Understanding and Detecting Concurrency Attacks

Paper #82

The University of Hong Kong
@cs.hku.hk

Abstract

1. Introduction

2. Background

A prior study [57] browsed the bug databases of 46 real-world concurrency bugs and made three major findings on concurrency attacks. First, concurrency attacks are severe threats: 35 of the bugs can corrupt critical memory and cause three types of violations, including privilege escalations [5**?**], malicious code injections [6], and bypassing security authentications [2–4].

Second, concurrency bugs and attacks can often be easily triggered via subtle program inputs. For instance, attackers can use inputs to control the physical timings of disk IO and program loops and trigger concurrency bugs with a small number of re-executions. Third, compared to traditional TOCTOU attacks, which stem from corrupted file accesses, handling concurrency attacks is much more difficult because they stem from corrupted, miscellaneous memory accesses.

These three findings reveal that concurrency attacks can largely weaken or even bypass most existing security defense tools, because these tools are mainly designed for sequential attacks. For instance, consider taint tracking tools, concurrency attacks can corrupt the metadata fields in these tools and completely bypass taint tracking. Anomaly detection tools, which rely on inferring adversary program behaviors (e.g., excessive re-executions), become ineffective, because concurrency attacks can easily manifest via subtle inputs.

This prior study raises an open research question: *what will be an effective tool for detecting concurrency attacks?* Specifically, can existing concurrency bugs detection tools effectively detect these bugs and their attacks? The answer is probably "NO" because literature has overlooked these attacks.

3. Quantitative Concurrency Attack Study

We studied concurrency attacks in 10 widely used programs, including 3 servers, 2 browsers, 1 library, and 4 kernel distributions. We added the shared memory concurrency bugs in the prior study [57], and we searched "concurrency bug vulnerability" in CVE and these programs' bug databases. We manually inspected bug reports and removed them if they were false reports or lack a clear description, and we conservatively kept the vulnerable ones caused by multithreading.

Unlike the prior study [57] which counted the number of security consequences in bug reports as the number of concurrency attacks, we counted only each bug's first security consequence. In total we collected 26 concurrency attacks with three more types of violations than the prior study [57], including HTML integrity violations (§**??**), buffer overflows (§**??**), and DoS attacks (§**??**). We built scripts to successfully exploit 10 attacks in 6 programs if we had source code.

To quantitatively analyze why concurrency attacks are overlooked, we considered data race detectors because they have effectively found concurrency bugs. We selected two popular tools: TSAN [**?**] for applications and SKI [**?**] for OS kernels. We ran the two tools on 6 programs that support these tools. We used the programs' common performance benchmarks as workloads. Table 1 shows a study summary.

Name	LoC	# Concurrency attacks	# Race reports
Apache	290K	4	715
MySQL	1.5M	2	1123
SSDB	67K	1	12
Chrome	3.4M	3	1715
IE	N/A	1	N/A
Libsafe	3.4K	1	3
Linux	2.8M	8	24641
Darwin	N/A	3	N/A
FreeBSD	680K	2	N/A
Windows	N/A	1	N/A
Total	8.0M	26	28209

Table 1: ***Concurrency attacks study results.*** This table contains both known and previously unknown concurrency attacks we detected. We made 6 out of 10 programs run with race detectors. We built exploit scripts for 10 concurrency attacks in these 6 programs.

3.1 Challenging Findings

I: Concurrency attacks are much more severe than concurrency bugs. Every studied program has concurrency attacks. Figure 1 shows a concurrency attack that bypassed stack overflow checks in the `Libsafe` [30] library and injected malicious code. Figure 2 shows a concurrency attack in the Linux `uselib()` system call. Attackers have leveraged this bug to trigger a NULL pointer dereference in the kernel and execute arbitrary code from user space.

One key difference between concurrency attacks and concurrency bugs is that fixing the buggy code is not sufficient to fix the vulnerabilities. For instance, once attackers have got OS root privileges [5**?**], they may stay forever in the system. Therefore, it's still crucial to study whether existing known concurrency bugs may lead to concurrency attacks.

