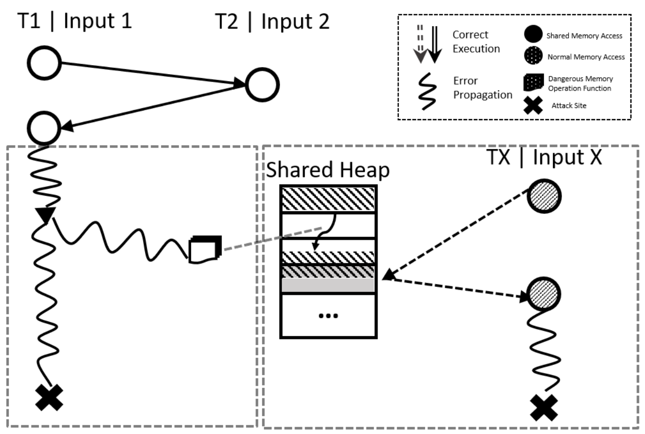


Figure 1: ***Concurrency Attack Model***

However, conventional sequential approach is not suitable in concurrency context. We conclude in two points. First,

Con-seq intra-procedural propagation ..

3. Overview

We introduce a model that explains most concurrency attacks we studied. Leveraging the model, we design a framework XXX to detect concurrency attacks. This section introduces some preliminaries, makes an overview of the model and architecture of XXX, and gives an example of how we employ our framework to exploit a concurrency attack.

3.1 Preliminary

Input To ease discussion, we use input to broadly refer to the data a program reads from its execution environment, including not only the data read from files and sockets, but also command line arguments, return values of external functions such as gettimeofday, and any external data that can affect program execution

Bug-inducing input The inputs that trigger a concurrency bug.

Attack-inducing input The inputs that trigger a concurrency attack.

Attacker thread The threads employed by attackers to race other thread.

Infected thread The threads raced by attacker thread. In some case, a thread can be both infected and victim.

Victim thread The threads where attacks happen. In some case, a thread can be both infected and victim.

3.2 Concurrency Attack Model

3.3 XXX’s Architecture

Concurrency bug detector wraps existing concurrency bug detection tools and detects concurrency bugs. It receives program executables and inputs, and produces runtime concurrency bug reports. Data race detectors are mature for detecting concurrency bugs in industry and has been widely de-

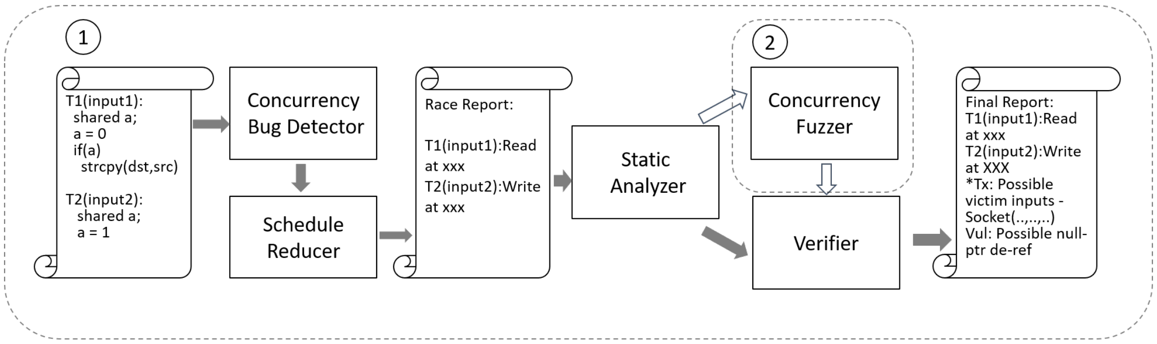


Figure 2: ***Architecture***

ployed and studied. We adopt current popular data race detector includes TSAN, VALGRIND, SKI and KTSAN (§5).

Schedule Reducer reduces benign schedules (e.g., adhoc synchronization), and verifies real concurrency bug happening. It receives concurrency bug reports and program IR file, and produces eliminated concurrency bug reports. We take static analysis approach to and leverage the runtime information from bug reports, which is much simpler and more precise than prior static adhoc synchronization identification tool SyncFinder[79].

Intra-procedural Static Analyzer does inter-procedural static analysis to see whether corrupted memory may propagate to any vulnerability sites (§**??**) through data flow or control flows. Also, it give hints for potential intra-procedural propagation (e.g., heap overflow). It receives eliminated concurrency bug reports and program IR file, and produces vulnerability reports. We introduces a static analysis algorithm and combines runtime results from concurrency bug detectors to produce precise and scalable vulnerability reports and input hints.

