Understanding and Detecting Concurrency Attacks

Rui Gu*, Bo Gan*, Jason Zhao*, Yi Ning+, Heming Cui+, and Junfeng Yang*

*Columbia University +The University of Hong Kong

Abstract

Just like bugs in single-threaded programs can lead to vulnerabilities, bugs in multithreaded programs can also lead to concurrency attacks. Unfortunately, there is little quantitative data on how well existing tools can detect these attacks. This paper presents the first quantitative study on concurrency attacks and their implications on tools. 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 portions of inputs and thread interleavings (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 likely lead to attacks and what will not (e.g., benign bug reports). With this insight, this paper presents a new directed concurrency attack detection approach and its implementation, OWL. It extracts hints from the reports with static analysis, augments existing detectors by pruning out the benign inputs and schedules, and then directs detectors and its own runtime vulnerability verifiers to work on the remaining, likely vulnerable inputs and schedules.

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

Multithreaded programs are already prevalent. However, despite much effort, these programs are still notoriously difficult to get right. Concurrency bugs (i.e., shared memory accesses without proper synchronization among threads) in these programs have led to severe consequences, including memory corruption, wrong outputs, and program crashes [52].

Worse, a prior study [85] shows that many real-world concurrency bugs can lead to *concurrency attacks*: once a concurrency bug is triggered, attackers can leverage the memory corrupted by this bug to construct various violations, including privilege escalations [7, 10], malicious code injections [8], and bypassing security authentications [4, 3, 5]. For instance, a data race [7] in the Linux kernel corrupted the kernel's memory management subsystem and led to a root privilege escalation. This study also elaborates that, because concurrency attacks are caused by concurrent, miscellaneous memory accesses, they can weaken most traditional defense techniques (e.g., TOCTOU [74, 79, 73]). However, this study did not provide exploit scripts to trigger these attacks, nor it quantitatively evaluated existing tools on these attacks.

This paper presents the first quantitative study on the severity of concurrency attacks. We studied 10 widely used programs, including 3 server programs, 2 browsers, 1 library, and 4 kernel distributions, in CVE and their own bug databases. Our study reveals 26 concurrency attacks that consist of three more types of violations than the prior study [85], including HTML integrity violations (§8.4), buffer overflows (§4.3), and DoS attacks (§8.4). We built scripts to successfully exploit 10 attacks, and we quantitatively studied these attacks with their input patterns, bug patterns in code (if available), and the efficacy of existing detection tools.

Our study makes four new findings. First, concurrency attacks are much more severe than concurrency bugs. Specifically, once concurrency attacks succeed, fixing only concurrency bugs in code won't help, because attackers may have broken in [6, 7, 10]. This finding suggests that analyzing the known, fixed concurrency bugs is still crucial, because they may have led to concurrency attacks that remain latent.

Second, unlike previous observations in consequence analysis tools [88, 47, 37] that software bugs are often close to their failure/error sites (e.g., bugs and failures are within the same function), our study shows that 12 out of 27 concurrency attacks are widely spread across different functions from their

bugs. Therefore, these consequence analysis tools may be insufficient to detect such concurrency attacks.

Third, although concurrency attacks can cause miscellaneous consequences, these consequences are triggered by several explicit types of vulnerable sites (e.g., `setuid()`). Moreover, although concurrency bugs and and their attack sites spread across different functions, at runtime, the bugs and their attacks often share similar call stack prefixes (§3.2). This finding reveals an opportunity to build a precise, scalable static analysis tool for tracking the bug-to-attack propagation.

Fourth, concurrency bugs and their attacks can often be easily triggered with different, subtle program inputs. Consider only the inputs to trigger concurrency bugs, 8 out of the 10 triggered attacks required less than 20 repetitive executions via subtle inputs. This finding not only contradicts traditional understanding (e.g., [60, 26]) that concurrency bugs are difficult to trigger in native executions and require tremendous retrials, but it also implies that these attacks can easily bypass existing anomaly detection tools [65, 27, 39].

