Locating Concurrency Errors in Windows .NET Applications by Fuzzing over Thread Schedules

Filip Kliber and Pavel Parízek

Charles University, Prague, Czech Republic
kliber@d3s.mff.cuni.cz, parizek@d3s.mff.cuni.cz

Abstract. We present a new fuzzing technique for multithreaded C# programs running on the .NET platform. It is built upon the .NET Profiling library, supported by CLR (Common Language Runtime) on Windows, and uses configurable strategies for the fuzzing process. During execution of the subject program, the fuzzing algorithm controls thread scheduling and preemption through suspending and resuming threads at specific code locations that we call stop points. For the purpose of driving the fuzzing process, we have designed a hybrid systematic-random strategy that gradually finds yet unexplored thread schedules. Results of experiments with programs from the SCT benchmark collection show that our tool is able to find errors triggered by specific thread interleavings, and within practical time limits.

1 Introduction

Fuzzing has become a very popular approach to discovering inputs and configurations that may trigger runtime errors in software systems [8, 26]. It has been successfully used to detect critical security vulnerabilities. The basic idea of automated fuzzing is to execute the subject program repeatedly many times, each time with a different input, and monitor its behavior and output for errors. Usually, the first input can be selected randomly, so that an arbitrary execution is tested, and then other subsequent inputs are derived by the fuzzer algorithm with the goal of exploring alternative execution traces (e.g., different control-flow paths). The main practical benefit of fuzzing is the ability to generate especially inputs that represent corner cases not considered and expected by human developers.

A specific category of programs are those involving multiple concurrent threads of execution. It is well known that threads may interleave in many ways, under the control of a system scheduler, and some thread schedules (interleavings) may trigger a concurrency error state, such as deadlock or atomicity violation (data race), or assertion violations caused by inconsistent concurrent data updates. Concurrency errors are, in general, very hard to find, because they are exposed only by specific rare interleavings of threads' execution [14].

Various approaches to detecting runtime concurrency errors, and related bugs in the program source code, have already been proposed, including those based on model checking [7, 19] with partial order reduction [1], reduction to sequential programs and subsequent verification [11, 12], dynamic analysis [5, 18, 25], systematic concurrency testing [4, 6, 13, 21], and static program analysis [15, 16]. Many techniques and tools in each category have been developed so far. But all of these approaches have their well-known limitations and challenges, for example state explosion in the case of model checking, practical scalability and performance in the case of systematic concurrency testing, and (im)precision caused by abstraction in the case of techniques based on static analysis. For any realistic large and complex program (software system), there is really a huge number of possible ways in which execution of program threads can interleave, so then it is very hard to find some of the rare interleavings that may actually trigger some concurrency error at runtime.

Therefore it is no surprise that fuzzing is yet another promising approach for detecting concurrency-related bugs and vulnerabilities in multithreaded programs [2,9,10,24]. With this perspective of fuzzing applied to multithreaded programs, a thread schedule may be considered as the subject program's runtime configuration or specific input too. A scheduler in the operating system (e.g., Windows kernel) or virtual machine (such as .NET CLR) then represents a part of environment that influences the program's execution and outcome.

We have created a new fuzzing tool for multithreaded C# programs to be executed with the .NET platform runtime (CLR) on Windows systems. The tool uses a custom runtime analysis, built upon the .NET profiling library [27], together with a component responsible for driving the exploration of thread schedules. It exercises the behavior of subject programs under many different thread schedules, but with fuzzer-controlled thread preemption only at statements and code locations (method calls) configured by the user. Our main contribution is a new algorithm that gradually computes the set of distinct thread schedules to be explored, with the goals of achieving high coverage and trying to avoid repeated execution of the same thread interleaving multiple times. In each iteration of the main fuzzing loop, the algorithm yields the specific thread schedule to be followed. The key characteristic of our algorithm is that it navigates the fuzzer towards thread schedules that were not covered already, doing that with sufficient precision and reliability.

In this paper, we present (1) an overview of the fuzzing tool and the main algorithms it uses, (2) a list of related technical challenges that we have faced and our solutions, including the discussion of key design choices and general insights, and (3) experimental evaluation together with discussion of our experience and lessons learned.

The paper has the following structure. We provide an overview of the proposed fuzzing approach in Section 2, and technical details about main components of the fuzzing tool in Sections 3 and 4, respectively. Specifically, the main algorithms used by the fuzzer are described in Sections 2-4. Then we present results of experimental evaluation in Section 5, also discussing our observations. We compare our approach with related work in Section 6, and conclude with the discussion of high-level insights and possible directions of future work in Section 7.

2 Overview

First we describe the overall architecture of our fuzzing tool and introduce few key concepts. The tool consists of three main components:

– a runtime profiler that monitors execution of a subject program and watches for specific actions (e.g., calls of thread synchronization API methods),
– thread scheduling controller (manager) that is able to suspend and resume threads running within the .NET platform during execution of the program, and
– the actual fuzzing driver that implements our algorithm for exercising different thread schedules.

Input of the fuzzer tool includes (1) a binary program executable by the .NET runtime platform and (2) a user-defined set of program code locations where thread scheduling choices (preemption) should be triggered to cover interesting thread interleavings. These code locations are called *stop points* in our paper, and they are in particular calls of library procedures that implement synchronization operations (lock acquire, lock release, wait, notify, etc) and calls of application methods. In the rest of this section, we present the whole fuzzing process and tasks performed by each component of the tool.