Inter-procedural Fuzzer finds potential inputs running on victim thread that can be corrupted by intra-procedural propagation(e.g., heap overflow). It receives hints from intro-procedural static analyzer, target program executables and crowd-sourced benchmarks. It produces hints for potential victim inputs.

3.4 Detecting Example

We take our exploitation of CVE-2017-7533 as a example of using our framework to produce hints for concurrency attack. When we running an inotify and rename concurrently, a race report of

4. Framework

4.1 Integrate Concurrency Bug Detectors

XXX has integrated two popular race detectors: SKI for Linux kernels and TSAN for application programs. To integrate XXX's algorithm (§**??**) with concurrency bug detectors, two elements are necessary from the detectors: the `load` instruction that reads the bug's corrupted memory and the instruction's call stack.

SKI's default detection policy is inadequate to our tool because it only reports the pair of instructions at the moment when race happens. This policy incurs two issues for our integration. First, the pair of instructions could both be write instructions, which does not match the algorithm's input format. Second, it is essential to provide to the algorithm an as detailed call stack, which reads from the corrupted racy variable, as possible.

We modified SKI's race detection policy as follows. After a race happens, the physical memory address of the variable will be added to a SKI watch list, marking such variable as corrupted. All the call stacks of the following read to the watched variable will be printed. If a write to a watched variable occurs, such write sanitizes the corrupt value and removes the variable from the watch list. In this way, we can catch all the call stacks of potential problematic use of racy variables. The final race report will show all the stacks of the reading thread.

Another issue for XXX to work with kernels is that SKI lacks call stack information. We configure Linux kernel with the CONFIG_FRAME_POINTER option enabled. Given a dump of the kernel stack and the values of the program counter and frame pointer, we were able to iterate the stack frames and constructed call stacks.

4.2 Reduce Benign Schedule

Developers use semaphore-like adhoc synchronizations, where one thread is busy waiting on a shared variable until another thread sets this variable to be "true". This type of adhoc synchronizations couldn't be recognized by TSAN or SKI and caused many false positives.

XXX uses static analysis to detect these synchronizations in two steps. First, by taking the race reports from detectors, it sees if the "read" instruction is in a loop. Then, it conducts a intra-procedural forward data and control dependency analysis to find the propagation of the corrupted variable. If XXX encounters a branch instruction in the propagation chain, it checks if this branch instruction can break out of the loop. Last, it checks if the "write" instruction of the instruction assigns a constant to the variable. If so, XXX tags this report as an "adhoc sync".

Compared to the prior static adhoc sync identification method SyncFinder [79], which finds the matching "read" and "write" instruction by statically searching program code, our approach leverages the actual runtime information from the race reports, so ours are much simpler and more precise.

XXX's dynamic race verifier checks whether the reduced race reports are indeed real races. It also generates security hints for the following analysis. The verifier is lightweight because it is built on top of the LLDB debugger. We find that a good way to trigger a data race is to catch it "in the racing moment". The verifier sets thread specific breakpoints indicated by TSAN race reports. "Thread specific" means when the breakpoint is triggered, we only halt that specific thread instead of the whole program. The rest of the threads are still able to run. In this way, we can actually catch the race when both of the racing instructions are reached by different threads and are accessing the same address.

For each run, XXX's dynamic filter verifies one race. Once a data race is verified, the verifier goes one step further. It prints the following dynamic information as security hints including, the racing instructions from source code, the value they're about to read and write and the type of the variable that these instructions are about to read or write. These hints show whether a NULL pointer difference can be triggered or an uninitialized data can be read because of the race.

It is possible that due to the suspension of threads, the program goes into a livelock state before verifying any data races. We resolve this livelock state by temporarily releasing one of the currently triggered breakpoints.

Previous works [**? ? ?**] adopt the same core idea of thread specific breakpoints and data race verification. XXX's dynamic race verifier provides a lightweight, general, easy to use way (integrated with existing debugger) in verifying potentially harmful data races and their consequences. Compared with RaceFuzzer [**?**], XXX's verifier achieves the goal without requiring heavyweight Java instrumentation. Compared with ConcurrentBreakpoint [**?**] and ConcurrentPredicate [**?**], we require no code annotations and importing libraries.