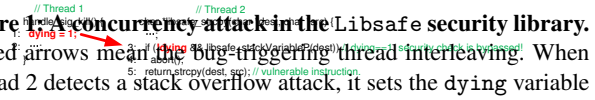

Figure 1: A concurrency attack in the `Libsafe` security library. Dotted arrows mean the bug-triggering thread interleaving. When Thread 2 detects a stack overflow attack, it sets the `dying` variable to 1 and kills current process shortly. However, access to `dying` is not correctly protected by mutex, so Thread 1 reads this variable, bypasses the security check in `stack_check()` (called at line 164), and runs into a stack overflow in `strcpy()` (at line 165).

```
// Thread msync                          // Thread uselib
static int msync_interval(...) {         int do_munmap(...) {
...                                      ...
  struct file * file = vma->vm_file;
...                                        file->f_op = NULL;
  if(file->f_op&&file->f_op->fsync){
                                         ...
    err = file->f_op->fsync(...);        }
  }
}
                  vulnerable site
```

Figure 2: A concurrency attack in the Linux `uselib()` and `msync()` system calls. Dotted arrows mean the bug-triggering thread interleaving. A data race on the `f_op` struct causes the Linux kernel to trigger a NULL pointer dereference and enables arbitrary code execution.

II: Concurrency bugs and their attacks are widely spread in program code. Among 10 attacks we had source code and constructed exploit scripts, 7 have their bugs and vulnerability sites among different functions. Moreover, bugs often affect vulnerability sites not only through data flows but also control flows (e.g., the `Libsafe` attack in Figure 1).

This finding suggests that a concurrency attack detection tool should incorporate both inter-procedural and control-flow analyses. Unfortunately, to scale to large programs, existing bug consequence analysis tools (e.g., [31, 60**?**]) lack either inter-procedural or control-flow analysis.

III: Concurrency bugs and their attacks are often triggered by separate, subtle program inputs. Consider the inputs to trigger concurrency bugs, unlike previous understanding [16, 40] that triggering concurrency bugs require intensive repeated executions, 8 out of the 10 reproduced concurrency attacks in our study can be easily triggered with less than 20 repetitive executions on our evaluation machines with carefully chosen program inputs. For instance, in a `MySQL` privilege escalation [**?**], we used the "`flush privileges;`" query to trigger a data race and corrupted another `MySQL` user's privilege table with only 18 repeated executions.

In addition to input values, carefully crafted input timings can also expand the *vulnerable window* [57] which increases the chance of running into the bug-triggering schedules. For instance, consider Figure 2, since the `if` statement and the `file->f_op->fsync()` statement in `msync_interval()` have an IO operation (not shown) in between, attackers could craft inputs with subtle timings for this IO operation and thus enlarged the time window of these two statements. Then, attackers could easily trigger the buggy schedule in Figure 2.

In addition to the inputs for triggering concurrency bugs, triggering the attacks of these bugs often require other subtle program inputs. A main reason is the bugs and their attacks are widely spread in program code and thus they may easily be affected by different inputs. In a Linux `uselib()` data race [5], we needed to carefully construct kernel swap IO operations to trigger the race, and we needed to call extra system calls to get a root shell out of this race. By constructing subtle inputs for both the bug and its attack, we needed only tens of repeated executions to get this root shell on our evaluation machines.

This `uselib()` attack reveals two issues. First, a small number of repeated executions indicates that attackers can easily bypass anomaly detection tools [17, 43**?**] with subtle inputs. Second, existing data race detectors are ineffective at revealing this attack because they will stop after they run a bug-triggering input and flag this race report. Such a one-shot detection will overlook a concurrency attack as it often requires extra inputs to trigger the attack. Therefore, extra analysis is required to identify the bug-to-attack propagation.