Algorithm 1 shows the key parts of our algorithm for exercising different thread schedules within the fuzzing process. The algorithm maintains a list of execution traces (thread schedules) that were already explored, and keeps running until it cannot find any new thread schedule that has not been covered yet. Details about the function that yields new thread schedules are provided in Section 4. In each iteration, the fuzzer picks one new thread schedule sch and executes the subject program under the schedule, with the runtime profiler attached. At the start of the whole fuzzing process, when the list of explored traces is empty, an arbitrary thread schedule is selected randomly.

Algorithm 1 Algorithm for exercising thread schedules

```
Input: The input binary program subject to fuzzing, represented by the symbol $binProg$, and
  the set of pre-configured stop points represented by the symbol $cfgStopPoints$.
  $exploredTraces \leftarrow []$
  while DRIVER::EXISTSUNEXPLOREDSCHEDULE($exploredTraces$) do
      $sch \leftarrow$ DRIVER::GETNEWTHREADSCHEDULE()
      EXECUTEWITHPROFILER($binProg, cfgStopPoints, sch$)
      $exploredTraces \leftarrow exploredTraces \cup sch$
  end while

  function ONREACHEDSTOPPOINT($curTh$)
      MANAGER::SUSPENDTHREADS($curTh$)
      $thsToEnable \leftarrow$ DRIVER::CHOOSETHREADSTORESUME($sch$)
      MANAGER::RESUMETHREADS($thsToEnable$)
  end function
```

The runtime profiler watches execution of the program, observing especially method calls, and creates a log of the whole trace that is reported to the user in case an error is detected. When the current active thread calls a method that corresponds to a predefined stop point, the following actions are performed by the fuzzer (see the function ONREACHEDSTOPPOINT in Algorithm 1):

1. Thread manager suspends (”freezes”) some of the currently active threads, notably either just the current thread or all threads, depending on the configuration.
2. Fuzzing driver takes the set of suspended threads and from this set chooses, according to the schedule sch and the configured strategy, a subset that contains threads to become active at the given point.
3. Right after that, execution of all these selected threads is resumed (”thawed”) by the thread manager.

Execution of the program then continues until another stop point is reached. Technical details about suspending and resuming threads are provided in Section 3.

When the execution of the subject program with profiling finishes, the top-level fuzzing algorithm continues with another iteration where an alternative thread schedule is considered. The explored schedules differ by the sets of threads suspended and resumed at individual stop points. Section 4 gives more details about: selection of threads to become runnable (active), strategies that we designed, and the procedure for generating thread schedules (traces) not yet explored.

In the rest of this paper, we illustrate specific aspects of the whole process on the example program in Figure 1. It consists of two threads communicating through a shared buffer. The function `ComputeData` just performs some computation that produces a new value. Labels with the prefix `op_` are used below to identify the respective method call statements, which correspond to stop points.

```
interface Buffer() {
  Add(int v);
  int Remove();
}
```

```
static void Main() {
  th1 = new Thread(thread1);
  th2 = new Thread(thread2);
  th1.start(); th2.start();
}
```

```
thread1() {
  op_11:
    int v = ComputeData();
  op_12:
    buffer.Add(v);
}
```

```
thread2() {
  op_21:
    int r = buffer.Remove();
  op_22:
    Console.WriteLine(r);
}
```

Fig. 1: Example program with two threads

If threads in the program execute concurrently, there are six possible ways they can interleave, displayed in Figure 2. These six interleavings correspond to thread schedules that should be explored to achieve full coverage.

The next two sections provide details about steps of the whole fuzzing process. In particular, we discuss our technical design decisions and challenges that we faced.

3 Runtime Profiler and Thread Manager

The runtime profiler component uses the low-level Microsoft Profiling API [27] for .NET applications, which is provided by the .NET runtime platform, to monitor pro-

```
op_11    op_11    op_11    op_21    op_21    op_21
op_21    op_12    op_21    op_11    op_22    op_11
op_22    op_21    op_12    op_22    op_11    op_12
op_12    op_22    op_22    op_12    op_12    op_22
```

Fig. 2: All possible interleavings of threads in the example program

gram execution and observe interesting events. However, this is why our fuzzing tool targets just applications running on Windows systems, because the Profiling API is supported only in releases of both .NET Framework and .NET Core on the Windows platform. When the fuzzing process is about to start, the profiler is attached to the subject program. During execution of the program, it receives notifications about relevant events from the .NET Common Language Runtime (CLR).

We have implemented and configured our profiler component such that it gets notified about calls of methods that represent possible stop points. In addition, the profiler is notified about certain system-level thread synchronization- and concurrency-related events, such as thread creation, and monitors the dynamic stack trace for each thread.

The thread manager, as already indicated, controls the actual interleaving of individual threads during program execution by the means of suspending and resuming threads at stop points. In this process, it tries to follow the given thread schedule, selected by the fuzzing driver component, as closely as possible.

One of the key design decisions behind our fuzzing algorithm was to enable *stop points* only at method call statements. This is, however, sufficient in practice because of the way the .NET platform works. All the relevant and interesting events from the perspective of concurrency fuzzing are represented by method calls — notably, accesses to properties (i.e., to object fields via getters/setters) and even basic thread synchronization operations (lock acquire, release, etc). Users define stop points by the name of the method and its owner class. We assume that, in practice, users will mark as possible stop points especially the calls of application methods that may access data shared by multiple concurrent threads.

Originally we have considered stop points of two kinds, strong and weak. The difference is that only the current active thread is suspended at a weak stop point, while all runnable application threads are suspended at a *strong stop point*. Nevertheless, based on our initial experiments, we have found that strong stop points are superior, meaning that usage of weak stop points does not have any benefits with respect to coverage of thread interleavings. Therefore, from now on, by the term ”stop point” we always refer to a strong stop point.

