ConFuzz: Coverage-guided Property Fuzzing for Event-driven Programs

Sumit Padhiyar and KC Sivaramakrishnan

IIT Madras, India
{sumitpad,kcsrk}@cse.iitm.ac.in

Abstract. Bug-free concurrent programs are hard to write due to non-determinism arising out of concurrency and program inputs. Since concurrency bugs typically manifest under specific inputs and thread schedules, conventional testing methodologies for concurrent programs like stress testing and random testing, which explore random schedules, have a strong chance of missing buggy schedules.
In this paper, we introduce a novel technique that combines property-based testing with mutation-based, grey box fuzzer, applied to event-driven OCaml programs. We have implemented this technique in ConFuzz, a *directed* concurrency bug-finding tool for event-driven OCaml programs. Using ConFuzz, programmers specify high-level program properties as *assertions* in the concurrent program. ConFuzz does not require any modification to the concurrent program, which is free to perform arbitrary I/O operations. Our experimental results show that ConFuzz is easy-to-use, effective, detects concurrency bugs faster than Node.Fz - a random fuzzer for event-driven JavaScript programs, and is able to reproduce known concurrency bugs in widely used OCaml libraries.

Keywords: Concurrency testing · Fuzzing.

1 Introduction

Event-driven concurrent programming is used in I/O heavy applications such as web browsers, network servers, web applications and file synchronizers. On the client-side, JavaScript natively supports event-driven programming through promises and async/await [3] in order to be able to retrieve multiple resources concurrently from the Web, without blocking the user-interface rendering. On the server-side, several popular and widely used frameworks such as Node.js (JavsScript) [28], Lwt (OCaml) [24,38], Async (OCaml) [2], Twisted (Python) [36], use event-driven concurrent programming model for building scalable network services.

Event-driven programs are typically single-threaded, with the idea that rather than performing I/O actions synchronously, which may block the execution of the program, all the I/O is performed asynchronously, by attaching a *callback function* that gets invoked when the I/O operation is completed. An *event loop* sits at the heart of the programming model that concurrently performs the I/O operations, and schedules the callback functions to be resumed when the corresponding I/O is completed. The concurrent I/O is typically offloaded to a library such as libuv [22] and libev [21], which in turn discharge concurrent I/O through efficient operating system dependent mechanisms such as epoll [11] on Linux, kqueue [19] on FreeBSD, OpenBSD and macOS, and IOCP [17] on Windows.

Single-threaded event-driven programs avoid concurrency bugs arising from multi-threaded execution such as data races and race conditions. Despite this, event-driven programs suffer from concurrency bugs due to the non-deterministic order in which the events may be resolved. For example, callbacks attached to a timer event and DNS resolution request may execute in different orders based on the order in which the events arrive. The fact that event-driven programs interact with the external world in potentially blocking fashion makes it unsuitable to apply data race detectors developed for detecting multi-threading bugs [13,40].

Moreover, the erroneous condition in a concurrent program may not be the mere presence of a race, but a complex assertion expressed over the current program state. For example, in the case of a timer event and DNS resolution request, the timer may be intended for timing out the DNS resolution request. On successful resolution, the timer event is cancelled. Then, the safety property is that if the timer callback is running, then the DNS resolution request is still pending. It is unclear how to express this complex property as races.

To help uncover such complex concurrency bugs that may arise in event-driven concurrent programs, we present a novel technique that combines property-based testing on the lines of QuickCheck [6] with AFL fuzzer [1], the state-of-the-art mutation-based, grey box fuzzer, and apply it to generate not only inputs that may cause the property to fail, but also to drive the various scheduling decisions in the event-driven program. AFL works by instrumenting the program under test to observe the control-flow edges, mutates the input such that new paths are uncovered. In addition to different paths, a concurrent program also has to contend with the exponential number of schedules available, many of which may lead to the same behaviour. Our key observation is that we can use AFL's grey box fuzzing capability to direct the search towards new schedules, and thus lead to property failure.

We have implemented this technique in ConFuzz, a concurrent property fuzz testing tool for concurrent OCaml programs using the popular Lwt [24,38] library. Properties are expressed as assertions in the source code, and ConFuzz aims to identify the input and the schedule that will cause the assertion to fail. ConFuzz supports record and replay to reproduce the failure. Once a bug is identified, ConFuzz can *deterministically reproduce* the concurrency bug. ConFuzz is developed as a drop-in replacement for the Lwt library and does not require any change to the code other than writing the assertion and the wrapper code to drive the tool.

The main contributions of this paper are as follows:

– We present a novel technique that combines property-based testing with mutation-based, grey box fuzzer applied to test the schedules of event-driven OCaml programs.
– We implement the technique in ConFuzz, a drop-in replacement for testing event-driven OCaml programs written using the Lwt library.
– We show by experimental evaluation that ConFuzz is more effective and efficient than the state-of-the-art random fuzzing tool Node.Fz and stress testing in finding concurrency bugs. We reproduce known concurrency bugs by testing ConFuzz on 8 real-world concurrent OCaml programs and 3 benchmark programs.

