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.

1. Introduction

Multi-threaded program is hard to be correct. Concurrency bugs are common in modern multi-threaded programs including atomic violation, ..., and especially data race[40, 41, 76, 77]. 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 [40, 44, 59, 68, 76, 77].

Recent studies[57, 73] show rise of concerns about *concurrency attacks*. By triggering concurrency bugs, hackers may leverage the corrupted memory to conduct attacks including privilege escalations[7, 9], hijacking code execution[**?**], bypassing security checks[**? ? ?**], and breaking database integrity[57]. These vulnerabilities are often hidden in concurrency bugs and implicit regarding program behaviors. For example, a privilege escalation attack may cause no outrageous effect, but possess a permanent security hole hiding in the system. Also, despite the 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 and practical tools for understanding and detecting concurrency attacks. ???Our study over known concurrency bugs[1, 2] shows that even concurrency bugs have been successfully detected and reported, professionals may still lack knowledge about how severe consequences these bugs may cause. ???For instance, *apache-25520*[1] has been reported over years and well studied by researchers[41]. ???We are the first to exploit a new heap overflow attack leveraging on this bug and break the HTML integrity. Knowledge and detection of concurrency attacks is of crucial importance.

We studied 26 attacks and find two major challenges for detecting concurrency attacks. First, concurrency attacks may be implicit and hidden in common concurrency bugs. Existing concurrent bug detectors may not analyze whether a concurrency bug is vulnerable and exploitable. ConMem[77] first propose to consider concurrency bugs that may cause severe consequences (e.g.program crash), but our observation shows that explicit error (e.g.program crash) is not necessary for concurrency attacks. ConSeq[76] care about severe consequences and do intra-procedure analysis to help diagnose concurrency bugs. However, propagation path ... . Worse still, crash bugs may even lead to more severe vulnerabilities. In *CVE-2017-7533* conducted by our team, although the data race primarily causes kernel crash, we crafted the input and successfully conduct a privilege escalation attack without crashing the kernel. Current concurrency bug detecting tools are not designed to analyze this kind of latent vulnerabilities, and hence may direct wrong level of warnings towards the bugs.

Second, extant work ignores indicating extra inputs to conduct concurrency attacks. A concurrency bug may become much more vulnerable when attackers 1.craft inputs that trigger the bug; 2.employee another input running on victim threads to construct their attacks. In CVE-2017-7533, we do not only leverage two crafted inputs running on two threads to trigger a data race and construct kernel heap overflow, but 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 and one input

Leveraging this 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 analyze bug-to-attack propagations. We easily employ existing race detectors, and design a benign schedule reducer to reduce race reports(§**??**). Then we

The second phase is *concurrency fuzzer?* to ... .

We implemented XXX on Linux, supporting both user space and kernel space attack detection.

We evaluated OWL on 6 diverse, widely used programs, including Apache, Chrome, Libsafe, Linux kernel, MySQL, and SSDB. OWLs 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, OWLs vulnerable input hints helped us identify subtle vulnerable inputs, leading to the detection of 7 known concurrency attacks as well as 3 previous unknown, severe ones in SSDB and Apache. The analysis performance of OWL 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 tool 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. §**??**...

2. Background

2.1 Concurrency Bug

Concurrency bugs in multi-threaded program is common and caused great loss in real world. Non-deadlock concurrency bugs atomicity violations[], order violations[] and others by extant studies[41, 77]. A extant work exclude data race as concurrency bug pattern because data race does not certainly cause a concurrency bug[41] and many data races are benign races. However, our observation shows that data race is necessary for the attacks we studied, although the bugs can be categorized to atomicity violation or order violation. And also, current race detectors can well readily detect the bug happening that causes attacks.

??? Here why data race detector? How about atomic violation detector? Order violation detector?

2.2 Concurrency Attack

Extant studies [57, 73] 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 con-

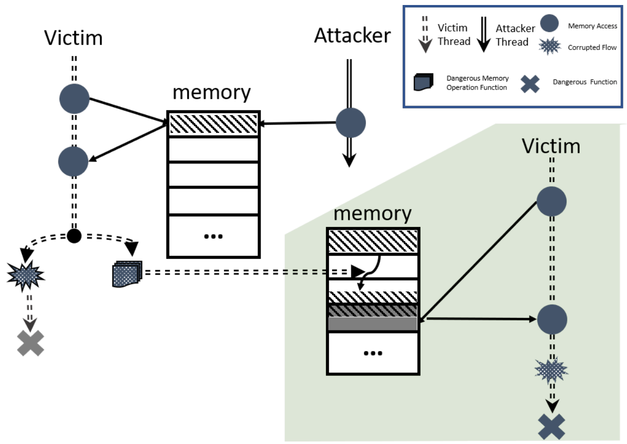

Figure 1: ***Concurrency Attack Model***

sequences, including privilege escalations[7, 10], malicious code injections [8], bypassing security authentications[3–5] , and breaking database integrity[57]. Some concurrency bugs Current concurrency bugs

Second, concurrency bugs and attacks can often be easily triggered crafted program inputs.