We had to define the procedure for suspending all threads very carefully, in particular to avoid unsafe states caused by preempting some application threads while they are executing within the kernel space or system libraries. The problem is that, when one application thread reaches a strong stop point, other threads may be located anywhere in managed or unmanaged code, executing syscalls or native library procedures, manipulating shared kernel resources and holding the respective global locks (e.g., when allocating heap memory on Windows), and so on. We have considered and implemented two solutions that we describe here:

– One solution, that we claim to be safe, is to stop threads only at well-defined locations in the application code — specifically, at the boundaries of application methods. When some application thread reaches a stop point, thread manager asks all the other application threads to stop, by setting a flag introduced for this purpose. The flag is checked by the runtime profiler handlers for method entry and exit events. In practice, thread manager just has to wait until every application thread is out of the kernel space (or finishes a library call), and reaches the nearest entry or exit point in the code of some application method. To be more specific, when thread T is executing a library method or syscall invoked by the application method M, the manager waits until T reaches either (i) the entry of some other application method (including getters and setters for properties) called from M or (ii) exit from M, whatever happens first.
– Another option is to stop threads ”by force” at their current locations, even if they reside in the kernel space or execute a system library procedure, and solve the problem on a different level. In particular, to avoid deadlocks in the Windows heap allocator (when one thread would be stopped inside the implementation of std::malloc while holding the global allocation lock), it is necessary to use a custom allocator that manipulates a fixed memory pool.

However, only the second solution is now enabled in the profiler and thread manager.

An important related technical detail is the usage of a special service thread within the profiler and the manager, a thread that performs the following steps repeatedly in an infinite loop: waits until some application threads are suspended, passes an updated list of all suspended threads to the fuzzing driver component, and then asks the driver to decide which of the suspended threads should be resumed. The action of resuming threads is therefore also triggered by the special service thread. Supported strategies for deciding which threads to resume are described in Section 4. It is necessary to use a separate service thread for this purpose, because the runtime profiler exists directly inside the CLR and has no thread allocated on its own. When some application thread triggers an event watched-for by the profiler, the corresponding event handler defined by our profiler is executed by that same application thread.

Note also that the fuzzing driver can select no threads (an empty set) to be resumed at a given stop point and moment of time. In that case, the service thread just continues and asks the driver again, periodically, every few milliseconds.

4 Fuzzing Driver: Exploring Thread Schedules

The fuzzing driver component of our tool has two main duties: (1) recording information about executed traces and (2) selecting thread schedules to be explored.

First we provide important details regarding the management and recording of traces. A trace is collected by the runtime profiler for every execution of the subject program. It is a list of thread scheduling decisions made at stop points, where each entry in the list contains the following information: an ID of the application thread that was suspended, code location of the stop point (method signature and owner class name), and IDs of all threads that were selected by the driver to be resumed. The driver stores the set of all traces (sequences of thread scheduling choices) already explored, and uses this knowledge in future iterations of the fuzzing process to navigate program execution towards alternative thread schedules.

We mentioned before that we designed the algorithm for selecting thread schedules with two goals in mind: (1) to explore as many unique thread schedules as possible within available resources, including the time budget, and (2) to avoid repeated exploration of already visited thread interleavings.

Control over schedules is realized through selection of threads to resume execution at stop points. During the first execution of the subject program in a fuzzing session, when there are no saved explored traces, the choice of threads is completely random. In the case of all other executions, the individual scheduling choices are driven by a configurable strategy for exploring many distinct traces. We support two automated strategies, *random* and *systematic*, and interactive user control too.

The random strategy selects the thread to be resumed from a list of all suspended threads completely at random. In addition, it can be configured to insert specific delays between the moments when threads are suspended and resumed, further affecting how the final schedule looks like. A consequence of random selection is that some already explored thread schedules may be visited repeatedly (multiple times).