Moreover, triggering concurrency bugs and their attacks often need different inputs. In a Linux root privilege escalation [7], although triggering the data race only required calling the `uselib()` and `mmap()` system calls, other system calls were also needed to get the root shell. This finding poses a significant issue to existing concurrency tools, including model checking tools (e.g., Chess [54]), because they are designed only to catch the race on one input and have no clue on its security consequence that needs another input.

To precisely quantify the efficacy of existing concurrency detection tools on concurrency attacks, we selected two popular dynamic data race detectors TSAN [69] and SKI [32], and we made 6 studied programs run with these tools. We found that most of the tools' reports were benign races, and all the concurrency bugs that can lead to the 10 concurrency attacks we found have been deeply buried in these tools' reports. Our evaluation found 94.3% of race reports benign (§8.2).

Two main reasons make these race detectors ineffective. First, only an extremely small portion of program inputs can lead to concurrency bugs and their attacks. Because race detectors are clueless on even which input will lead to a harmful bug (vulnerable bugs are just a subset of harmful bugs), these detectors can only blindly flag bug reports driven by testing workloads and search "in the dark".

Second, even if a bug-triggering input is identified, a program may still run into too many thread interleavings (or *schedules*), depending on runtime effects (e.g., hardware timings) and synchronization implementations (e.g., adhoc synchronizations [83]). Only a very small portion of schedules will trigger the bug, while the rest may generate excessive, benign reports. For instance, we ran `MySQL` with TSAN and repeatedly generated the same bug-triggering SQL query [10]. We got 202 race reports, but after our manual inspection, only two reports will lead to attacks (§3); most benign reports were caused by `MySQL`'s adhoc synchronizations or benign schedules. In sum, the excessive inputs and schedules caused excessive reports and buried real bugs and attacks.

It's challenging for existing analysis techniques to accurately pinpoint the potentially vulnerable inputs and schedules. One common technique to detect software bugs is static analysis because it can throughly analyze program code and identify what branch statements may be controlled by inputs and may lead to bugs. However, because it lacks runtime effects such as which inputs may take which side of a branch statement, static analysis will typically generate many more false reports than the two dynamic race detectors we ran.

Our key insight is that the reports from existing detectors have implied moderate hints on what inputs and schedules will likely lead to attacks and what will not (e.g., benign bugs). We identify two types of hints. The first hint is *benign schedules*. For instance, 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. Our analysis automatically identified 22 unique static adhoc synchronizations (§8.2).

The second 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 (§3.1), and concurrency attacks sites are often explicit in program code (§3.2). 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.

We did not make this vulnerable input hint automatically generate concrete inputs (can be done via symbolic execution [19, 63]), because we found the call stacks and branches in hints are already expressive enough for us to manually infer vulnerable inputs (§4.3). This input hint helped us identify subtle inputs to trigger both known and unknown attacks (§8.4).

In sum, by directing concurrency bug detectors to focus on the potentially vulnerable inputs and schedules, we can greatly augment existing detectors to approach and detect concurrency attacks. We implemented this directed concurrency attack detection approach in OWL. It first runs concurrency bug detectors on a program to generate reports, and it extracts benign schedule hints (e.g., adhoc synchronizations) and vulnerable input hints from the reports with static analysis. It then automatically annotates the benign schedule hints in a program's code, greatly reducing benign schedules and thus their reports. Finally, it directs detectors and its own runtime vulnerability verifiers (§4.2) to work on the remaining, likely vulnerable inputs and schedules.

We evaluated OWL on 6 diverse, widely used programs, including `Apache`, `Chrome`, `Libsafe`, Linux kernel, `MySQL`,

and SSDB. OWL'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, OWL's 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 first quantitative study on concurrency attacks** and their implications on detection tools. This study will motivate and guide researchers to develop new tools to detect and defend against concurrency attacks (§3).
2. **A new directed concurrency attack detection approach** and its implementation, OWL. OWL can be applied in existing security tools to bridge the gap between concurrency bugs and their security consequences (§7.2).

