# Taming Deep Concurrency Bugs With ConFu

Han Liu, Zhiqiang Yang, Yue Zhao, Ying Fu, Jian Gao, Yixiao Yang, Yu Jiang, Jiaguang Sun
Institute of System and Software Engineering, School of Software
Tsinghua University, Beijing, China
{liuhan2017, jy1989}@mail.tsinghua.edu.cn

## ABSTRACT

Concurrency bugs in shared-memory multithreaded programs are notoriously hard to detect and reproduce. The major challenge lies in the exploration on the large state space, *i.e.*, interleaving and path space. Existing concurrency testing methods either can hardly scale on complex multithreaded programs (*e.g.*, systematic testing and symbolic analysis based), or hit the bug with little probability (*e.g.*, probabilistic scheduling based), thus are insufficient.

To address the challenge, we have developed a dynamic analyzer called ConFu and instantiated it for concurrent JVM software (*e.g.*, multithreaded programs, server-side applications, microservices *etc.*). Generally, ConFu takes JVM class files as input and reports a collection of bugs found in the detection, *e.g.*, crashes, data races, deadlocks *etc.*. The key insight behind ConFu is a combination of guided schedule fuzzing and symbolic trace analysis techniques. While fuzzing enables an efficient exploration of thread interleaving space, symbolic trace analysis further identifies previously uncovered program paths to be fuzzed. In the preliminary evaluation, ConFu demonstrated its potential in finding bugs more efficiently and digging out hidden ones as well. ConFu is available at https://github.com/njaliu/ConFu-release. A demo video is at https://youtu.be/R2YCyOrMPd4.

## 1 INTRODUCTION

Concurrency bugs, *e.g.*, data races [5], atomicity violations [8], ordering violations [2] and deadlocks [13], are widely rooted in shared-memory multithreaded programs [11]. Even worse, they are notoriously hard to detect and reproduce. With the rapid growth of real-world applications in terms of size and complexity, the impact of concurrency bugs gets further amplified. Unlike bugs in sequential programs which are commonly triggered by specific inputs, concurrency bugs usually manifest themselves on a specific thread interleaving (*i.e.*, ordering of operations among different threads). Typical bug-triggering operations include shared-memory accesses and various forms of synchronizations (*e.g.*, `fork`, `lock`, `wait`, `join` *etc.*), which we refer to as *events*. The schedule of *events* leads to an exponential growth of the state space of multithreaded programs. However, since multithreaded programs are commonly scheduled by the underlying operating system in an arbitrary manner, triggering a bug is non-deterministic in practice. Especially, for real-world applications with a large amount of possible schedules, chances of bug detection get further decreased. Therefore, we highlight the first challenge in finding concurrency bugs as below.

**Challenge 1: Expose bug-triggering schedules.** With the increase on the number of threads and *events*, exploring all the possible schedules is not only expensive, but sometimes infeasible. Consequently, triggering a bug via the "right" schedule is intractable in practice.

Moreover, thread schedules can greatly affect the path coverage in concurrency testing as well. To be specific, we consider a program with a set of control flow paths. The execution of a specific path is determined by whether the branch condition is met or not. In the concurrency setting, valuation of the branch condition may be sensitive to different thread schedules, which in turn steers the execution towards different paths. That said, without covering the branch-turning schedule, we may miss the execution of the uncovered code, thus miss bugs. To sum up, we highlight the second challenge of analyzing multithreaded programs.

**Challenge 2: Cover schedule-sensitive branches.** In order to cover as many paths as possible in testing, we have to identify schedule-sensitive branches and generate specific schedules to take all the branch paths. This requires reasoning branch constraints in a multithreaded setting, which is hard in practice.

**ConFu Solution.** To overcome the limitations, we have developed ConFu for analyzing concurrent software and instantiated it for multithreaded JVM applications. ConFu employs a potent combination of guided schedule fuzzing and symbolic trace analysis, which identifies bug-triggering interleaving in complex applications (via fuzzing) and achieves a good coverage on program paths through control flow information (via symbolic trace analysis). Our preliminary evaluation on widely-used benchmarks and industrial microservices showed that ConFu managed to find concurrency bugs efficiently and dig out deep ones. ConFu is available at https://github.com/njaliu/ConFu-release. A demo video is at https://youtu.be/R2YCyOrMPd4.