2 Motivating example

We describe a simple, adversarial example to illustrate the effectiveness of ConFuzz over Node.Fz and stress testing. Figure 1 shows an OCaml concurrent program written using the Lwt library [38]. The program contains a single function `linear_eq` that takes an integer argument `i`. `linear_eq` creates three concurrent tasks `p1`, `p2`, and `p3`, each modifying the shared mutable reference `x`. The `pause` operation pauses the concurrent task, registering the function `fun ()-> ...` following the `>>=` operator as a callback to be executed in the future. Importantly, the tasks `p1`, `p2`, and `p3` may be executed in any order.

```
let linear_eq i =
  let x = ref i in
  let p1 = pause () >>= fun () ->
    x := !x - 2; return_unit in
  let p2 = pause () >>= fun () ->
    x := !x * 4; return_unit in
  let p3 = pause () >>= fun () ->
    x := !x + 70; return_unit in
  Lwt_main.run (join[p1;p2;p3]);
  assert (!x <> 0)
```

Fig. 1: A program with a concurrency bug

This program has a concurrency bug; there exists a particular combination of input value `i` and interleaving between the tasks that will cause the value of `x` to become 0, causing the assertion to fail. There are 2^{63} [1] possibilities for the value of `i` and 6 (3!) possible schedules for the 3 tasks. Out of these, there are only 3 possible combinations of input and schedule for which the assertion fails.

– `i` = -17 and schedule = `[p2; p1; p3]` : $((-17 * 4) - 2) + 70 = 0$.
– `i` = -68 and schedule = `[p1; p3; p2]` : $((-68 - 2) + 70) * 4 = 0$.
– `i` = -68 and schedule = `[p3; p1; p2]` : $((-68 + 70) - 2) * 4 = 0$.

As the bug in example program depends on input and interleaving, concurrency testing techniques focusing only on generating different interleavings will fail to find this bug. This is evident when the program is executed under different testing techniques. Table 2 shows a comparison of ConFuzz with the random concurrency fuzzing tool Node.Fz[7] and stress testing for the example program.

[1] OCaml uses tagged integer representation [20]

Fig. 2: Comparing different testing techniques

Testing Technique	**Executions (millions)**	**Time (minutes)**	**Bug Found**
ConFuzz	3.26	18	**Yes**
Node.Fz[7]	110	60	**No**
Stress	131	60	**No**

Node.Fz is a concurrency fuzzing tool similar to ConFuzz, which generates random interleavings rather than being guided by AFL. Node.Fz focuses only on finding buggy interleavings. Stress testing runs the program normally without controlling the scheduling decisions. We test the example program with each technique until a bug is found or a timeout of 1 hour is reached. We report the number of executions and time taken if the bug was found. Only ConFuzz was able to find the bug. Although this example is synthetic, we observe similar patterns in real world programs where the bug depends on the combination of the input value and the schedule, and cannot be discovered with a tool that only focuses on one of the sources of non-determinism.

```
let pipe_chars a b c =
  let res = ref [] in
  let ic, oc = pipe () in
  let sender =
    write_char oc a >>= fun () ->
    write_char oc b >>= fun () ->
    write_char oc c >>= fun () ->
    return_unit
  in
  let recvr () =
    read_char ic >>= fun c ->
    res := Char.uppercase_ascii c::!res;
    return_unit
  in
  Lwt_main.run (join [recvr(); recvr();
                      recvr(); sender]);
  assert (!res <> ['B';'U';'G'])
```

Fig. 3: A program with a concurrency bug

Real world event-driven programs also involve file and network I/O, timer completions, etc. ConFuzz can test *unmodified* programs that involve complex I/O behaviour. Figure 3 shows a function `pipe_chars` that takes three character arguments. The function creates a shared `pipe` as a pair of input (`ic`) and output (`oc`) file descriptors. The `sender` task sends the characters over `oc`. The three `recvr` tasks each receive a single character, convert that to the corresponding upper case character, and append it to a global list reference `res`. The assertion checks that the final result in `res` is not `['B';'U';'G']`. Due to input and scheduling non-determinism, there are plenty of schedules. However, the assertion failure is triggered with only 6 distinct inputs, each of which is a permutation of `'b'`, `'u'`, `'g'` for the input arguments to the function, and a corresponding permutation of the `recvr` tasks. ConFuzz was able to find a buggy input and schedule in under a minute. This illustrates that ConFuzz is applicable to real world event-driven concurrent programs.

3 Lwt: Event-driven model

In this section, we discuss the event-driven model in Lwt. Lwt [38] is the most widely used asynchronous I/O library in the OCaml ecosystem. Lwt lies at the heart of the stack in the MirageOS [25], a library operating system for constructing Unikernels. MirageOS is embedded in Docker for Mac and Windows apps [8] and hence, runs on millions of developer machines across the world. Hence, our choice of Lwt is timely and practical. That said, the ideas presented in this paper can be applied to other event-driven libraries such as Node.js [28]. Lwt event model is shown in Figure 4.

Under cooperative threading, each task voluntarily yields control to other tasks when it is no longer able to make progress. Lwt event model consists of an event loop engine and a worker pool. The event loop engine manages timers, read and write I/O events on registered file descriptors and executes the callbacks registered with the events. Lwt event loop engine can be configured to use various engines such as `libev` [21], Unix's select [34] and poll [31].

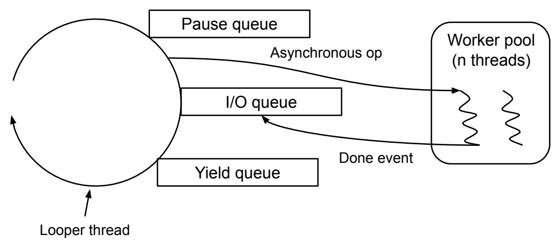

Fig. 4: Lwt event model

Lwt event loop consists of three event queues, each holding a different class of events with their attached callbacks. The three queues are `yield`, `pause` and `I/O` queue. All yielded and paused callbacks are inserted in yield and pause queue respectively. The `I/O` queue comprises of the timer and I/O events and is handled by `libev` engine. The *looper thread* examines each of the queues and executes the pending callbacks without interruption until they give up control. Lwt does not guarantee the relative execution order between two events in the same or different queues. The computationally intensive tasks and blocking system calls are offloaded to the *worker pool* of threads which execute the tasks so that they do not block the event loop. The non-determinism in the execution order of the events gives rise to concurrency bugs.