The rest of this paper is structured as follows. §2 introduces concurrency attack background, and §3 presents our quantitative study. §4 gives an overview of our OWL framework. §5 describes OWL's schedule reduction and §6 its input reduction techniques. §7 discusses OWL's limitations and broad applications. §8 presents our evaluation results for OWL, §9 discusses related work, and §10 concludes.

2 Background

A prior study [85] 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 [7, 10], malicious code injections [8], and bypassing security authentications [4, 3, 5].

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 [85], 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 [85] 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 [85], including HTML integrity violations (§8.4), buffer overflows (§4.3), and DoS attacks (§8.4). 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 [69] for applications and SKI [32] 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 [48] 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 [6, 7], they may stay forever in the system.

Therefore, it's still crucial to study whether existing known concurrency bugs may lead to concurrency attacks.

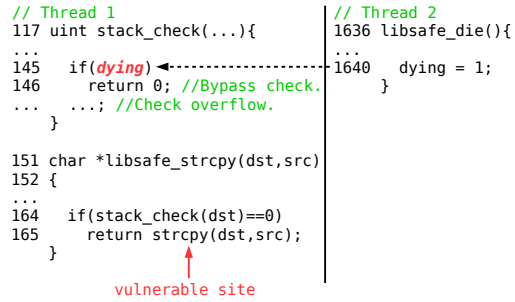

Figure 1: A concurrency attack in the `Libsafe` security library. Dotted arrows mean the bug-triggering thread interleaving. When Thread 2 detects a stack overflow attack, it sets the `dying` variable to 1 and kills current process shortly. However, access to `dying` is not correctly protected by mutex, 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).

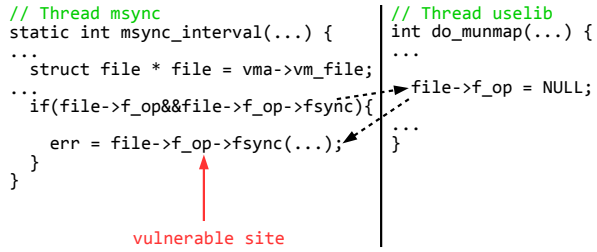

Figure 2: A concurrency attack in the Linux `uselib()` and `msync()` system calls. Dotted arrows mean the bug-triggering thread interleaving. A data race on the `f_op` struct causes the Linux kernel to trigger a NULL pointer dereference and enables arbitrary code execution.

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., [88, 84, 49]) 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 [60, 26] 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 [10], 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* [85] 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 [7], 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 [65, 27, 39] 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 [52]. 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 [10]. 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

This section first presents a major challenge on realizing the directed concurrency attack detection approach (§4.1), gives an overview of OWL's architecture with main components and workflow (§4.2), and then gives an example to show how it works (§4.3).

4.1 Challenge: Accuracy v.s. Scalability

A crucial component for OWL is a good bug-to-attack analysis, but it is technically challenging to make this analysis both accurate (report few false reports and miss few real bugs) and scalable (work with large programs). As mentioned in §1, static analyses are often easy to be scalable as they can see what a compiler can see, but because of the lack of runtime effects (e.g., functions being executed and branches taken), they suffer excessive false reports (e.g., 84% reports in RELAY [75] were false reports).

To better identify runtime effects, symbolic execution [19] systematically explores branches and leverages the program paths to identify buggy inputs. However, this technique is notoriously difficult to scale to large programs (e.g., `Apache` and Linux kernel) because these programs typically have too many functions and program paths [25].

Alternatively, dynamic analyses can accurately capture runtime effects (e.g., [60, 13]), but they can analyze only the executed program path with the exact input and schedule. If a concurrency bug's attack requires another input to trigger in another program path, dynamic analyses may miss the attack.