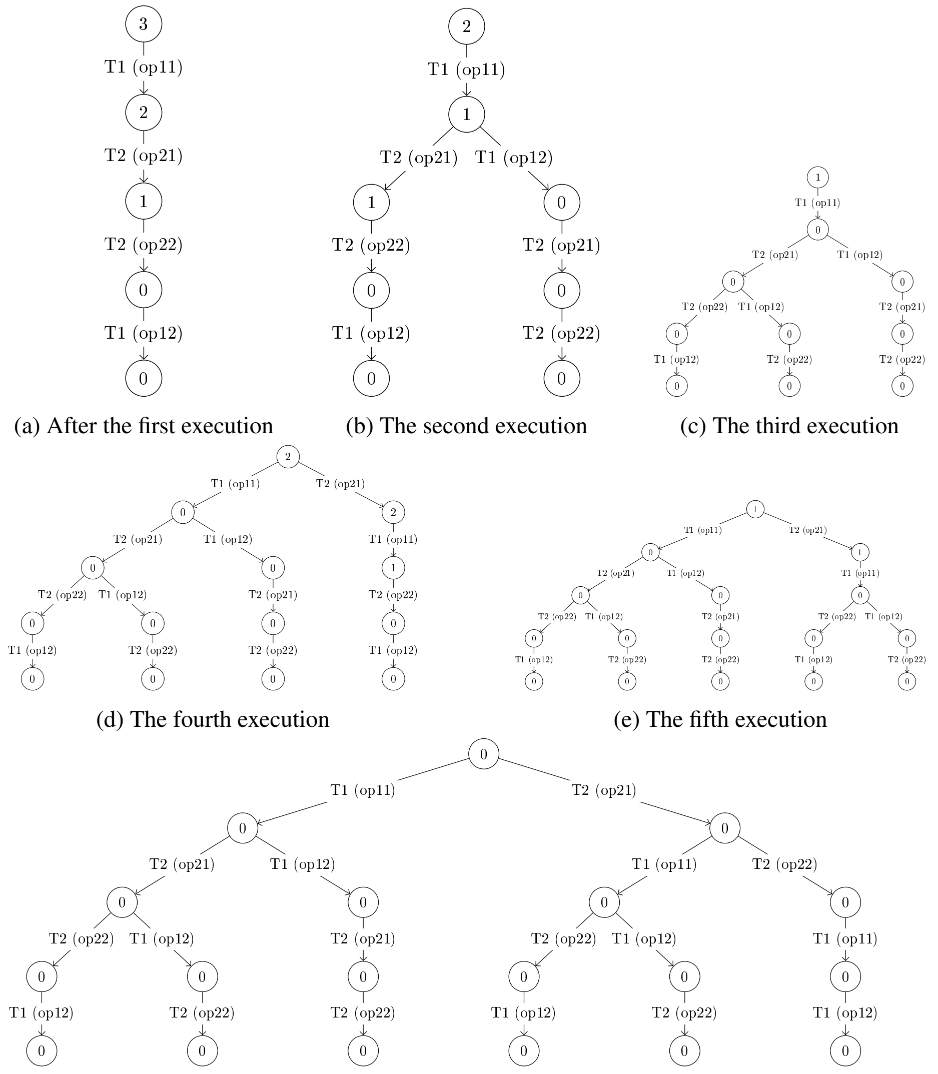
We have also created the interactive user driver, which human developers can use through a command-line interface, mainly for debugging and error reproduction. Users can simply resume threads by providing their IDs on the command line. A standard interactive debugger, when attached to the application, can be used to inspect the program state and confirm hypotheses about root causes of the observed errors.

4.1 Systematic Strategy

The proposed systematic strategy for exploring thread schedules is designed around a special data structure, which we call a *scheduling graph*. It is created at the start of each iteration of the fuzzing process, and captures the set of thread schedules corresponding to all previously executed traces. Nodes of the scheduling graph represent stop points and edges represent taken scheduling decisions. Every node is labeled with an integer value that expresses the total number of possible *distinct* unexplored thread schedules that could be executed from the stop point associated with the node. These values are used to find not-yet-explored alternative thread schedules (traces). Edges are labeled with thread IDs and (sequences of) executed program statements.

We illustrate the concept of scheduling graphs on our example program (Figure 1) and the list of possible thread schedules (Figure 2). Assuming that each execution of the program by the fuzzer takes a different schedule, in the order presented in Figure 2 from left to right, then Figure 3 shows what the scheduling graphs created after the respective iterations (executions) of the fuzzing process would look like.

Having such a graph after some iteration of the fuzzing process, a new thread schedule to be explored in the next round is derived simply by depth-first traversal of the graph from the root node along a path over nodes with non-zero labels, representing thread scheduling choices with possibly unexplored options. The following cases are considered in each step of the traversal:



(a) After the first execution (b) The second execution (c) The third execution

(d) The fourth execution (e) The fifth execution

(f) After the sixth execution

Fig. 3: Scheduling graphs for execution traces of the example program in Figure 1

– If the current node is labeled with a non-zero value, and it has at least one child node with a non-zero value, then traversal proceeds by moving to one of the child nodes with a non-zero value attached.

– If the current node is labeled with a non-zero value, but all of its existing child nodes in the graph are labeled with zero, then the search produces a new thread schedule by choosing a thread that is not yet covered by any edge leading from the

Algorithm 2 Producing new thread schedules

```
Input: List of traces explored in previous iterations, $exploredTraces = [T_1, T_2, \ldots, T_n]$.

  function ExistsUnexploredSchedule($exploredTraces$)
      $graph \leftarrow$ CreateGraph($explordTraces$)
      return $graph.root.value > 0$
  end function

  function GetNewThreadSchedule
      $schedule \leftarrow$ ExtendScheduleByDFS($[]$, $graph.root$, $graph$)
      return $schedule$
  end function

  function ExtendScheduleByDFS($sch, cn, G$)
      if $cn.value > 0 \land \exists w$ s.t. $(cn, w) \in G.edges \land w.value > 0$ then
          $sch \leftarrow sch$ ++ $(cn, w).label$
          ExtendScheduleByDFS($sch, w, G$)
      end if
      if $cn.value > 0 \ \land \neg\exists(cn, w) \in G.edges$ s.t. $w.value > 0$ then
          $sch \leftarrow sch$ ++ $th$ s.t. $\forall(cn, w) \in G.edges \cdot (cn, w).label \neq th$
      end if
      return $sch$
  end function
```

node. See, for example, the root node of a graph in Figure 3c. Traversal ends in this case, and the new thread schedule is returned.

– If the current node is labeled with zero, traversal backtracks.

Specifically, if the root node of the whole graph is labeled with zero, all thread schedules were explored and the fuzzing process terminates. The procedure for deriving new thread schedules is defined more formally in Algorithm 2. The helper function ExtendScheduleByDFS attempts to extend the current schedule sch with thread choices represented by edges leading from the given node cn.

The procedure to create a scheduling graph for a set of already explored traces (thread schedules) is formalized in Algorithm 3. It is a key part of our systematic driver strategy for the fuzzer. The helper function CreatePathForTrace extends the graph with a path that represents the given trace, and returns a leaf node on the newly created path in the scheduling graph. The helper function UpdateValuesForNodes iteratively updates the attached value for each node on the new path, in the direction from a leaf node to the root.