Overall, XXX's dynamic filter makes developers be less dependent on the particular front end race detector, because no matter how many false positive the front end race detector generates, this verifier will make sure the end result is accurate.

There are two cases that could cause XXX's race verifier to miss real races. First, if the race detector doesn't detect the race upfront, the verifier won't report the race either. Second, depending on runtime effects (e.g., schedules), some races can't be reliably reproduced with 100% success rate [**?**].

4.3 Static Analysis

Algorithm 1 show XXX's vulnerability analyzer's algorithm. It takes a program's LLVM bitcode in SSA form, an LLVM `load` instruction that reads from the corrupted memory of a bug report, and the call stack of this instruction.

Algorithm 1: Vulnerable input hint analysis

```
Input  : program prog, start instruction si, si call stack cs
Global: corrupted instruction set crptIns, vulnerability set
        vuls
DetectAttack(prog, si, cs)
    crptIns.add si
    while cs is not empty do
        function ← cs.pop
        ctrlDep ← false
        DoDetect(prog, si, function, ctrlDep)
DoDetect(prog, si, function, ctrlDep)
    set localCrptBrs ← empty
    foreach succeeded instruction i do
        bool ctrlDepFlag ← false
        foreach branch instruction cbr in localCrptBrs do
            if i is control dependent on cbr then
                ctrlDepFlag ← true
        if ctrlDep or ctrlDepFlag then
            if i.type() ∈ vuls then
                ReportExploit(i, CTRL_DEP)
        if i.isCall() then
            foreach actual argument arg in i do
                if arg ∈ crptIns then
                    crptIns.add i
                    if i.type() ∈ vuls then
                        ReportExploit(i, DATA_DEP)
            if f.isInternal() then
                cs.push f
                DoDetect(prog, f.first(), f, ctrlDep or
                ctrlDepFlag)
                cs.pop
        else
            foreach operand op in i do
                if op ∈ crptIns then
                    if i.type() ∈ vuls then
                        ReportExploit(i, DATA_DEP)
                    crptIns.add i
                    if i.isBranch() then
                        localCrptBrs.add i
ReportExploit(i, type)
    if i is never reported on type then
        ReportToDeveloper()
```

The algorithm then does inter-procedural static analysis to see whether corrupted memory may propagate to any vulnerable site (§**??**) through data or control flows. If so, the algorithm outputs the propagation chain in LLVM IR format as the vulnerable input hint for developers.

The algorithm works as follows. It first adds the corrupted read instruction into a global corrupted instruction set, it then traverses all following instructions in the current function and if any instruction is affected by this corrupted set ("affected" means any operand of current instruction is in this set), it adds the instruction into this corrupted set. The algorithm looks into all successors of branch instructions as well as callees to propagate this set. It reports a potential concurrency attack when a vulnerable site (§**??**) is affected by this set.

To achieve reasonable accuracy and scalability, we made three design decisions. First, based on our finding that bugs and attacks often share similar call stack prefixes, the algorithm traverses the bug's call stack (§**??**). If the algorithm does not find a vulnerability on current call stack and its callees, it pops the latest caller in current call stack and checks the propagation through the return value of this call, until the call stack becomes empty and the traversal of current function finishes. This targeted traversal makes the algorithm scale to large programs with greatly reduced false reports (Table **??**).

Second, the algorithm tracks propagation through LLVM virtual registers [45]. Similar to relevant systems [86**?**], our design did not incorporate pointer analysis [41, 76] because one main issue of such analysis is that it typically reports too many false positives on shared memory access in large programs (§**??**).

Our analyzer compensates the lack of pointer analysis by: (1) tracking read instructions in the detectors at runtime (§**??**), and (2) leveraging the call stacks to precisely resolve the actually invoked function pointers (another main issue in pointer analysis).

Third, some detectors do not have read instructions in the reports (e.g., write-write races), and we modified the detectors to add the first `load` instruction for these reports during the detection runs (§**??**).

All five types of vulnerability sites we found (§**??**) have been incorporated in this algorithm. The generated vulnerability reaching branches from this algorithm serve as vulnerable input hints and helped us identify subtle inputs to detect 7 known attacks and 3 previously unknown ones (§**??**).

4.4 Find Potential Victims

4.5 Dynamic Vulnerability Verifier

XXX's dynamic vulnerability verifier is built on LLDB so it is lightweight. It takes the input from its static vulnerability analysis, including the vulnerability site and the associated branches. It re-runs the program again and prints out whether one could reach the vulnerability site and trigger the attack. If the site cannot be reached, it prints out the diverged branches as further input hints.