Fortunately, this challenge can be mitigated via the two optimistic patterns (§3.2) in our study: concurrency attack sites are explicit, and bugs and their attacks often share similar call stack prefixes. These patterns imply that a concurrency bug only affects a small portion of functions and program paths (and thus a small portion of inputs). Thus, we can combine the attack sites (static effects) and calls stacks in reports (dynamic effects), then our static analysis can skip analyzing many functions and program paths that do not comply with these effects, making OWL reasonably accurate and scalable.

4.2 OWL Architecture

Figure 3 presents OWL's architecture with five key components: the concurrency error detector, the static adhoc synchronization detector, dynamic race verifier, static vulnerability detector and dynamic vulnerability verifier.

OWL's work as follows. (1) A concurrency bug detector first detects bugs for the given program inputs. (2) Then, based on the detected results, OWL's adhoc synchronization detector analyzes the reports and LLVM bitcode to find adhoc synchronizations. After obtaining the adhoc synchronization locations, OWL automatically annotates program source code with TSAN markups and re-runs the detector. (3) Then, OWL passes the re-generated bug reports to its race verifier to check whether bugs will actually occur. (4) OWL's static vulnerability analyzer conducts a forward data & control flow analysis to identify potential bug-to-attack propagations as vulnerable input hints. (5) Eventually, OWL's vulnerability verifier reruns the program and checks whether an attack can actually be realized.

4.3 Example

Figure 1 shows a concurrency attack in `Libsafe`, a user-level security library which dynamically intercepts all the Libc memory functions to detect buffer overflows. When `Libsafe` detects a buffer overflow, it sets a global variable `dying` to 1 to indicate that current process will be killed shortly, and then it kills the program. If this variable is set, `Libsafe` will stop performing security checks on memory functions. Unfortunately, there is a data race on `dying` because access to this variable is not protected by mutex. Therefore, between the moment `dying` is set and the moment the entire process is killed, a thread calling memory functions in this process may leverage the race on `dying` to bypass buffer overflow checks.

We have constructed a C program with `Libsafe` to trigger this race, bypassed a stack overflow check for a vulnerability site, `strcpy()`, and gotten a shell by injecting our own malicious code. Note that in this attack, the race and the vulnerability site are in different functions, and the race affects the vulnerability site through an `if` control-dependency at line 164. Existing consequence analysis tools [88, 84, 49] are inadequate to detect this attack because they lack either interprocedural or control-flow analysis.

OWL started from the detection of the data races between

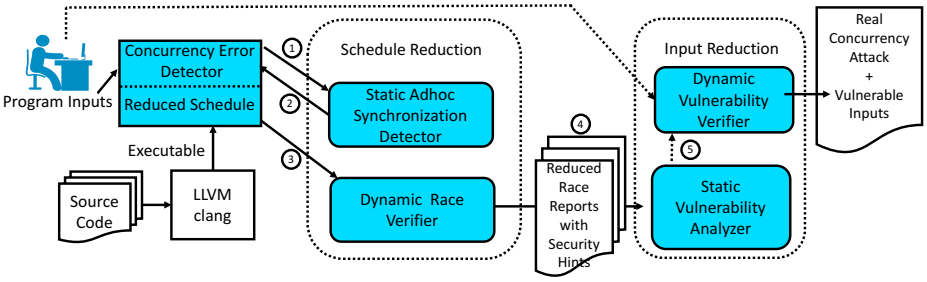

Figure 3: *The* OWL *Architecture.* OWL components are the blue (shaded) ones.

```
libsafe_strcpy (intercept.c:151)
stack_check (util.c:164)
```

Figure 4: `Libsafe` *call stack*. Line numbers refer to Figure 1.

```
---- Ctrl Dependent Vulnerability----
[ 632 ]
%632:   br      %631 if.end13 if.then11  (intercept.c:164)
Vulnerable Site Location: (intercept.c:165)
```

Figure 5: OWL *vulnerable input hint on the* `Libsafe` *attack.* Line numbers refer to Figure 1.

line 145 and line 1640. Our dynamic race verifier verifies this race and pass this report to our vulnerability analyzer. Our vulnerability analyzer starts from the "read" side call stack of the race shown in Figure 4 and conducts a inter-procedural static analysis to detect which vulnerability site may be affected by this race by tracking data and control flows.