More ever, attack - instruction

2.3 Bug-to-Attack 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 gives an overview of the model and architecture of XXX.

3.1 Preliminary

This subsection gives definitions of items we use in this paper.

Input

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

3.4 Detecting Example

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 in-

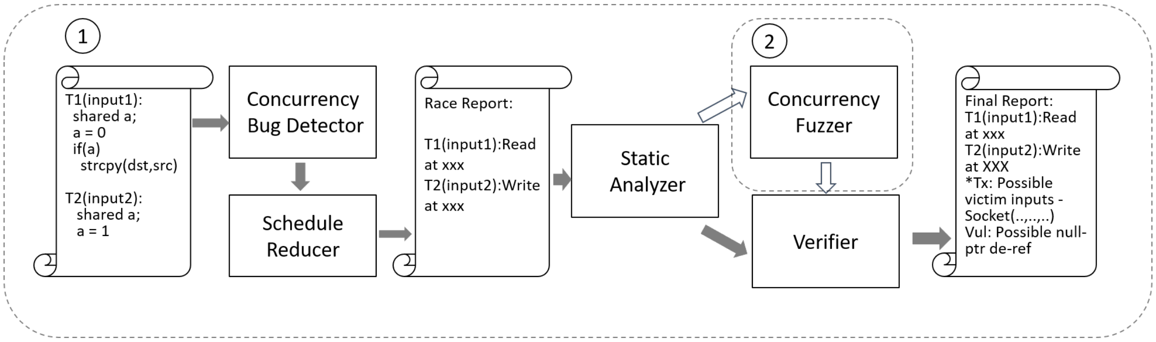

Figure 2: ***Architecture***

tegrate 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 [69], 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 Concurrent-

Predicate [**?**], 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. 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 [39]. Similar to relevant systems [76**?**], our design did not incorporate pointer analysis [36, 66] 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).

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()
```

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` [12] web server, `MySQL` [11] database server, and SSDB [56] key-value store server), one library (`Libsafe` [37]), 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

7.2 Reducing False-positive Race Reports

7.3 Find potential victim inputs

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

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

Table 2: XXX*'s Concurrency Fuzzer.* Description

8. Related Work

TOCTTOU attacks. Time-Of-Check-to-Time-Of-Use attacks [16, 58, 60, 64] 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 [71] elaborates that concurrency attacks are much broader and more difficult to track than TOCTTOU 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 [7] , injecting malicious code [6], to corrupting critical memory [1]. Second, concurrency attacks stem from miscellaneous kinds of memory access, and TOCTTOU stem from file accesses, thus handling concurrency attacks is much more difficult than TOCTTOU.

Another prior study [63] further defined two more kinds of vulnerabilities: One is **TOATTOU**(Time-Of-Check-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-Check-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 [24, 43, 44, 47], anomaly detection [23, 51], address space randomization [54], and static analysis [15, 17, 26, 31, 61].

However, with the presence of multithreading, most existing sequential defense tools can be largely weakened or even completely bypassed [72]. 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 [25, 35, 40, 42, 50, 65, 75–77], diagnosis [14, 34, 45, 46, 52], and correction [32, 33, 62, 67]. They focus on concurrency bugs themselves, while OWL focuses on security related consequences of concurrency bugs. Therefore, these systems are complementary to OWL.

Conseq [76] 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 vulnerability detection tools. There are already a variety of static vulnerability detection approaches [27, 27, 38, 55, 70, 78], which fall into two categories based on whether they target general or specific programs.

The first category [38, 70] targets general programs and these 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. [38] 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 4.3**[TOCHECK]**. Yamaguchi et al. [70] 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., [70] 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 [13, 27, 55, 78] makes static analysis focused on specific behaviors (e.g., APIs) in specific programs to achieve scalability and accuracy. These approaches check web application logic [27], Android applications [13], cross checking security APIs [55], and Linux Security Module [78]. OWLs 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 programs execution paths to find bugs. Researchers have built scalable and effective symbolic execution systems to detect software bugs [17–19, 21, 28–30, 48, 53, 74], block malicious inputs [49], preserve privacy in error reports [20], and detect programming rule violations [22]. Specifically, UCKLEE [48] 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] Apache bug 25520. `https://bz.apache.org/bugzilla/show_bug.cgi?id=25520`.

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] `ssdb.io/`.

[57] 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.

[58] 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.

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

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

[61] 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.

[62] 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.

[63] 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.

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

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.