5. Implementation

6. Discussions

7. Evaluation

We evaluated XXX on 6 widely used C/C++ programs, including three server applications (`Apache` [14] web server, `MySQL` [13] database server, and SSDB [65] key-value store server), one library (`Libsafe` [43]), the 4.11.9 Linux kernel, and one web browser (`Chrome`). We used the programs' common performance benchmarks as workloads. Our evaluation was done on XXX.

We focused our evaluation on four key questions:

1. Can XXX detect known concurrency attacks in the real-world (§**??**)?
2. Can XXX detect previously unknown concurrency attacks in the real-world (§**??**)?
3. How many false-positive reports from concurrency error detection tools can XXX reduce (§**??**)?
4. How many potential victim inputs can XXX indicate(§3)?

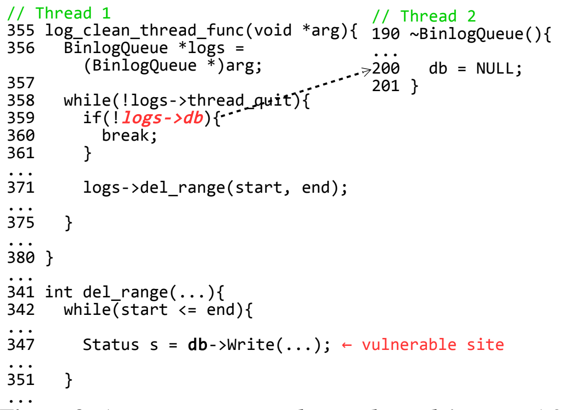
7.1 Detecting Known and New Concurrency Attacks

We applied XXX on 7 concurrency attacks listed in Table **??**. XXX detected all the vulnerabilities. Currently XXX incorporates two race detectors. There are other types of concurrency bugs that can also lead to concurrency attacks, including atomicity violations [47]. Atomicity violations can be detected by other detectors (e.g., CTrigger [53]). By integrating these detectors (future work), XXX's analysis and verifier components can detect more concurrency attacks.

Because all XXX's dynamic verifiers of are implemented based on LLDB, which only supports applications, we haven't run these verifiers in Linux kernel. Nevertheless, XXX's static vulnerability analyzer was applied to the Linux kernel and detected the evaluated concurrency attacks. For Linux kernel, our dynamic verifiers can be implemented in QEMU [55]. We leave the implementation in future work.

Aside from the discussed known and unknown concurrency attacks, XXX generates 180 reports in total. Due to the lack of domain knowledge and semantic understanding of program code, we didn't verify all of these potential vulnerability reports yet. These reports could either be benign races or new concurrency attacks. Nevertheless, by greatly reducing the number of reports from 31K to 180 (Table **??**, XXX has greatly mitigated developers' burdens.

XXX detected 3 previously unknown concurrency attacks caused by one new data race and two known data races. Analyzing whether known data races can lead to unknown concurrency attacks is still crucial (§**??**), because once attackers break in, they may remain latent for a long time.



```
// Thread 1
log_clean_thread_func(void *arg){
  BinlogQueue *logs =
    (BinlogQueue *)arg;

  while(!logs->thread_quit){
    if(!logs->db){
      break;
    }
...
    logs->del_range(start, end);
...
  }
...
}
...
int del_range(...){
  while(start <= end){
...
    Status s = db->Write(...); ← vulnerable site
...
  }
...
```

```
// Thread 2
~BinlogQueue(){
...
  db = NULL;
}
```

Figure 3: ***A new concurrency bug and attack in*** *SSDB****-1.9.2.***

Name	Vul. Type	Subtle Inputs	Bug Known	Attack Known
Apache-2.0.48	Double Free	PhP queries	Yes	Yes
Chrome-6.0.472.58	Use after free	Js console.profile	Yes	Yes
Libsafe-2.0-16	Buffer Overflow	Loops with strcpy()	Yes	Yes
Linux-2.6.10	Null Func Ptr Deref	Syscall parameters	Yes	Yes
Linux-2.6.29	Privilege Escalation	Syscall parameters	Yes	Yes
MySQL-5.0.27	Access Permission	FLUSH PRIVILEGES	Yes	Yes
MySQL-5.1.35	Double Free	SET PASSWORD	Yes	Yes
Linux-4.11.9(CVE)	Buffer Overflow	Three Syscalls	No	No
Apache-25520	Buffer Overflow	Loop	Yes	No

Table 1: XXX*'s detection results on concurrency attacks.* With the listed subtle inputs, all these attacks were often triggered within 20 repeated queries or loops except the Apache one.