As shown in Figure 5, OWL reported one memory operation at line 165 as a vulnerable operation. Our vulnerability report includes the reasoning (a dangerous function will be control-dependent on the corrupted branch statement at line 164) and what are the branches to reach the vulnerability operation. Our report indicates that a `strcpy()` function will be called with the original parameters without the actual security checks in `stack_check()`. At the end, our vulnerability verifier verifies this vulnerability by re-running the program and satisfies the branches to eventually trigger the vulnerability. In order to satisfy the branches, our vulnerability verifier requires user intervention to decide the execution order of the racing instructions and input tunning.

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 §8.2).

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 [83], 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 [66, 58, 36] 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 [66], OWL's verifier achieves the goal without requiring heavyweight Java instrumentation. Com-

pared with ConcurrentBreakpoint [58] and ConcurrentPredicate [36], 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 [36].

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 (§4.1). 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 3).

Second, the algorithm tracks propagation through LLVM virtual registers [50]. Similar to relevant systems [88, 43], our design did not incorporate pointer analysis [81, 46] 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 vulnera-

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()
```

bility 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 (§8.4).

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 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 (§8.3). A typical way to ensure soundness is to plug in a sound alias analysis tool (e.g., [81, 46]) 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 (§8.3).

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 [65, 27, 39] and intrusion detection [40, 76, 77] 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 [45] and atomicity bugs [59]) 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 [11, 12] 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

We evaluated OWL on 6 widely used C/C++ programs, including three server applications (`Apache` [14] web server, `MySQL` [9] database server, and SSDB [72] key-value store server), one library (`Libsafe` [48]), the Linux kernel, and one web browser (`Chrome`). We used the programs' common performance benchmarks as workloads. Our evaluation was done on a 3.60 GHz 8-core Intel Xeon machine with 32 GB memory and 1TB SSD running Linux 3.19.0-49.

We focused our evaluation on four key questions:

1. Is OWL easy to use (§8.1)?
2. How many false reports from concurrency error detection tools can OWL reduce (§8.2)?
3. Can OWL detect known concurrency attacks in the real-world (§8.3)?

4. Can OWL detect previously unknown concurrency attacks in the real-world (§8.4)?

Name	LoC	# atks	# atks found	# OWL's reports
Apache	290K	3	3	10
Chrome	3.4M	1	1	115
Libsafe	3.4K	1	1	3
Linux	2.8M	2	2	34
MySQL	1.5M	2	2	16
SSDB	67K	1	1	2
Total	5.36M	11	10	180

Table 2: OWL *concurrency attack detection results.* We selected 10 concurrency attacks because we were able to trigger their bugs on our machines. OWL detected all 10 evaluated concurrency attacks.

8.1 Ease of Use