4 ConFuzz

In this section, we present the architecture of the ConFuzz tool, starting with a background of the technique ConFuzz builds upon.

4.1 Background

Fuzzing is an effective technique for testing software by feeding random input to induce the program to crash. American Fuzzy Lop(AFL) [1] is coverage-guided fuzzer [29], which inserts lightweight instrumentation in the program under test to collect code coverage information such as program execution paths. AFL starts with a seed test case, which it mutates with a combination of random and deterministic techniques that aims to find new execution paths in the program. On detecting a new execution path, the corresponding input test case is saved for further mutation. During fuzzing, the input test cases that results in a crash are saved, thus finding the exact test case that results in a crash.

Property-based testing, introduced by QuickCheck [6], works by testing an executable predicate (predicate) on a stream of randomly generated inputs. Property-based testing is typically equipped with a generator that randomly generates inputs for the executable predicate. While property based testing works well in many cases, random generation of inputs may not cover large parts of the programs where the bugs may lie. Crowbar [9] is a testing tool for OCaml that combines property-based testing with AFL. Rather than generating random inputs, the inputs are generated by AFL to maximise the discovery of execution paths in the function under test.

4.2 Architecture

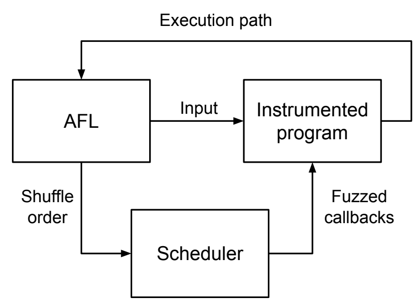

Fig. 5: ConFuzz architecture

ConFuzz extends Crowbar in order to generate inputs that maximize the coverage of the state space introduced by non-deterministic execution of concurrent event-driven programs. Figure 5 shows ConFuzz's architecture. ConFuzz controls Lwt's scheduler by capturing the non-determinism present in the Lwt programs. To explore properties on a wide range of different schedules, ConFuzz generates various legal event schedules by alternating the order of event callback execution with the help of AFL. AFL generates execution order (*shuffle order*) for the captured concurrent events, which is then enforced by controlled scheduler (*fuzzed callbacks*). The properties are tested repeatedly with different test inputs and event schedules. The test input and the event schedules that result in property failures are detected as a crash by AFL, resulting in the detection of concurrency bug.

Unlike other concurrency testing tools [7,27], which fuzz the schedules for a specific input, ConFuzz can fuzz both input and the schedule, which improves both the ease-of-use and effectiveness of the tool. Similar to other concurrency testing tools, ConFuzz also supports record and replay feature, which records the event schedule that leads to a concurrency bug. The input and the buggy schedule are saved in a separate file, which when executed with test binary, deterministically reproduces the bug. Thus, ConFuzz helps to debug concurrency bug by reliably reproducing the bug.

5 Fuzzing under Non-determinism

In this section, we discuss the non-determinism present in Lwt concurrent programs. We then show how ConFuzz captures and fuzzes this non-determinism.

5.1 Non-determinism in Lwt

I/O and timer non-determinism Lwt permits asynchronous network and file I/O by registering callbacks against I/O operations. Since the completion of I/O operations is non-deterministic, the callbacks may be triggered in a non-deterministic order. Moreover, there is also the possibility of races between callbacks in order to access shared resources such as sockets and file descriptors. Lwt also supports registering callbacks to timer events which are triggered at *some time* after the expiration of the timer. Any assumption about the precise time at which the callbacks will run may lead to concurrency bugs. As described in Section 3, both the timer and the I/O events are processed in the I/O event queue. For example, a user code expecting a network request to complete within a certain time period may go wrong if the network callback is executed after the timer callback. This bug can be caught by fuzzing the I/O event queue that reorders the timer and the network callbacks.

Worker pool non-determinism Lwt offloads blocking system calls and long running tasks to the worker pool. Lwt uses a worker pool comprising of a fixed number of kernel threads. The kernel threads are scheduled by the operating system and makes the execution of offloaded tasks non-deterministic.

Callback non-determinism Lwt `yield` and `pause` primitives enable long running computation to give up execution to another callback voluntarily. In Lwt, the yielded and paused callbacks may be evaluated in any order. Any assumption on the order in which the yielded and paused callbacks are executed may lead to concurrency bugs. These bugs can be identified by fuzzing the yield and pause queues.

5.2 Capturing Non-determinism

In this section, we discuss how ConFuzz controls the non-determinism described in Section 5.1.

Event loop queues Lwt event loop queues – yield, pause and I/O – are the primary sources of non-determinism in an Lwt program. To capture and control the non-determinism arising out of these queues, we insert calls to ConFuzz scheduler in the event loop before executing the callbacks of yield and pause queues. ConFuzz scheduler then fuzzes the order of callbacks in the queues to generate alternative schedules.