4.2 Discussion and Remarks

Here we discuss important properties of the fuzzing approach, especially with the systematic driver strategy, that we observed when performing initial experiments during the development.

Algorithm 3 Construction of scheduling graphs

```
Input: List of traces collected from previous iterations.
Output: Scheduling graph $G = (nodes, edges)$.

  function CreateGraph($[T_1, T_2, \ldots, T_n]$)
      $root \leftarrow$ NewNode(), $root.value \leftarrow 0$
      $G \leftarrow (nodes \leftarrow root, edges \leftarrow \emptyset)$
      for $T_i \in T_1, \ldots, T_n$ do
          $ln \leftarrow$ CreatePathForTrace($G, T_i$)
          UpdateValuesForNodes($G, ln$)
      end for
  end function

  function CreatePathForTrace($G, T_i$)
      $v \leftarrow G.root$
      for $th \in T_i$ do
          if $\neg\exists w$ s.t. $(v, w) \in G.edges \wedge (v, w).label = th$ then
              $w \leftarrow$ NewNode(), $w.value \leftarrow 0$, $G.nodes \leftarrow G.nodes \cup \{w\}$
              $e \leftarrow (v, w)$, $e.label \leftarrow th$, $G.edges \leftarrow G.edges \cup \{e\}$
          end if
          $v \leftarrow w$
      end for
      return $v$
  end function

  function UpdateValuesForNodes($G, v$)
      repeat
          $\exists!(w, v) \in G.edges$
          $w.value \leftarrow 0$
          $X \leftarrow \{x \mid (w, x) \in G.edges\}$
          for $x \in X$ do
              $w.value \leftarrow w.value + x.value$
          end for
          $m \leftarrow max($CountEnabledThreads$(x) \mid x \in X)$
          $w.value \leftarrow w.value + max(0, m - |X|)$
          $v \leftarrow w$
      until $v = G.root$
  end function
```

A practical limitation of the presented approach to systematic exploration of thread schedules is that it requires stable thread IDs across program executions, to be able to compare traces explored in different iterations. Using system thread IDs, provided by the OS kernel or .NET CLR, is not possible, because they change in each execution of the subject program. In our prototype implementation of the fuzzer tool, we have decided to use abstract thread IDs, integer numbers starting from 1 and incremented for every created thread. However, this solution does not work for programs that use a thread pool, such as in the case of .NET Task Parallel Library, where it is not certain that the same thread from a pool will handle a specific Task object in each execution.

The behavior of our fuzzing algorithm in practice also greatly depends on the set of pre-defined code locations to be used as stop points. Consider this scenario. The driver strategy resumes a small subset of suspended threads at a particular stop point, for example just one thread, and no other stop point is hit during continuation of that single thread's execution. In that case, execution of the whole program may end in a deadlock state, when the single thread finishes its execution and other threads are suspended. Users need to be aware of the fact that many iterations (runs) of the fuzzer, on many thread schedules (traces), may finish by such a spurious deadlock.

We recommend, based on our experience so far, to define a sufficiently large set of code locations as possible stop points. The stop points can be viewed as breakpoints and preemption locations in executions of individual threads. A small set of defined stop locations may result in a relatively high ratio of pathological thread schedules.

5 Experiments

We have created a prototype implementation of the proposed fuzzing approach, described in previous sections, in an open source tool. Complete source code is available in the repository `https://github.com/d3sformal/threadfuzzer-net`.

We evaluated the practical usefulness of our tool based on these important criteria: its ability to find runtime errors (e.g., assertion violations, crashes and deadlocks) triggered by specific thread interleavings, whether it has good performance, and coverage of distinct thread schedules. In this section, we describe the setup of our experiments and present the results of measurements, and then we dicuss main observations.

For the purpose of this evaluation, we performed experiments with programs from the SCTBench collection [23, 28], which is quite a representative set of benchmarks regarding usage of concurrency. But first we had to translate all programs to the C# language, and modify some of the benchmark programs by refactoring class fields into class properties with automatically generated get and set methods. This was needed because our tool currently supports stop points only at method calls. Most of the programs in SCTBench are small, containing just few methods, so our configuration of stop point locations for each benchmark contains all of its applications methods, together with methods of the library class SemaphoreSlim.

The fuzzing tool can be, in practice, applied only to a standalone executable program, e.g. with the Main method, or to a partial program (library) with test cases. Additionally, it makes sense to use the fuzzer with programs that contain some checks for runtime error states during their execution. We reflect this in our evaluation by running the fuzzer on existing (unit) concurrency tests.

To properly analyze the benefits and limitations of our fuzzing tool, we ran experiments for three different configurations of the fuzzer, and for plain executions (without the profiler attached) as the baseline. In case of the baseline configuration, thread scheduling is fully controlled by the OS and .NET CLR. The other configurations are the following:

- Usage of the random strategy in the fuzzing driver component, denoted by the term *Random Fuzzing* in the table with data.
- Systematic strategy for the driver in the variant that always selects the first unexplored thread schedule found in the graph by DFS, denoted as *Systematic First* in the table with results.
- Systematic driver strategy combined with random selection of an unexplored thread schedule from all that could be found in the scheduling graph (*Systematic Random*).

All experiments were done on a personal computer running Windows 10 Pro 64-bit (10.0, Build 19045) with processor AMD Ryzen 7 PRO 5850U and 16 GB of memory. We have provided all that is needed to run experiments (scripts, benchmark programs, etc) in the public repository with our implementation.