XXX detected a new data race and a previously unknown use-after-free concurrency attack in SSDB. Figure 3 shows the details of this vulnerability. During server shutdown, SSDB uses adhoc synchronization to synchronize between threads. However, it's possible that line 359 is executed before line 200. This race causes log_clean_thread_fun to fail to break out of the while loop. Moreover, log_clean_thread_fun could execute del_range which could use db and cause a use after free. Even more, line 347 is a function pointer dereference which could cause log corruption or program crash if the memory area was reused by other threads.

XXX's static analyzer (§**??**) identified the vulnerability site at line 347 because it is a pointer dereference. This site is control dependent on the corrupted branch on line 359. XXX's dynamic vulnerability verifier (§**??**) further verified that the other thread will free the memory area and set the pointer to NULL before the dereference within current thread. We reported this race and attack to SSDB developers.

```
1327 ap_buffered_log_writer(void *handle, ...)
...
1334 {
1335   char *str;
1336   char *s;
1337   int i;
1338   apr_status_t rv;
1339   buffered_log *buf = (buffered_log*)handle;
...
1342   if (len + buf->outcnt > LOG_BUFSIZE) {
1343       flush_log(buf);
1344   }
...
1357   else {
1358    for(i=0,s=&buf->outbuf[buf->outcnt];i<nelts;++i) {
1359      memcpy(s,strs[i], trl[i]);← vulnerable site
1360      s += strl[i];
1361    }
1362    buf->outcnt += len;
1363    rv = APR_SUCCESS;
1364   }
...
1366 }
```

Figure 4: ***A new HTML integrity violation in*** *Apache****-2.0.48.***

The second previously unknown concurrency attack stems from a known data race in Apache. This attack made Apache's own request logs be written into other users' HTML files stored in Apache, causing a HTML integrity violation and information leak. Figure 4 shows the code of this vulnerability from the Apache-25520 bug [2]. buf->outcnt is shared among threads and serves as an index of a buffer array. Due to a lack of proper synchronization when modifying this variable on line 1362, a data race occurred and caused the server to write wrong contents to buf->outbuf.

Worse, the wrong contents could also overflow buf->outbuf and cause a buffer overflow. Even worse, Apache stores the file descriptor of its HTTP request log next to buf->outbuf. We constructed a one-byte overflow of buf->outbuf, corrupted this file descriptor, and made Apache's own HTTP request logs be written to an HTML file with the exact corrupted value of this file descriptor.

Although this data race has been well studied by researchers [47], people thought the worst consequence of this bug would just be corrupting Apache's own request log. We were the first to detect this HTML integrity violation attack with XXX and the first to construct the actual exploit scripts.

XXX's vulnerability analysis (§**??**) pinpointed the vulnerable site at line 1359 and inferred that this line is data dependent on the corrupted variable on line 1358. XXX's dynamic race verifier (§**??**) triggered the race and showed how many bytes in buf->outbuf were overflowed.

```
size_t busy;  /* busyness factor */

// Thread 1
588 static int proxy_balancer_post_request(…)
...
616 if (worker && worker->s->busy)
617   worker->s->busy--;
                          // Thread 2
                          616 if (worker && worker->s->busy)
                          617   worker->s->busy--;

    // Thread 3
    1138 static proxy_worker *find_best_bybusyness(...)
    ...
    1144      proxy_worker *mycandidate = NULL;
    ...
    1192 if (!mycandidate
    1193     || worker->s->busy < mycandidate->s->busy
    1194     || ...
    1195    mycandidate = worker; ← vulnerable site
```

Figure 5: ***A new integer overflow and DoS attack based on*** *Apache****-46215.***

The third previously unknown concurrency attack was an integer overflow DoS attack based on a known Apache-46215 data race. Figure 5 shows the Apache-46215 bug [3]. Each Apache worker thread contains a field worker->s->busy to indicate its busyness. An Apache load balancer component contains threads to concurrently increment or decrement these flags for worker threads when they start or finish serving requests. However, as shown in line 616, this is a data race because developers forgot to use a lock during the counter increment and decrement.