IV: Most concurrency bugs that triggered concurrency attacks can be detected by race detection tools. There are several types of concurrency bugs, including data race, atomicity violation, and order violation [34]. Although some types of concurrency bugs are difficult to detect (e.g., order violation), we found that all concurrency bugs we studied were data races and these bugs can readily be detected by TSAN or SKI. This finding suggests that a race detector is a necessary component for detecting concurrency attacks.

V: Concurrency attacks are overlooked mainly due to the excessive reports from race detectors. We identified two major reasons for this finding. First, existing race detectors generate too many bug reports which deeply bury the vulnerable ones. For instance, we ran `MySQL` with TSAN and repeatedly generated the same bug-triggering SQL query [**?**]. We got 202 race reports, but after our manual inspection, only two reports will lead to attacks. Table 1 shows more programs with even more reports. These excessive reports make finding concurrency attacks from the reports just like "finding needles in a haystack."

Second, even if a developer luckily opens a true bug report that can actually lead to an attack, she still has no clue whether what attacks the bug may lead to, because the report only shows the bug itself (e.g., the corrupted variable), but not its security consequences. Therefore, it's crucial to have an analysis tool that can accurately identify the bug-to-attack propagation for bug reports.

3.2 Optimistic Findings

To assist the construction of a practical concurrency attack detection tool, we identified two common patterns for concurrency attacks. First, although the consequences of concurrency attacks are miscellaneous, these consequences are triggered by five explicit types of vulnerable sites, including memory operations (e.g., `strcpy()`), NULL pointer deferences, privilege operations (e.g., `setuid()`), file operations (e.g., `access()`), and process-forking operations (e.g., `eval()` in shell scripts). Our study found that these vulnerable sites have independent consequences to each other, thus more types can be easily added.

Second, concurrency bugs and their attacks often share similar call stack prefixes. From the 10 concurrency attacks with source code, 7 of them have the vulnerability site in the callees (i.e., the call stack of the bug is a prefix of the call stack of the vulnerability site). For the rest them, the vulnerability site is just one or two levels up of the bug's call stack. These two patterns reveal an opportunity to build a precise, scalable static analysis tool for tracking the bug-to-attack propagation.

4. OWL Overview

5. Reducing Benign Schedules

This section presents OWL's benign schedule reduction component, including automatically annotating adhoc synchronizations (§5.1) and pruning benign schedules (§5.2). This component in total greatly reduced 94.3% of the total reports (see §**??**).

5.1 Annotating Adhoc Synchronization

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.

OWL 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 OWL 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, OWL tags this report as an "adhoc sync".

Compared to the prior static adhoc sync identification method SyncFinder [56], 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.

5.2 Verifying Real Data Races

OWL'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, OWL'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. OWL'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 [**?**], OWL's verifier achieves the goal without requiring heavyweight Java instrumentation. Compared with ConcurrentBreakpoint [**?**] and ConcurrentPredicate [**?**], we require no code annotations and importing libraries.

Overall, OWL'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 OWL'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 [**?**].

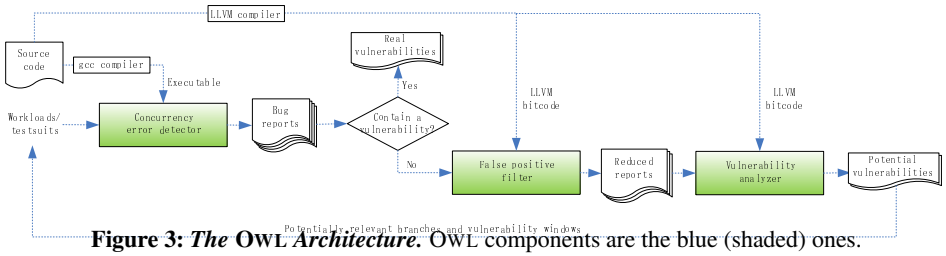

Figure 3: *The* OWL *Architecture.* OWL components are the blue (shaded) ones.