Table 1 presents the results of our experiments for all benchmarks and configurations. In each experiment, i.e. for each pair of configuration and benchmark, we ran our fuzzing tool with a hard upper limit of 150 iterations, meaning that at most 150 thread schedules were explored. The actual number of explored unique thread schedules varies just very slightly over the set of all runs; is in the range 145-150 for almost all experiments. We also set a time limit for each iteration of the fuzzing algorithm, that means for a single execution of the subject program with profiler, to 20 seconds. Values of these metrics are reported in the table for each benchmark and experiment:

- DoC, Degree of Concurrency, which is a statically approximated number of OS threads running during program execution.
- Lines of code, LoC, and size of the program code in kB.
- The percentage of runs that ended with an error detected by the fuzzer.
- Percentage of runs that timed out, typically after reaching a deadlock state where some applications threads are suspended and unable to finish their execution.
- Average execution time for all non-timedout iterations.

Data in Table 1 show that those configurations of the fuzzing tool (exploration strategies) that involve random choice have the greatest ability to find errors — indicated by the high percentages of runs where an error was detected and reported. Looking at results for these strategies, namely ”Random Fuzzing” and ”Systematic Random”, we observe that percentages of explored thread schedules (within the bound of 150 runs) that trigger errors are comparable over the set of all benchmarks. The systematic strategy in the select-first variant discovers less error-triggering thread schedules, because it may reach the limit of 150 runs while exploring just one segment of the scheduling graph (and thus executing many thread schedules with a common prefix).

We also note that the percentage of timed out runs (iterations of the fuzzing algorithm) across the set of all experiments is very low. It means that just a small number of thread schedules ended in a deadlock state.

On the other hand, there is one notable exception to the general observed pattern — results obtained for the benchmark ”TwoStage*100”, which has an extremely high degree of concurrency. The scheduling graph is really big in this case, so that traversal of the graph causes the fuzzer to run out of the time limit.

Benchmark Name	DoC	LoC (kB)	Metrics	Baseline	Random Fuzzing	Systematic First	Systematic Random
Account	4	51 (1.2)	Violated	2.7 %	28.7 %	14.0 %	30.0 %
			Timed out	0.0 %	0.0 %	4.0 %	1.3 %
			Time/Iter	33.8 ms	210.3 ms	233.6 ms	214.5 ms
BluetoothDriver	2	74 (1.7)	Violated	0.0 %	6.0 %	0.0 %	1.3 %
			Timed out	0.0 %	0.0 %	0.0 %	0.0 %
			Time/Iter	33.9 ms	550.8 ms	676.2 ms	532.1 ms
Carter01	5	67 (1.3)	Violated	0.0 %	0.0 %	0.0 %	0.0 %
			Timed out	0.0 %	12.0 %	7.3 %	20.7 %
			Time/Iter	29.8 ms	213.6 ms	213.2 ms	200.4 ms
CircularBuffer	3	86 (2.2)	Violated	0.0 %	59.3 %	6.0 %	69.3 %
			Timed out	0.0 %	0.0 %	1.3 %	1.3 %
			Time/Iter	29.7 ms	476.6 ms	226.5 ms	558.3 ms
Deadlock01	3	42 (1.0)	Violated	0.0 %	0.0 %	0.0 %	0.0 %
			Timed out	0.7 %	2.7 %	8.0 %	5.3 %
			Time/Iter	28.5 ms	201.3 ms	202.2 ms	196.1 ms
Lazy01	4	40 (0.9)	Violated	0.7 %	66.0 %	48.0 %	59.3 %
			Timed out	0.0 %	0.0 %	0.7 %	2.3 %
			Time/Iter	32.3 ms	571.3 ms	466.0 ms	538.5 ms
Queue	3	99 (2.6)	Violated	98.7 %	100.0 %	84.7 %	98.7 %
			Timed out	0.0 %	0.0 %	2.0 %	1.3 %
			Time/Iter	48.5 ms	763.0 ms	290.5 ms	748.6 ms
Reorder3	17	83 (2.3)	Violated	0.0 %	6.0 %	0.0 %	8.0 %
			Timed out	0.0 %	0.0 %	0.0 %	1.3 %
			Time/Iter	30.9 ms	464.4 ms	596.6 ms	473.4 ms
Reorder10	52	83 (2.3)	Violated	0.0 %	3.3 %	7.3 %	2.7 %
			Timed out	0.0 %	0.0 %	7.3 %	3.3 %
			Time/Iter	29.8 ms	3416.5 ms	5840.3 ms	3431.6 ms
Stack	3	73 (1.6)	Violated	1.3 %	61.3 %	0.0 %	44.7 %
			Timed out	0.0 %	0.0 %	0.0 %	0.7 %
			Time/Iter	34.9 ms	848.0 ms	821.3 ms	828.3 ms
TokenRing	5	57 (1.3)	Violated	8.7 %	10.7 %	9.3 %	15.3 %
			Timed out	0.0 %	0.0 %	2.7 %	0.0 %
			Time/Iter	29.5 ms	208.8 ms	202.7 ms	210.7 ms
TwoStage100	101	71 (1.9)	Violated	0.0 %	0.0 %	0.0 %	0.0 %
			Timed out	0.0 %	100.0 %	9.3 %	100.0 %
			Time/Iter	30.0 ms	N/A	17032.8 ms	N/A
TwoStageSmall	3	89 (2.3)	Violated	0.0 %	5.3 %	0.0 %	0.7 %
			Timed out	0.0 %	0.0 %	2.0 %	0.7 %
			Time/Iter	29.0 ms	284.5 ms	310.7 ms	283.5 ms
WrongLock	9	61 (1.7)	Violated	0.0 %	13.3 %	13.3 %	20.7 %
			Timed out	0.0 %	0.0 %	8.7 %	4.0 %
			Time/Iter	29.8 ms	211.4 ms	218.4 ms	223.8 ms

Table 1: Results for all configurations and benchmarks

6 Related Work

Lot of research has already been done on various techniques for detecting bugs and errors related to concurrency in software, including also techniques for fuzzing multithreaded programs that were published recently. Here we compare our approach to several closely related techniques with similar goals and ideas.