Worker pools Non-determinism in the worker pool is influenced by multiple factors such as the number of threads, the thread scheduling by the operating system and the order in which the tasks are offloaded. For deterministic processing and completion order of tasks, we reduce the worker pool size to one. This change serializes the tasks handled by the worker pool. The worker pool tasks are executed one after another. By reducing the worker pool to one thread, ConFuzz can deterministically replay the order of worker pool task execution.

To signal task completion, the worker pool thread writes to a single common file descriptor registered which is intercepted by the event loop and processed as an I/O event. The single file descriptor is shared by all the tasks for indicating task completion. Thus, Lwt multiplexes a single file descriptor for many worker pool tasks. Multiplexing prevents changing the order of task completion relative to I/O and and as a result, miss some of the bugs.

To overcome this, ConFuzz eliminates multiplexing by assigning a file descriptor per task. During the event loop I/O phase, task completion I/O events are fuzzed along with other I/O events and timers. De-multiplexing enables ConFuzz to shuffle the order of task completion relative to other tasks as well as timer and I/O events.

To change the processing order of worker pool tasks, we delay the execution of offloaded tasks. During each iteration of the event loop, the offloaded tasks are collected in a list. At the start of the next iteration of the event loop, ConFuzz scheduler shuffles the processing order of the tasks. The tasks are then executed synchronously. By delaying the task execution by one iteration, ConFuzz collects enough tasks to shuffle. We believe that delaying tasks by one iteration would suffice to generate the task processing orders that would occur in production environments. It is highly unlikely that a task from the second iteration is started and completed before tasks from first the iteration, given that Lwt tasks are started off in a FIFO manner.

Synchronous task execution also helps in deterministically generating a buggy schedule. As the number of completed tasks remains the same in every schedule, ConFuzz has to just reorder tasks to reproduce a bug. This design choice lets ConFuzz generate task processing and completion order independently. However, delaying and synchronous task execution can prevent ConFuzz from missing schedule containing bugs arising from the worker pool related races. In ConFuzz, we trade-off schedule space generation to reliably reproducing concurrency bug by deterministic schedule generation. ConFuzz does not guarantee the absence of bugs but reliably reproduces discovered concurrency bugs.

Promise callbacks As promise callbacks are executed non-deterministically, promise callback ordering is also fuzzed by ConFuzz. Before execution, the order of callbacks attached to a promise is changed by ConFuzz scheduler. By fuzzing promise callbacks, ConFuzz generates alternative ordering of callback execution.

5.3 ConFuzz scheduler

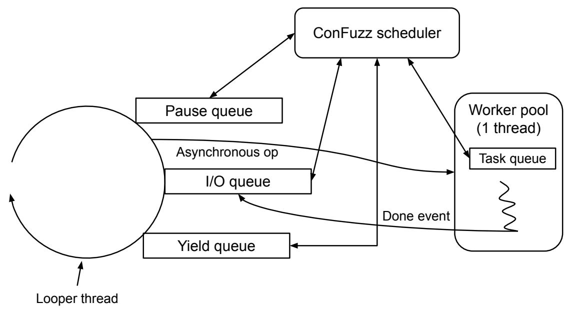

Fig. 6: ConFuzz scheduler

To generate a varied event schedules, ConFuzz scheduler controls the Lwt event loop and the worker pool as shown in Figure 6. To change the order of events, ConFuzz scheduler exposes `fuzz_list : 'a list -> 'a list` function, which takes a list and returns a shuffled list. The changes to the Lwt scheduler that require changing the order of events (Section 5.2) call this function to shuffle the callback list. On executing the shuffled list, the program is executed under a particular schedule.

To reorder callbacks, ConFuzz scheduler asks AFL to generate random numbers. The random numbers then determine the ordering of the callbacks according to Fisher-Yates shuffle algorithm [12]. On detecting a concurrency bug, the generated random numbers are saved in a separate file as a schedule trace. With the schedule trace, the scheduler can reproduce a schedule. Using this capability of the scheduler, ConFuzz can replay a schedule to reliably expose the detected concurrency bugs. Deterministic replay helps programmers find the exact cause of concurrency bugs.

The order of callback execution affects the program's execution path. Due to the program instrumentation, AFL recognises the program execution path in every program run. AFL being a coverage guided fuzzer, tries to increase coverage (execution paths). AFL thus generates random numbers that produce alternative callback orderings. Alternative callback orderings result in new schedules that exercise new program execution paths. ConFuzz scheduler keeps on generating new schedules until AFL is able to find new execution paths. ConFuzz thus uses AFL fuzzing to execute program under different execution schedules.

6 Evaluation

In this section, we evaluate the effectiveness of ConFuzz in finding concurrency bugs in real-world OCaml applications and benchmark programs. Additionally, we check the efficiency of ConFuzz in terms of time required to detect concurrency bugs in comparison to Node.Fz and stress testing. Node.Fz[7] is a concurrency bug finding fuzzing tool for event-driven JavaScript programs. As Node.Fz randomly perturbs the execution of a JavaScript program, we use ConFuzz's random testing mode(Section 4.2) to simulate Node.Fz. Stress testing runs the program normally without controlling the scheduling decisions. We design and conduct experiments to answer the following questions:

1. **RQ1: Effectiveness** – How frequently is ConFuzz able to find bugs?