6. Computing Vulnerable Input Hints

This section presents the algorithm of OWL's static vulnerability analysis (§6.1) and dynamic verifier (§6.2). Since the input of the static analysis is the reports from concurrency bug detectors, this section then describes how OWL integrates this analysis with two existing race detectors (§6.3).

6.1 Analysis Algorithm

Algorithm 1 show OWL'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. The algorithm then does inter-procedural static analysis to see whether corrupted memory may propagate to any vulnerable site (§3.2) 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 (§3.2) 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 [32]. Similar to relevant systems [60**?**], our design did not incorporate pointer analysis [29, 54] because one main issue of such analysis is that it typically reports too many false positives on shared memory access in large programs (§7.1).

Our analyzer compensates the lack of pointer analysis by: (1) tracking read instructions in the detectors at runtime (§6.3), 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 (§6.3).

All five types of vulnerability sites we found (§3.2) 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 (§**??**).

6.2 Dynamic Vulnerability Verifier

OWL'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.

6.3 Integration with Concurrency Bug Detectors

OWL has integrated two popular race detectors: SKI for Linux kernels and TSAN for application programs. To integrate OWL's algorithm (§6.1) 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

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()
```

variables. The final race report will show all the stacks of the reading thread.

Another issue for OWL 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.

7. Discussions

This section discusses OWL's limitations (§7.1) and its broad applications (§7.2).

7.1 Limitations

OWL's main design goal is to achieve reasonable accuracy and scalability, and it trades off soundness (i.e., do not miss any bugs), although OWL did not missed the evaluated attacks (§**??**). A typical way to ensure soundness is to plug in a sound alias analysis tool (e.g., [29, 54]) to identify all LLVM `load` and `store` instructions that may access the same memory. However, typical alias analyses are known to be inaccurate (e.g., too many false positives).

OWL currently handles five types of regular vulnerability operations (§3.2), and it requires these operations to exist in the LLVM bitcode. These five types of operations are sufficient to cover all 10 concurrency attacks we have reproduced, and more types can be added. If developers are concerned about some library code that may contain vulnerabilities, they should compile this code into the bitcode for OWL.

OWL's consequence analysis tool integrates the call stack of a concurrency bug to direct static analysis toward vulnerable program paths, but OWL's vulnerable input hints (§6.1) may contain false reports (e.g., the outcomes in OWL's collected bug-to-attack propagation branches may have conflicts). Developers can inspect the propagation chains and refine their program inputs to validate the outcomes. In our evaluation, we found these input hints expressive as they helped us identify subtle inputs for real attacks (§**??**).

7.2 OWL has Broad Applications

We envision two immediate applications for OWL's techniques. First, OWL can augment existing defense tools on concurrency attacks. For instance, we can leverage anomaly detection [17, 43**?**] and intrusion detection [25, 49, 50] tools to audit only the vulnerable program paths identified by OWL, then these runtime detection tools can greatly reduce the amount of program paths that need to be audited and improve performance. OWL can also integrate with other bug detection tools (e.g., process races [28] and atomicity bugs [39]) to detect concurrency attacks caused by such bugs.

Second, OWL's consequence analysis tool has the potential to detect various consequences of software bugs. Software bugs have caused many extreme disasters [**? ?**] in the last few decades, including losing big money and taking lives. By adding new vulnerability and failure sites of such consequences, OWL can be applied to flagging bugs that can cause severe consequences among enormous bug reports.

8. Evaluation

9. Related Work

TOCTOU attacks. Time-Of-Check-to-Time-Of-Use attacks [8, 47, 48, 52] 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 [57] elaborates that concurrency attacks are much broader and more difficult to track than TOCTOU attacks for two main reasons. First, TOCTOU mainly causes illegal file access, while concurrency attacks can cause a much broader range of security vulnerabilities, ranging from gaining root privileges [5], injecting malicious code [**?**], to corrupting critical memory [1]. Second, concurrency attacks stem from miscellaneous memory accesses, and TOCTOU stem from file accesses, thus handling concurrency attacks is much more difficult than TOCTOU.

Sequential security techniques. Defense techniques for sequential programs are well studied, including taint tracking [18, 36, 37, 41], anomaly detection [17, 43], address space randomization [46], and static analysis [7, 9, 20, 24, 49].

However, with the presence of multithreading, most existing sequential defense tools can be largely weakened or even completely bypassed [57]. For instance, concurrency bugs in global memory may corrupt metadata tags in metadata tracking techniques. Anomaly detection lacks a concurrency model to reason about concurrency bugs and attacks.