A popular way of detecting concurrency bugs is through systematic unit testing. The Coyote tool [4] provides infrastructure for writing and running concurrency unit tests for .NET. It focuses, in particular, on the task-based asynchronous programming model that is very popular in the context of C#, and therefore has a limited ability to detect low-level data races. Unlike our approach, which uses .NET Profiling library to get notifications about relevant events and to control thread scheduling, Coyote instruments the subject binary program with code that enables the Coyote engine to control execution and thread scheduling.

Search for concurrency errors driven by random choice is another popular approach, implemented by many techniques and tools. For example, Sen [21] has proposed an algorithm that uses random sampling to cover the whole space of partial orders (thread schedules) more uniformly, and later proposed a technique for detecting low-level races where the random testing (selection) of thread schedules to be explored in directed by information about possible data races that is provided by dynamic analysis [22]. The key idea of [22] is the following. Simple random testing is performed most of the time, with one exception — when the next statement to be executed in the current thread belongs to some candidate pair of racy statements, execution of that statement is delayed until the other statement from the pair is about the be executed in some other thread, in this way checking whether the two statements in a candidate pair can really be executed concurrently. The follow-up work by Park and Sen [17] focuses on detecting higher-level atomicity violations through combination of dynamic analysis with random search and delaying of lock operations. However, all these techniques detect concurrency errors (deadlocks, races, etc), while our approach supports detection of general errors (violated assertions, crashes, failed tests, etc) triggered by specific thread interleavings.

Many other algorithms for search over the space of possible thread schedules, in the context of systematic concurrency testing, have been published so far. Thomson et al. [23] present results of an empirical study that compares the performance of selected algorithms, including the following: basic depth-first search, preemption bounding, delay bounding, and controlled random scheduler. Our fuzzing tool supports most of these algorithms, either directly or through user configuration. The basic depth-first search corresponds to our systematic (first) strategy. Preemption bounding can be simulated through configuring the stop points in a specific way. Delay bounding then corresponds to usage of weak stop points combined with a timeout (bound) on resuming threads. Finally, the controlled random scheduler algorithm corresponds to our systematic random strategy with a strong stop point enabled at every method call statement.

Table 2 contains the results of additional experiments that we performed for the purpose of comparison with specific related techniques. We present data for the configuration of our fuzzer that corresponds to the delay-bounding algorithm. Data in both tables show that the configurations ”systematic random” and ”delay bounding” have

Benchmark	Delay Bounding		
	Violated	Timed out	Time/Iter
Account	22.7 %	0.0 %	1039.9 ms
BluetoothDriver	2.7 %	0.0 %	1108.9 ms
Carter01	0.0 %	94.0 %	1704.2 ms
CircularBuffer	68.0 %	0.0 %	2474.1 ms
Deadlock01	0.0 %	57.3 %	1110.8 ms
Lazy01	50.7 %	0.0 %	1227.4 ms
Queue	98.0 %	0.0 %	2640.3 ms
Reorder3	0.7 %	0.0 %	3123.7 ms
Reorder10	7.3 %	0.0 %	4882.4 ms
Stack	37.3 %	0.0 %	4219.3 ms
TokenRing	25.3 %	0.0 %	1237.9 ms
TwoStage100	0.0 %	100.0 %	n/a
TwoStageSmall	8.7 %	0.0 %	1213.8 ms
WrongLock	29.3 %	2.7 %	2751.8 ms

Table 2: Results for the delay-bounding algorithm

roughly the same ability to find errors, measured as the percentage of runs that ended with an error detected by the fuzzer, but ”systematic random” is significantly faster from the two. This also confirms an observation, reported in [23], that search strategies based on random choice perform surprisingly well. Another empirical study with similar observations has been published by Rungta and Mercer [20].

The fuzzing techniques that explicitly support concurrent software extend the well-established approaches by recording and using concurrency-related information for the purpose of exercising program behavior under different thread schedules, in particular to detect vulnerabilities triggered by concurrent execution of threads. But they are not dedicated primarily on detecting concurrency errors (deadlocks, races). Chen et al. [2] developed MUZZ, a fuzzing technique that computes program inputs with the goal of achieving higher coverage of program behaviors (execution paths) in the multithreaded context. It targets scenarios where the execution of specific control-flow paths in the code of individual threads depends on the input values, exercising both input-dependent and also thread interleaving-dependent paths. For this purpose, MUZZ performs thread-aware instrumentation of program code, with special handling of thread scheduling-related operations (e.g., lock, unlock, and fork), and uses the collected data to prioritize exploration of traces (executions) that correspond to previously-unseen thread interleavings. Jiang et al. [9] developed CONZZER, a framework for concurrency fuzzing that introduces two new features: (1) more precise coverage metric based on tracking runtime calling contexts, and (2) control of thread scheduling based on inserting breakpoints. Knowledge of runtime calling contexts is used by CONZZER to identify pairs of methods, whose concurrent execution should be tested. The breakpoints, inserted before-hand (statically) by code instrumentation, are used to enforce actual concurrent execution of specific method pairs through careful management of time delays at runtime. This idea of using breakpoints and injected delays is similar to our stop points, where threads are suspended and resumed. Wolff et al. [24] proposed another fuzzing technique for multithreaded programs, which avoids redundant exploration of similar thread interleavings through an approach inspired by partial order reduction used in model checking. The key idea is to explore just one thread interleaving from each set of thread interleavings that are equivalent in terms of observable sequences of memory access events and values read by invididual threads at specific code locations.