Table 2 shows a summary of our concurrency attack detection results. Overall, OWL was able to automatically run the evaluated applications and generate verified concurrency attacks with moderate developer intervention (inspect vulnerable inputs from input hints).

It's critical to report the connection between a concurrency bug and its vulnerability. OWL provides an expressive and helpful reports to (1) let developers know why certain places are vulnerable due to the concurrency bug (2) find the right inputs to trigger concurrency attacks easily. Figure 5 shows a snippet of OWL's report on the `Libsafe` attack in Figure 1.

8.2 Reducing False Reports from Detectors

Table 3 shows OWL'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, OWL 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 OWL's static analysis tool is critical because OWL aims to be scalable to large programs. The last column of Table 3 shows the average time cost of OWL's static analysis tool per bug report. Overall, except for Linux kernel and `Chrome`, OWL's analysis finished analyzing each program's bug reports within a few hours.

8.3 Detecting Known Attacks

We applied OWL on 7 concurrency attacks listed in Table 4. OWL detected all the vulnerabilities. Currently OWL incorporates two race detectors. There are other types of concurrency bugs that can also lead to concurrency attacks, including atomicity violations [52]. Atomicity violations can be detected by other detectors (e.g., CTrigger [59]). By integrating these detectors (future work), OWL's analysis and verifier components can detect more concurrency attacks.

Because all OWL's dynamic verifiers of are implemented based on LLDB, which only supports applications, we haven't run these verifiers in Linux kernel. Nevertheless,

Name	R.R.	A.S.	*R.V.E.	R.	A.C.
Apache	715	7	1506	10	1.5m
Chrome	1715	1	1587	126	190m
Libsafe	3	0	0	3	3s
Linux	24641	8	N/A	1718	42.5m
Memcached	5376	0	5372	4	1.2m
MySQL	1123	6	783	18	5.1m
SSDB	12	0	10	2	1.3m
Total	31870	22	9258	1881	N/A

Table 3: OWL*'s reduction on race detector reports.* R.R. represents Race Reports, A.S. is the static Adhoc Synchronizations annotated by OWL. R.V.E. is Race Verifier Elimination. * This result is the elimination on the race reports after annotation. The number could be bigger than the original race report number. R. is the Remaining result after Race Verifier. A.C. is Average analysis time Cost.

OWL'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 [61]. We leave the implementation in future work.

Aside from the discussed known and unknown concurrency attacks, OWL 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 3 and Table 2), OWL has greatly mitigated developers' burdens.

Name	Vul. Type	Subtle Inputs
Apache-2.0.48	Double Free	PhP queries
Chrome-6.0.472.58	Use after free	Js console.profile
Libsafe-2.0-16	Buffer Overflow	Loops with `strcpy()`
Linux-2.6.10	Null Func Ptr Deref	Syscall parameters
Linux-2.6.29	Privilege Escalation	Syscall parameters
MySQL-5.0.27	Access Permission	FLUSH PRIVILEGES
MySQL-5.1.35	Double Free	SET PASSWORD

Table 4: OWL*'s detection results on known concurrency attacks.* With the listed subtle inputs, all these attacks were often triggered within 20 repeated queries or loops except the `Apache` one.

8.4 Detecting Previously Unknown Attacks

OWL 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 (§3.1), because once attackers break in, they may remain latent for a long time.

OWL detected a new data race and a previously unknown use-after-free concurrency attack in `SSDB` confirmed as `CVE-2016-1000324`. Figure 6 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