Concurrency reliability tools. Various prior systems work on concurrency bug detection [19, 33, 35, 42, 53, 59**?** –61], diagnosis [38, 39, 44**? ?**], and correction [26, 27, 51, 55]. They focused on concurrency bugs themselves, while OWL focuses on the security consequences of concurrency bugs. Therefore, these systems are complementary to OWL.

Conseq [60] detects harmful concurrency bugs by analyzing its failure consequence. Its key observation is that concurrency bugs and the bugs' failure sites are usually within a short control and data flow propagation distance (e.g., within the same function). Concurrency attacks targeted in OWL usually exploit corrupted memory that resides in different functions, thus Conseq is inadequate for concurrency attacks. Conseq's proactive harmful schedule exploration technique will be useful for OWL to trigger more vulnerable schedules.

Static vulnerability detection tools. There are already a variety of static vulnerability detection approaches [31, 62**? ? ? ?**]. These approaches fall into two categories based on whether they target general or specific programs.

The first category [31**?**] targets general programs and their approaches have been shown to find severe vulnerabilities in large code. However, these pure static analyses may not be adequate to cope with concurrency attacks. Benjamin et al. [31] leverages 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 §**??**. Yamaguchi et al. [**?**] 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., [**?**] 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 OWL's 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 [62**? ? ?**] lets static analysis focus on specific behaviors (e.g., APIs) in specific programs to achieve scalability and accuracy. These approaches check web application logic [**?**], Android applications [**?**], cross checking security APIs [**?**], and verifying the Linux Security Module [62]. OWL's analysis is complementary to these approaches; OWL can be further integrated with these approaches to track concurrency attacks.

Symbolic execution. Symbolic execution is an advanced program analysis technique that can systematically explore a program's execution paths to find bugs. Researchers have built scalable and effective symbolic execution systems to detect software bugs [9–11, 13, 21–23, 45, 58**?**], block malicious inputs [14], preserve privacy in error reports [12], and detect programming rule violations [15]. Specifically, UCKLEE [**?**] has been shown to effectively detect hundreds of security vulnerabilities in widely used programs. Symbolic execution is orthogonal to OWL; it can augment OWL's input hints by automatically generating concrete vulnerable inputs.

10. Conclusion

We have presented the first quantitative study on real-world concurrency attacks and OWL, the first analysis framework to effectively detect them. OWL accurately detect a number of known and previously unknown concurrency attacks on large, widely used programs. We believe that our study will attract much more attention to further detect and defend against concurrency attacks. Our OWL framework has the potential to bridge the gap between concurrency bugs and their attacks. All our study results, exploit scripts, and OWL source code with raw evaluation results are available at `github.com/asplos17-paper82/owl`.

References

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

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

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

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

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

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

[7] 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.

[8] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 117–130, Oct. 2007.

[15] 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.

[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] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[26] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 221–236, 2012.

[27] H. Jula, D. Tralamazza, Z. Cristian, and C. George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Dec. 2008.

[28] O. Laadan, N. Viennot, C. che Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[29] 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.

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

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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, Oct. 1997.

[37] 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.

[38] 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.

[39] 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.

[40] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, Oct. 2009.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file tocttou races with hardness amplification. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 13:1–13:18, 2008.

[48] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 17–17, 2003.

[49] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, 2001.

[50] 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.

[51] 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.

[52] J. Wei and C. Pu. Tocttou vulnerabilities in unix-style file systems: an anatomical study. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 12–12, 2005.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.