7 Conclusion

The proposed approach to fuzzing multithreaded programs achieves high coverage of thread schedules and has a reasonable performance, as the results of our evaluation show. In particular, it is able to find errors triggered by specific thread interleavings quite efficiently, within practical limits on time and memory. For all these reasons, we believe that it is useful in practice. Still, there is a large space for extensions, improvements of user experience, and optimizations. Here we outline several directions for future work.

In the current version of our prototype implementation, the set of stop point locations is defined manually. This could be, at least partially, automated with the help of static analysis of the application program code and usage of information provided by dynamic analysis running on-the-fly in scope of the fuzzing process. The SharpDetect tool [3] with appropriate plugins could be used for the dynamic analysis. Our prototype implementation could be extended with additional functionality too, such as with (1) the ability to fuzz-test programs compiled specifically for the AMD64 platform and (2) support for the very recent versions of the .NET platform.

While looking for available concurrency benchmarks, we have also noticed that there is no established benchmark suite of programs written in the C# language. Already published research papers refer to individual bugs from all sorts of different software packages. We believe that having a diverse collection of benchmark programs in C# for .NET would foster the future research. Such collection of benchmarks should contain also few large programs, to facilitate evaluation of scalability. In the specific case of our fuzzing tool, a challenge related to analyzing large programs with many threads would be the specification of a right set of stop points.

Acknowledgments. This work was partially supported by the Czech Science Foundation project 23-06506S and partially supported by the Charles University institutional funding project SVV 260698.

References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal Dynamic Partial Order Reduction. In Proceedings of POPL 2014, ACM.
2. H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y Li, H. Wang, and Y. Liu. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In Proceedings of USENIX Security 2020.
3. A. Cizmarik and P. Parízek. SharpDetect: Dynamic Analysis Framework for C#/.NET Programs. In Proceedings of RV 2020, LNCS 12399.

4. P. Deligiannis, A. Senthilnathan, F. Nayyar, C. Lovett, and A. Lal. Industrial-Strength Controlled Concurrency Testing for C# Programs with Coyote. In Proceedings of TACAS 2023, LNCS 13994.
5. C. Flanagan and S.N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In Proceedings of PLDI 2009, ACM.
6. P. Fonseca, R. Rodrigues, and B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In Proceedings of OSDI 2014, USENIX.
7. P. Godefroid. Software Model Checking: The VeriSoft Approach. Formal Methods System Design, 26(2), 2005.
8. P. Godefroid. Fuzzing: Hack, Art, and Science. Communications of the ACM, 63(2), 2020.
9. Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In Proceedings of NDSS 2022, The Internet Society.
10. Y. Ko, B. Zhu, and J. Kim. Fuzzing with Automatically Controlled Interleavings to Detect Concurrency Bugs. Journal of Systems and Software, vol. 191, 2022.
11. A. Lal and T.W. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In Proceedings of CAV 2008, LNCS 5123.
12. S. La Torre, P. Madhusudan, and G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In Proceedings of CAV 2009, LNCS 5643.
13. M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In Proceedings of PLDI 2007, ACM.
14. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs, In Proceedings of OSDI 2008, USENIX.
15. M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In Proceedings of PLDI 2006, ACM.
16. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In Proceedings of ICSE 2009, IEEE CS.
17. C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In Proceedings of FSE 2008, ACM.
18. S. Park, R.W. Vuduc, and M.J. Harrold. Falcon: Fault Localization in Concurrent Programs. In Proceedings of ICSE 2010, ACM.
19. I. Rabinovitz and O. Grumberg. Bounded Model Checking of Concurrent Programs. In Proceedings of CAV 2005, LNCS, vol. 3576.
20. N. Rungta and E. Mercer. Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs. In Proceedings of PADTAD 2009, ACM.
21. K. Sen. Effective Random Testing of Concurrent Programs. In Proc. of ASE 2007, ACM.
22. K. Sen. Race Directed Random Testing of Concurrent Programs. In Proceedings of PLDI 2008, ACM.
23. P. Thomson, A.F. Donaldson, and A. Betts. Concurrency Testing Using Controlled Schedulers: An Empirical Study. ACM Transactions on Parallel Computing, 2(4), 2016.
24. D. Wolff, Z. Shi, G.J. Duck, U. Mathur, and A. Roychoudhury. Greybox Fuzzing for Concurrency Testing. In Proceedings of ASPLOS 2024, ACM.
25. A. Yoga, S. Nagarakatte, and A. Gupta. Parallel Data Race Detection for Task Parallel Programs with Locks. In Proceedings of FSE 2016, ACM.
26. A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. The Fuzzing Book. CISPA Helmholtz Center for Information Security, `https://www.fuzzingbook.org/` (accessed in February 2025).
27. .NET Profiling API (library), `https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/` (accessed in February 2025).
28. SCTBench: A collection of benchmarks, `https://github.com/mc-imperial/sctbench` (accessed in February 2025).