Table 1: Experimental subjects

Type	Name(abbreviation)	Description	GitHub Stars	Size (LoC)	Issue #
Real world applications	irmin(IR)	Distributed database	1,284	18.6K	270
	lwt (LWT)	Concurrent programming library	448	12.2k	583
	mirage-tcpip (TCP)	Networking stack for Mirage OS	253	4.9K	86
	ghost (GHO)	Blogging engine	35,000	50K	1834
	porybox (PB)	Pokmon platform	29	7.9K	157
	node-mkdirp (NKD)	Recursive mkdir	2,200	0.5K	2
	node-logger-file (CLF)	Logging module	2	0.9K	1
	fiware-pep-steelskin (FPS)	Policy enforcement point proxy	11	8.2K	269
Benchmark programs	Motivating example(MX)	Linear equation with concurrency	-	-	-
	Benchmark 1(B1)	Bank transactions	-	-	-
	Benchmark 2(B2)	Schedule coverage	-	-	-

2. **RQ2: Efficiency** – How many executions are required to detect bugs by ConFuzz as compared to Node.Fz and stress testing?
3. **RQ3: Practicality** – Can ConFuzz detect and reproduce known concurrency bugs in real-world OCaml applications?

6.1 Experimental subjects and setup

We evaluated ConFuzz on both real-world OCaml applications and benchmark programs. Table 1 summarises the applications and benchmark programs used for the evaluation. We have used eight real-world applications and three benchmark programs as experimental subjects for evaluating ConFuzz. All of the programs contain at least one known concurrency bug.

To identify known concurrency bugs, we searched across GitHub[2] bug reports for closed bugs in Lwt based OCaml projects. We select a bug only if the bug report contains a clear description or has an automated test case to reproduce the bug. We found three Lwt based OCaml bugs - IR, LWT and TCP as shown in Table 1. Apart from OCaml bugs, we have build a dataset of 15 known concurrency real-world JavaScript bugs mentioned in the related work [5,7,39]. We abstracted the buggy concurrent code of JavaScript bugs and ported it to standalone OCaml programs. We excluded those bugs from the JavaScript dataset which could not be ported to OCaml or have an incomplete bug report. We were able to port 5 JavaScript bugs from the dataset. The five JavaScript bugs used in the evaluation are GHO, PB, NKD, CLF and FPS. MX is the motivating example from section 2. Benchmark B1 simulates concurrent bank transactions, adapted from the VeriFIT repository of concurrency bugs [37]. Concurrent bank transactions in B1 causes the bank account log to get corrupted. Benchmark B2

[2] https://github.com

simulates a bug depending on a particular concurrent interleaving and gets exposed only when B2 is executed under that buggy interleaving. B2 is explained in detail in section RQ2.

We design our experiments to compare ConFuzz's bug detection capability with Node.Fz and stress testing (hereby referred to as the testing techniques). We perform 30 testing runs for each experimental subject (Table 1) and testing technique. A *testing run* is a single invocation of the testing technique. The performance metric we focus on is mean time to failure (MTTF), which measures how quickly a concurrency bug is found in terms of time. A single *test execution* indicates one execution of the respective application's test case. For each subject and testing technique, we execute respective subject application until the first concurrency bug is found or a timeout of 1 hour occurs. For each such run, we note the time taken to find the first concurrency bug and whether a bug was found or not. We ran all of our experiments on a machine with 6-Core Intel i5-8500 processor, 16GB RAM, running Linux 4.15.0-1034.

Table 2: Bug detection capability of the techniques. Each entry is the fraction of the testing runs that manifested the concurrency bug.

	Stress	**Node.Fz**	**ConFuzz**
IR	1.00	0.00	1.00
LWT	0.00	1.00	1.00
TCP	0.00	1.00	1.00
GHO	0.00	0.00	1.00
PB	0.00	0.00	1.00
NKD	0.4	0.53	1.00
CLF	0.43	0.56	1.00
FPS	0.00	0.96	1.00
MX	0.00	0.00	1.00
B1	0.87	0.6	1.00
B2	0.00	0.00	1.00
Avg	0.24	0.42	1.00

6.2 Experimental results

RQ1: Effectiveness Table 2 shows the bug detection capabilities of the three testing techniques. The first column shows the abbreviation of the experimental subjects. The second to fourth column shows the bug detection results of Stress, Node.Fz and ConFuzz testing, respectively. Each cell in the table shows the fraction of the testing runs that detected a concurrency bug out of the total 30 testing runs per experimental subject and testing technique.

As shown in Table 2, ConFuzz detected concurrency bugs in every testing run for all experimental subjects (all cells are 1.00). In the case of GHO, PB, MX and B2, only ConFuzz was able to detect bug. Despite capturing the non-determinism, Node.Fz could not

Table 3: Mean time to find the concurrency bug (seconds)

	Stress	**Node.Fz**	**ConFuzz**
IR	37.7	-	1.03
LWT	-	295.73	243.3
TCP	-	315.03	94.16
GHO	-	-	0.33
PB	-	-	0.3
NKD	1738.83	1104.62	42.23
CLF	685.1	1086.2	231.96
FPS	-	696.55	103.13
MX	-	-	981.17
B1	918.8	1333.89	384.6
B2	-	-	59.26

detect a bug in IR, GHO, PB, MX and B2. This confirms that ConFuzz was able to generate concurrent schedules along with inputs more effectively. Stress testing was more effective in the case of IR and B1 than Node.Fz with a ratio of 1.00 and 0.87 respectively. Both IR and B1 comprises a lot of files I/O. We suspect that due to OS-level non-determinism, stress testing is more effective than Node.Fz, as Node.Fz finds it difficult to generate the exact buggy schedule for file I/O. This provides a helpful insight that ConFuzz is good at generating a prefix or exact schedule that can cause concurrency errors. In addition, ConFuzz does not produce *false positives*, as schedules explored by ConFuzz are all legal schedules in Lwt. Thus, the results confirm that ConFuzz is effective at detecting concurrency bugs.