Over years, this busyness counter has been viewed a statistic and its data race does not matter much. Unfortunately, this counter is an unsigned integer, and an integer overflow could be triggered during the decrement and could make the counter the largest unsigned integer (i.e., marking a thread the "busiest" one). The check in line 617 can be easily bypassed because of the race. Because load balancer assigns future requests based on the worker threads' counters, arbitrary worker threads in `Apache` can be viewed the busiest ones and be completely ignored, causing a DoS attack on these threads and a significant downgrade of `Apache`'s throughput.

XXX detected this concurrency attack as follows. XXX's race detector detected a race between line 617 and line 1192. XXX's dynamic race verifier reported a detailed dynamic race information including the racing instructions, the value they could read or write to the variable and the type of the variable. We then found `worker->s->busy` in some worker threads had an overflowed value: `18,446,744,073,709,551,614`. XXX's vulnerability analysis (§**??**) reported that a pointer assignment could be control dependent on the corrupted branch of line 1192. XXX's vulnerability verifier verified that the branch was indeed corrupted and line 1195 was reachable.

These three previously unknown concurrency attacks were overlooked by prior reliability and security tools mainly due to three reasons. First, compared to XXX's reduced vulnerable reports, the data races of these three attacks were buried within at least 87X more false reports in `Apache` and 6X more in `SSDB` produced by the prior TSan race detector. Second, without XXX's static bug-to-attack propagation analysis (§**??**), even though the races can be detected by existing race detectors, the security consequences of these bugs were unknown to detectors. Third, without XXX's dynamic race verifier (§**??**) and vulnerability verifier (§**??**), whether these races and their attacks can be realized were unknown either.

7.2 Reducing False-positive Race Reports

Table **??** shows XXX's race report reduction results. The second column indicates the number of raw reports generated by our race detector. The third column shows how many adhoc synchronizations we found. The fourth column shows how many reports our dynamic race verifier had removed. The fifth column shows the number of the remaining reports.

Overall, XXX is able to prune 94% cases of false positives in Linux kernel and 97.7% for the other applications. This significant reduction will help developers save much diagnostic time. The performance of XXX's static analysis tool is critical because XXX aims to be scalable to large programs. The last column of Table **??** shows the average time cost of XXX's static analysis tool per bug report. Overall, except for Linux kernel and `Chrome`, XXX's analysis finished analyzing each program's bug reports within a few hours.

Name	LoC	# r.r.	# XXX's r.	# atks	# atks found
`Apache`	290K	715	10	3	3
`Chrome`	3.4M	1715	115	1	1
`Libsafe`	3.4K	3	3	1	1
`Linux`	2.8M	24641	34	2	2
`MySQL`	1.5M	1123	16	2	2
`SSDB`	67K	12	2	1	1
Total	-	-	-	-	-

Table 2: XXX *race report reduction and concurrency attack detection results.* Description

7.3 Find potential victim inputs

Name	Type	# Fed Inputs	# Victim Inputs	# atks
`Apache`	apr_palloc	-	-	-
`Linux`	kmalloc	180	6	1
Total	-	-	-	-

Table 3: XXX*'s Concurrency Fuzzer.* Description

8. Related Work

TOCTTOU attacks. Time-Of-Check-to-Time-Of-Use attacks [18, 67, 69, 73] target mainly the file interface, and leverage atomicity violation on time-of-check (access()) and time-of-use (open()) of a file to gain illegal file access.

A prior concurrency attack study [81] elaborates that concurrency attacks are much broader and more difficult to track than TOCTOU attacks for two main reasons. First, TOCTTOU mainly causes illegal file access, while concurrency attacks can cause a much broader range of security vulnerabilities, ranging from gaining root privileges [8] , injecting malicious code [7], to corrupting critical memory [2]. Second, concurrency attacks stem from miscellaneous kinds of memory access, while TOCTTOU stem from file accesses, thus handling concurrency attacks is much more difficult than TOCTTOU.

Another prior study [72] further defined two more kinds of vulnerabilities: One is **TOATTOU**(Time-Of-Audit-to-Time-Of-Use), in which the audit log diverges due to non-atomicity so that attackers could mask activities to avoid IDS triggering; The other is **TORTTOU**(Time-Of-Replace-to-Time-Of-Use), unique to system call wrappers, in which attackers could modify system call arguments after the wrapper has replaced the arguments but before the kernel accesses them.