```
// Thread 1                                  // Thread 2
355 log_clean_thread_func(void *arg){ 190 ~BinlogQueue(){
356   BinlogQueue *logs =             ...
        (BinlogQueue *)arg;           200   db = NULL;
357                                   201 }
358   while(!logs->thread_quit){
359     if(!logs->db){
360       break;
361     }
...
371     logs->del_range(start, end);
...
375   }
...
380 }
...
341 int del_range(...){
342   while(start <= end){
...
347     Status s = db->Write(...); ← vulnerable site
...
351   }
...
```

Figure 6: ***A new concurrency bug and attack in SSDB-1.9.2.***

which could cause log corruption or program crash if the memory area was reused by other threads.

OWL's static analyzer (§6.1) 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. OWL's dynamic vulnerability verifier (§6.2) 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 7: ***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 7 shows the code of this vulnerability from the Apache-25520 bug [1]. `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 [52], 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 OWL and the first to construct the actual exploit scripts.

OWL's vulnerability analysis (§6.1) pinpointed the vulnerable site at line 1359 and inferred that this line is data dependent on the corrupted variable on line 1358. OWL's dynamic race verifier (§5.2) 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 8: ***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 8 shows the Apache-46215 bug [2]. 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.

OWL detected this concurrency attack as follows. OWL's race detector detected a race between line 617 and line 1192. OWL'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`. OWL's vulnerability analysis (§6.1) reported that a pointer assignment could be control dependent on the corrupted branch

of line 1192. OWL'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 OWL'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 OWL's static bug-to-attack propagation analysis (§6.1), even though the races can be detected by existing race detectors, the security consequences of these bugs were unknown to detectors. Third, without OWL's dynamic race verifier (§5.2) and vulnerability verifier (§6.2), whether these races and their attacks can be realized were unknown either.

9 Related Work

TOCTOU attacks. Time-Of-Check-to-Time-Of-Use attacks [18, 73, 79, 74] 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 [85] 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 [7], injecting malicious code [6], 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 [28, 62, 55, 56], anomaly detection [27, 65], address space randomization [70], and static analysis [38, 30, 76, 17, 19].

However, with the presence of multithreading, most existing sequential defense tools can be largely weakened or even completely bypassed [85]. 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 [87, 64, 29, 51, 53, 89, 88, 44, 80], diagnosis [67, 59, 57, 16, 43], and correction [42, 78, 82, 41]. They focused on concurrency bugs themselves, while OWL focuses on the security consequences of concurrency bugs. Therefore, these systems are complementary to OWL.

Conseq [88] 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 [49, 84, 31, 15, 71, 90]. These approaches fall into two categories based on whether they target general or specific programs.

The first category [49, 84] 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. [49] 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 §4.3. Yamaguchi et al. [84] 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., [84] 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 [31, 15, 71, 90] lets static analysis focus on specific behaviors (e.g., APIs) in specific programs to achieve scalability and accuracy. These approaches check web application logic [31], Android applications [15], cross checking security APIs [71], and verifying the Linux Security Module [90]. 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 [34, 68, 33, 35, 19, 86, 20, 23, 21, 63], block malicious inputs [24], preserve privacy in error reports [22], and detect programming rule violations [25]. Specifically, UCKLEE [63] 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 https://github.com/ruigulala/ConAnalysis.

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. http://www.mysql.com/.

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

[11] Ten historical software bugs with extreme consequences. http://royal.pingdom.com/2009/03/19/10-historical-software-bugs-with-extreme-consequences/.

[12] Ten seriously epic computer software bugs. http://listverse.com/2012/12/24/10-seriously-epic-computer-software-bugs/.

[13] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, Oct. 2009.

[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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 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*, 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 and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 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] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *Fifth Workshop on Hot Topics in System Dependability (HotDep '09)*, 2009.

[24] 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.

[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] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security '10)*, pages 10–10, 2010.

[32] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 415–431, Oct. 2014.

[33] 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.

[34] 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.

[35] 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.

[36] J. Gottschlich, G. Pokam, C. Pereira, and Y. Wu. Concurrent predicates: A debugging technique for every parallel programmer. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 331–340, Sept 2013.

[37] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of linux kernel behavior under errors. In *DSN '03*, pages 459–468, 2003.

[38] 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.

[39] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 291–301, 2002.

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

[41] 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.

[42] 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.

[43] 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.

[44] B. Kasikci, C. Zamfir, and G. Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[45] 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.

[46] 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.

[47] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 181–192, 2007.

[48] Libsafe. http://directory.fsf.org/wiki/Libsafe.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, Dec. 2008.

[55] 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.

[56] 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.

[57] 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.

[58] C. S. Park and K. Sen. Concurrent breakpoints. Technical Report UCB/EECS-2011-159, EECS Department, University of California, Berkeley, Dec 2011.

[59] 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.

[60] 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.

[61] http://www.qemu.org.

[62] 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.

[63] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Aug. 2015.

[64] 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.

[65] 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.

[66] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[67] 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.

[68] 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.

[69] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*, pages 62–71, 2009.

[70] 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.

[71] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple api implementations. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 343–354, 2011.

[72] Ssdb. `https://github.com/ideawu/ssdb`.

[73] 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.

[74] 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.

[75] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE '07)*, pages 205–214, 2007.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

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

[82] 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.

[83] 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.

[84] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*, pages 590–604, 2014.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.