RQ2: Efficiency Table 3 shows the efficiency results of the three testing techniques. The second to fourth column shows the efficiency results of stress, Node.Fz and ConFuzz testing respectively. Each cell represents the average time (in seconds) taken to detect the first concurrency bug per experimental subject and testing technique over 30 testing runs. '-' in the cell indicates that none of the 30 testing runs detected a concurrency bug within the timeout of 1 hour.

As shown in Table 3, for every experimental subject, ConFuzz took significantly less time (column 4) to find bug than other techniques. ConFuzz is 26×, 6× and 4.7× faster than Node.Fz for NKD, FPS and CLF bugs respectively. For NKD and IR bugs, ConFuzz is 41× and 36× faster than stress testing respectively. Except for LWT, ConFuzz is at least 2× faster than second fastest technique.

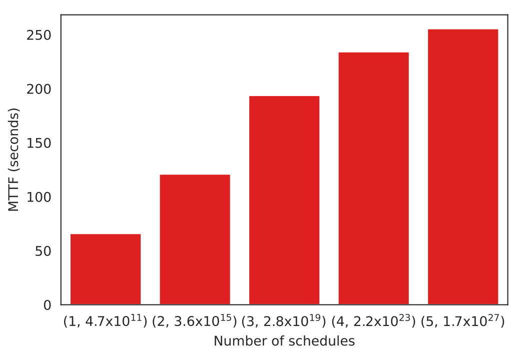

Fig. 7: Efficiency of ConFuzz as schedule space increases. The total number of schedules is given by $f(n) = (3!)^{(10*n+20)/2)}$. The labels on the x-axis show $(n, f(n))$.

Note that for NKD, CLF, FPS and B1 bugs, the average time of Node.Fz and stress testing does not include testing runs which failed to detect concurrency bug. Due to its efficiency, ConFuzz enables a developer to explore a broader schedule space of the concurrent program than Node.Fz and other techniques with the same test time budget. Thereby increasing the chances of finding bug in the same limited test time budget. Thus, these results illustrate that ConFuzz is efficient in detecting concurrency bugs.

To evaluate the efficiency of ConFuzz on a program containing a large schedule space, we modify the motivating example in Figure 1 to have a large number of concurrent schedules. We define a concurrent schedule as the order in which the callbacks attached to the events are executed. The total number of concurrent schedules of the modified program is given by the following formula parameterised over n:

$$Total\ number\ of\ schedules = (3!)^{(10*n+20)/2}$$

where n controls the degree of concurrency in the program. Only one concurrent schedule out of the many schedules results in a concurrency bug. Figure 7 shows the efficiency of ConFuzz over large schedule spaces. We increase n from 1 to 5 to generate a large schedule space. Note that benchmark B2 used as an experimental subject in evaluation is modified program with n equals to 1. Figure 7 graph shows mean time to failure (MMTF) as the schedule space is increased. As evident from the graph, even for the program with a large schedule space, ConFuzz was able to detect the bug within minutes. Note that Node.Fz and stress testing fails to detect the bug for the modified program. Despite the number of schedules increasing exponentially, MTTF increased linearly. This shows the efficiency of ConFuzz to find bugs even in programs with large schedule spaces.

RQ3: Practicality As shown in Table 2, ConFuzz is able to reliably reproduce concurrency bugs in real-world applications and widely used software. Table 1 includes links to the original bug reports. We have made available an anonymised replication package comprising of the ConFuzz tool and the experimental subjects online[3]. In the sequel, we discuss the details of the irmin(IR) bug from Table 1.

Irmin #270 [4] Irmin is a distributed database built on the principles of Git. Similar to Git, the objects in Irmin are organised into directories. Irmin allows users to install *watchers* on directories which are callback functions that get triggered once when for every change in the directory. The bug had to do with the callbacks begin invoked multiple times if multiple watchers were registered to the same directory in quick succession. The patch is shown in Figure 8. When there are concurrent calls to `start_watchdog` in succession, it might turn out that all of them are blocked at

```
let start_watchdog ~delay dir =
  match watchdog dir with
  | Some _ ->
    assert (nb_listeners dir <> 0);
    Lwt.return_unit
  | None   ->
-   Log.debug "Start watchdog for %s" dir;
+   (* Note: multiple threads can wait here
  *)
    listen dir ~delay ~callback >|=
        fun u ->
-   Hashtbl.add watchdogs dir u
+   match watchdog dir with
+   | Some _ -> u ()
+   | None ->
+   Log.debug "Start watchdog for %s" dir;
+   Hashtbl.add watchdogs dir u
```

Fig. 8: Irmin bug #270

[3] See https://github.com/ConFuzzAnon/PADL_2021

[4] https://github.com/mirage/irmin/issues/270

`listen`. When the callback is triggered, each of these callbacks now adds an entry to the `watchdogs` hash table. The fix is to only add one entry to the hash table and for the rest, directly call the callback function. The property that we tested was that the callback function is invoked only once. Observe that the bug is input dependent; the bug is triggered only if concurrent calls to `start_watchdog` work on the same directory `dir` and the `delay` is such that they are all released in the same round.

7 Limitations

While our experimental evaluation shows that ConFuzz is highly effective in finding bugs, we discuss some of the limitations in our approach. While ConFuzz captures most of the non-determinism present in event-driven concurrent programs, it cannot capture and control external non-determinism such as file read/write or network response. External non-determinism arises when interacting with external resources like file system, database, etc. which are outside the scope of ConFuzz.