Sequential security techniques. Defense techniques for sequential programs are well studied, including taint tracking [27, 49, 50, 56], anomaly detection [26, 60], address space randomization [63], and static analysis [17, 19, 29, 35, 70].

However, with the presence of multithreading, most existing sequential defense tools can be largely weakened or even completely bypassed [82]. For instance, concurrency bugs in global memory may corrupt metadata tags in metadata tracking techniques. Anomaly detection is lack of a concurrency model to reason about concurrency bugs and attacks.

Concurrency reliability tools. Various prior systems work on concurrency bug detection [28, 39, 40, 46, 48, 59, 75, 85–87], diagnosis [16, 38, 40, 52, 53, 61], and correction

[36, 37, 71, 77]. They focus on concurrency bugs themselves, while OWL focuses on security related consequences of concurrency bugs. Therefore, these systems are complementary to OWL.

Conseq [86] detects harmful concurrency bugs by analyzing their failure consequence. Its key observation is that concurrency bugs and those bugs' failure sites are usually within a short control and data flow propagation distance (e.g., within the same function). Concurrency attacks(targets of OWL) usually exploit corrupted memory that resides in different functions, thus Conseq is inadequate for concurrency attacks. Though, Conseqs proactive harmful schedule exploration technique will be useful for OWL to trigger more vulnerable schedules.

Static & Dynamic vulnerability detection tools. There are already a variety of static and dynamic vulnerability detection approaches [31, 31, 44, 51, 64, 80, 88], which fall into two categories based on whether they target general or specific programs.

The first category [44, 80] targets general programs and these approaches have been shown to find severe vulnerabilities in large code. However, pure static or dynamic analyses may not be adequate to cope with concurrency attacks. Benjamin et al. [44] leverage pointer analysis to detect data flows from unchecked inputs to sensitive sites. This approach ignores control flow and thus it is not suitable to track concurrency attacks like the Libsafe one in **[TBC - 4.3]**. Yamaguchi et al. [80] did not incorporate inter-procedural analysis and thus is not suitable to track concurrency attacks either. Moreover, these general approaches are not designed to reason about concurrent behaviors (e.g., [80] can not detect data races).

OWL belongs to the first category because it targets general programs. Unlike the prior approaches in this category, OWL incorporates concurrency bug detectors to reason about concurrent behaviors, and OWLs consequence analyzer integrates critical dynamic information (i.e., call stacks) into static analysis to enable comprehensive data-flow, control- flow, and inter-procedural analysis features.

The second category [15, 31, 51, 54, 64, 88] makes analysis focused on specific behaviors (e.g., APIs) in specific programs to achieve scalability and accuracy. These approaches check web application logic [31] and interaction among scripts [51], Android applications [15], cross checking security APIs [64], and Linux Security Module [88]. Pure Dynamic approaches could reason about s specific execution path, but only the covered ones during execution observation. And they could not give semantic information upon the specific programs they are targeting [51]. OWLs analysis is complementary to these approaches; OWL can be further integrated with these approaches to track concurrency attacks.

Scheduler Control and Symbolic execution. Scheduler control is a way to exploiting synchronization bugs by using interrupting and scheduling threads. AsyncShock [74] is such a tool for thread manipulation. Scheduler control can be utilized by OWL to find thread execution order of triggering concurrency attacks. Symbolic execution is an advanced program analysis technique that can systematically explore a programs execution paths to find bugs. Researchers have built scalable and effective symbolic execution systems to detect software bugs [19–21, 24, 32–34, 57, 62, 84], block malicious inputs [58], preserve privacy in error reports [22], and detect programming rule violations [25]. Specifically, UCKLEE [57] has been shown to effectively detect hundreds of security vulnerabilities in widely used programs. Symbolic execution is orthogonal to OWL; it can augment OWLs input hints by automatically generating concrete vulnerable inputs.

9. Conclusion

References

[1] All concurrency vulnerabilities studied. `http://vivace.cs.columbia.edu/bugzilla3/`.

[2] Apache bug 25520. `https://bz.apache.org/bugzilla/show_bug.cgi?id=25520`.

[3] Apache bug 46215. `https://bz.apache.org/bugzilla/show_bug.cgi?id=46215`.

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

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

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

[7] Freebsd CVE-2009-3527. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3527`.

[8] Linux kernel bug on uselib(). `http://osvdb.org/show/osvdb/12791`.

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

[10] Mysql bug 14747. `https://bugs.mysql.com/bug.php?id=14747`.