To be completely certain about the order in which the asynchronous tasks are executed, ConFuzz serializes the worker pool tasks which might result in missing some of the concurrency bugs arising out of the worker pool-related races (although the concurrency bug study by Davis et al. [7] did not identify any such races). Serializing worker pool tasks help ConFuzz to *deterministically reproduce* detected bugs. We trade-off missing some of the worker pool related concurrency bugs with the deterministic reproducibility of the detected bugs. Being a property-based testing framework, ConFuzz aims to generate failing tests cases that falsify the property. Hence, ConFuzz does not aim to detect traditional concurrency bugs such as data races and race conditions.

8 Related Work

To the best of our knowledge, ConFuzz is the first tool to apply coverage-guided fuzzing, not just to maximize the coverage of the source code of program, but also to maximize the schedule space coverage introduced by a non-deterministic event-driven program. In this section, we compare ConFuzz to related work.

Concurrency fuzzing: AFL has been used previously to detect concurrency vulnerabilities in a Heuristic Framework [23] for multi-threaded programs. Unlike ConFuzz, Heuristic Framework generates interleavings by changing thread priorities instead of controlling the scheduler directly, thereby losing the bug replay capability. Due to its approach, Heuristic Framework can only find specific type of concurrency bugs and has false positives. Heuristic Framework is applied to multi-threaded programs whereas ConFuzz is applied to event-driven programs. The most similar work to ConFuzz is the concurrency fuzzing tool Node.Fz [7]. Node.Fz fuzzes the order of events and callbacks randomly to explore different schedules. Node.Fz can only find bugs that manifest purely as a result of particular scheduling, not as a property of program inputs. As Section 6 illustrates, the coverage-guided fuzzing of ConFuzz is much more effective than Node.Fz at finding the same concurrency bugs.

Multithreaded programs: Many approaches and tools have been developed to identify concurrency bugs in multi-threaded programs. FastTrack [14], Eraser [33], CalFuzzer [18] aims to detect multi-threaded concurrency bugs like data races, deadlock. ConTest [10], RaceFuzzer [35] uses random fuzzing to generate varied thread schedules. These approaches apply to multi-threaded programs for detecting concurrency bugs such as atomicity violations and race conditions on shared memory and are not directly applicable to event-driven programs interacting with the external world by performing I/O. Systematic exploration techniques such as model checking attempt to explore the schedule space of a given program exhaustively to find concurrency bugs. CHESS [27] is a stateless model checker exploring the schedule space in a systematic manner. While exhaustive state space exploration is expensive and given a limited test time budget, ConFuzz explores broader input and schedule space, which is more likely to detect bugs.

Application domains: There are bug detection techniques to identify concurrency errors in client-side JavaScript web applications. WAVE [15], WebRacer [30] and EventRacer [32] propose to find concurrency bugs in client-side elements like browser's DOM and webpage loading through static analysis or dynamic analysis. Though client-side web apps are event-driven, these techniques are tuned for client-side key elements like DOM and web page loading which are not present in server-side like OCaml concurrent programs. Thus, the above approaches cannot be directly applied to event-driven OCaml applications. Android is another event-driven programming environment in combination with multi-threaded programming model. Several dynamic data race detectors [16,26,4] have been proposed for Android apps. These tools are tightly coupled with the Android system and target mainly shared memory races rather than violations due to I/O events.

9 Conclusions and Future Work

In this paper, we have presented a novel technique that combines QuickCheck-style property-based testing with coverage-guided fuzzing for finding concurrency bugs in event-driven programs. We implemented the technique in a tool called ConFuzz using AFL for coverage-guided fuzzing for event-driven OCaml programs written using the Lwt library. Our performance evaluation shows that coverage-guided fuzzing of ConFuzz is more effective and efficient than the random fuzzing tool Node.Fz in finding the bugs. We also show that ConFuzz can detect bugs in large and widely used real-world OCaml applications without having to modify the code under test. As future work, we are keen to extend ConFuzz to test shared memory multi-threaded applications.

References

1. American fuzzy lop (2020), https://lcamtuf.coredump.cx/afl

2. Async: Typeful concurrent programming (2020), https://opensource.janestreet.com/async/
3. Asynchronous javascript (2020), https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous
4. Bielik, P., Raychev, V., Vechev, M.: Scalable race detection for android applications. SIGPLAN Not. **50**(10), 332348 (Oct 2015). https://doi.org/10.1145/2858965.2814303, https://doi.org/10.1145/2858965.2814303
5. Chang, X., Dou, W., Gao, Y., Wang, J., Wei, J., Huang, T.: Detecting atomicity violations for event-driven node.js applications. In: Proceedings of the 41st International Conference on Software Engineering. p. 631642. ICSE '19, IEEE Press (2019). https://doi.org/10.1109/ICSE.2019.00073, https://doi.org/10.1109/ICSE.2019.00073
6. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ICFP '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266, https://doi.org/10.1145/351240.351266
7. Davis, J., Thekumparampil, A., Lee, D.: Node.fz: Fuzzing the server-side event-driven architecture. In: Proceedings of the Twelfth European Conference on Computer Systems. pp. 145–160. EuroSys '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3064176.3064188, https://doi.org/10.1145/3064176.3064188
8. Improving docker with unikernels: Introducing hyperkit, vpnkit and datakit (2020), https://www.docker.com/blog/docker-unikernels-open-source/
9. Dolan, S., Preston, M.: Testing with Crowbar. OCaml Workshop (2017)
10. Edelstein, O., Farchi, E., Goldin, E., Nir-Buchbinder, Y., Ratsaby, G., Ur, S.: Framework for testing multithreaded java programs. Concurrency and Computation: Practice and Experience **15** (2003)
11. epoll: I/o event notification facility (2020), http://man7.org/linux/man-pages/man7/epoll.7.html
12. Fisher, R.A., Yates, F.: Statistical tables: For biological, agricultural and medical research. Oliver and Boyd (1938)
13. Flanagan, C., Freund, S.N.: Fasttrack: Efficient and precise dynamic race detection. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 121–133. PLDI '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1542476.1542490, https://doi.org/10.1145/1542476.1542490
14. Flanagan, C., Freund, S.N.: Fasttrack: Efficient and precise dynamic race detection. SIGPLAN Not. **44**(6), 121133 (Jun 2009). https://doi.org/10.1145/1543135.1542490, https://doi.org/10.1145/1543135.1542490
15. Hong, S., Park, Y., Kim, M.: Detecting concurrency errors in client-side java script web applications. In: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation. p. 6170. ICST '14, IEEE Computer Society, USA (2014). https://doi.org/10.1109/ICST.2014.17, https://doi.org/10.1109/ICST.2014.17
16. Hsiao, C.H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C.L., Pokam, G.A., Chen, P.M., Flinn, J.: Race detection for event-driven mobile applications. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language

Design and Implementation. p. 326336. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594330, https://doi.org/10.1145/2594291.2594330
17. I/o completion ports (2020), https://docs.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports
18. Joshi, P., Naik, M., Park, C.S., Sen, K.: Calfuzzer: An extensible active testing framework for concurrent programs. In: Proceedings of the 21st International Conference on Computer Aided Verification. p. 675681. CAV '09, Springer-Verlag, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-642-02658-4_54
19. kqueue, kevent – kernel event notification mechanism (2020), https://www.freebsd.org/cgi/man.cgi?kqueue
20. Leroy, X.: The ZINC experiment : An Economical Implementation of the ML Language. Tech. Rep. RT-0117, INRIA (Feb 1990), https://hal.inria.fr/inria-00070049
21. Libev event loop (2019), http://software.schmorp.de/pkg/libev.html
22. libuv: Cross-platform asynchronous i/o (2020), https://libuv.org/
23. Liu, C., Zou, D., Luo, P., Zhu, B.B., Jin, H.: A heuristic framework to detect concurrency vulnerabilities. In: Proceedings of the 34th Annual Computer Security Applications Conference. p. 529541. ACSAC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3274694.3274718, https://doi.org/10.1145/3274694.3274718
24. Lwt: Ocaml promises and concurrent i/o (2020), https://ocsigen.org/lwt/5.2.0/manual/manual
25. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: Library operating systems for the cloud. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 461–472. ASPLOS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2451116.2451167, https://doi.org/10.1145/2451116.2451167
26. Maiya, P., Kanade, A., Majumdar, R.: Race detection for android applications. SIGPLAN Not. **49**(6), 316325 (Jun 2014). https://doi.org/10.1145/2666356.2594311, https://doi.org/10.1145/2666356.2594311
27. Musuvathi, M., Qadeer, S.: Chess: Systematic stress testing of concurrent software. In: Puebla, G. (ed.) Logic-Based Program Synthesis and Transformation. pp. 15–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
28. Node.js is a javascript runtime built on chrome's v8 javascript engine. (2020), https://nodejs.org/en/
29. Padhye, R., Lemieux, C., Sen, K.: Jqf: Coverage-guided property-based testing in java. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 398–401. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3339002, https://doi.org/10.1145/3293882.3339002
30. Petrov, B., Vechev, M., Sridharan, M., Dolby, J.: Race detection for web applications. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 251262. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254095, https://doi.org/10.1145/2254064.2254095
31. Poll system call (2019), http://man7.org/linux/man-pages/man2/poll.2.html

32. Raychev, V., Vechev, M., Sridharan, M.: Effective race detection for event-driven programs. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. p. 151166. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2509136.2509538, https://doi.org/10.1145/2509136.2509538
33. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391411 (Nov 1997). https://doi.org/10.1145/265924.265927, https://doi.org/10.1145/265924.265927
34. Select system call (2019), http://man7.org/linux/man-pages/man2/select.2.html
35. Sen, K.: Race directed random testing of concurrent programs. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1121. PLDI '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375584, https://doi.org/10.1145/1375581.1375584
36. Twisted: an event-driven networking engine written in python (2020), https://twistedmatrix.com/trac/
37. Verifit repository of test cases for concurrency testing (2020), http://www.fit.vutbr.cz/research/groups/verifit/benchmarks/
38. Vouillon, J.: Lwt: A cooperative thread library. In: Proceedings of the 2008 ACM SIGPLAN Workshop on ML. pp. 3–12. ML '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1411304.1411307, https://doi.org/10.1145/1411304.1411307
39. Wang, J., Dou, W., Gao, Y., Gao, C., Qin, F., Yin, K., Wei, J.: A comprehensive study on real world concurrency bugs in node.js. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. pp. 520–531. ASE 2017, IEEE Press (2017)
40. Yu, Y., Rodeheffer, T., Chen, W.: Racetrack: Efficient detection of data race conditions via adaptive tracking. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. pp. 221–234. SOSP '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1095810.1095832, https://doi.org/10.1145/1095810.1095832