[11] Mysql bug 24988. `https://bugs.mysql.com/bug.php?id=24988`.

[12] PCWorld. `http://www.pcworld.com/businesscenter/article/255911/`.

[13] MySQL Database. `http://www.mysql.com/`, 2014.

[14] Apache web server. `http://www.apache.org`, 2012.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[16] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, volume 12, pages 307–320, 2012.

[17] 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, Feb. 2010.

[18] M. Bishop, M. Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[19] 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.

[20] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, pages 322–335, Oct.–Nov. 2006.

[21] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st Symposium on Cloud Computing (SOCC '10)*, 2010.

[22] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 319–328, Mar. 2008.

[23] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[24] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *Fifth Workshop on Hot Topics in System Dependability (HotDep '09)*, 2009.

[25] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013.

[26] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, pages 1–10, Mar. 1990.

[27] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–6, 2010.

[28] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, Oct. 2003.

[29] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Invited paper: Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI04)*, pages 191–210, Jan. 2004.

[30] 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.

[31] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[32] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 206–215, 2008.

[33] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, June 2005.

[34] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of 15th Network and Distributed System Security Symposium*, Feb. 2008.

[35] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, June 2002.

[36] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, volume 12, pages 221–236, 2012.

[37] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, 2008.

[38] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[39] B. Kasikci, C. Zamfir, and G. Candea. Racemob: crowdsourced data race detection. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 406–422. ACM, 2013.

[40] B. C. C. Kasikci. Techniques for detection, root cause diagnosis, and classification of in-production concurrency bugs. 2015.

[41] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, 2007.

[42] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[43] Libsafe. `http://directory.fsf.org/wiki/Libsafe`.

[44] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[45] The LLVM compiler framework. `http://llvm.org`, 2013.

[46] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[47] 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)*, pages 329–339, Mar. 2008.

[48] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[49] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 129–142, 1997.

[50] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[51] R. Paleari, D. Marrone, D. Bruschi, and M. Monga. On race vulnerabilities in web applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 126–142. Springer, 2008.

[52] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, Nov. 2008.

[53] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[54] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):3, 2011.

[55] `http://www.qemu.org`.

[56] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[57] D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium*, pages 49–64, 2015.

[58] C. Sapuntzakis. Personal communication. Bug in OpenBSD where an interrupt context could call blocking memory allocator, Apr. 2000.

[59] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, Nov. 1997.

[60] D. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, 1989.

[61] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[62] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 263–272, Sept. 2005.

[63] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, Proceedings of the 11th ACM conference on Computer and communications security (CCS '04), pages 298–307, 2004.

[64] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple api implementations. *ACM SIGPLAN Notices*, 46(6):343–354, 2011.

[65] `ssdb.io/`.

[66] P. B. Todd Warszawski. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pages 5–20. ACM, 2017.

[67] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file tocttou races with hardness amplification. In *FAST*, volume 8, pages 1–18, 2008.

[68] Threadsanitizer. `https://code.google.com/p/data-race-test/wiki/ThreadSanitizer`, 2015.

[69] E. Tsyrklevich and B. Yee. *Dynamic detection and prevention of race conditions in file accesses*. PhD thesis, University of California, San Diego, 2003.

[70] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P '01)*, 2001.

[71] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Dec. 2008.

[72] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX workshop on Offensive Technologies*, pages 2:1–2:8, 2007.

[73] J. Wei and C. Pu. Tocttou vulnerabilities in unix-style file systems: An anatomical study. In *FAST*, volume 5, pages 12–12, 2005.

[74] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[75] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy. Parallelizing data race detection. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 27–38, Mar. 2013.

[76] J. Whaley. bddbddb Project. `http://bddbddb.sourceforge.net`.

[77] 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.

[78] Z. Wu, K. Lu, X. Wang, and X. Zhou. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming*, 43(2):260–285, 2015.

[79] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[80] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[81] J. Yang. Concurrency attacks and defenses. Technical report, The Trustees of Columbia University in the City of New York DUNS 049179401 New York United States, 2016.

[82] J. Yang, A. Cui, J. Gallagher, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. Technical Report CUCS-028-11, Columbia University, 2011.

[83] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)*, June 2012.

[84] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P '06)*, pages 243–257, May 2006.

[85] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, Oct. 2005.

[86] 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, Mar. 2011.

[87] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, Mar. 2010.

[88] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Aug. 2002.