## 2 OVERVIEW 重点看overview这部分，2.1和2.2

### 2.1 ConFu Architecture

Generally, ConFu takes as input the software under test (SUT), including class files, jar and war packages. In terms of output, ConFu generates a bug report including specific types of bugs found in the analysis and runtime coverage information which helps users better understand the results and tune the analysis. Currently, ConFu is able to detect four common types of concurrency bugs, *i.e.*, assertion failures, data races, deadlocks and null pointer dereferences. Since the main strength of ConFu is to efficiently cover both thread interleaving and program paths rather than the detection itself, we implement well-designed detection algorithms in ConFu. Specifically, we check assert exceptions to capture assertion failures. Data races and deadlocks are detected using the FastTrack [5] and GoodLock [6] algorithms, respectively. In terms of null pointer dereferences, we extend FastTrack by dumping heap memory involved in a data race and flag a bug if the race incurs a read operation and the memory value is `null`.

The architecture of ConFu is shown in Figure 1. Specifically, ConFu works in three modes, *i.e.*, schedule fuzzing mode, symbolic trace analysis mode and bug detection mode. Generally, the schedule fuzzing mode aims at efficiently exploring the state space based

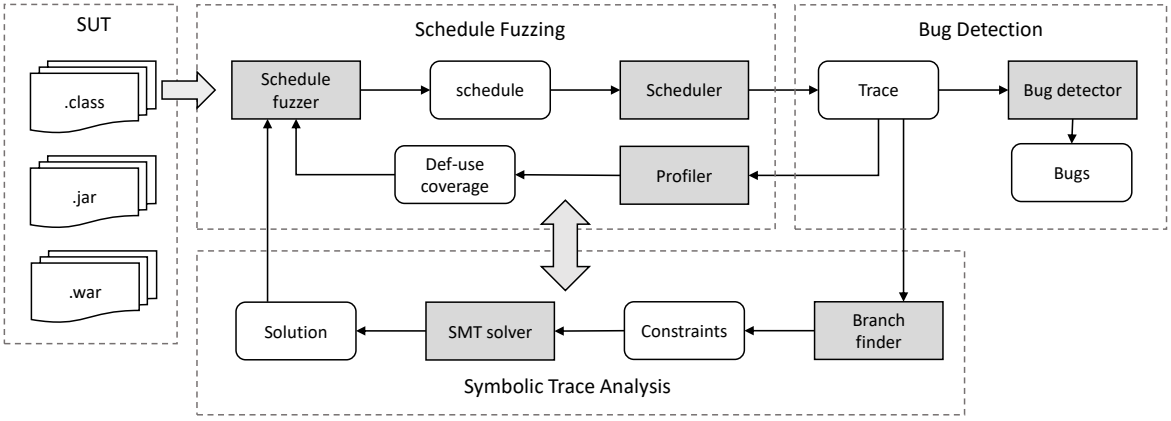


**Figure 1: The ConFu framework.**

on observed traces. Differently, the symbolic trace analysis mode tries to infer the existence of possible unobserved paths and drive the execution to cover them. Lastly, the bug detection mode targets at capturing various types of concurrency bugs. Specifically, ConFu consists of six modules, *i.e.*, *Schedule fuzzer*, *Scheduler*, *Profiler*, *Branch finder*, *SMT solver* and *Bug detector*. Next, we describe the general workflow of ConFu and interaction among modules.

First, the *Schedule fuzzer* aims at fuzzing thread schedules. To this end, it runs the program and randomly generates a set of priority change (PC) points in the program. At PC points, context switches are enforced. At non-PC points, the thread which owns the CPU will proceed. Then, *Scheduler* leverages the computed schedule to execute threads and produce runtime traces. Based on a trace, a *Bug detector* checks whether specific types of bugs occur or not. At the same time, *Profiler* computes the coverage information as a feedback to the *Schedule fuzzer*. In our setting, we mainly consider the define-use coverage metric [10]. This metric is suitable for detecting memory access related bugs, which is a majority in real-world applications. The coverage information serves as a guidance to help the *Schedule fuzzer* better generate new schedules in iterative runs. Moreover, ConFu switches to the symbolic trace analysis mode when fuzzing saturates. A *Branch finder* is notified to find schedule sensitive branches [7] based on previously collected traces. To do that, it builds a set of constraints of event ordering and memory dependency. The constraints are passed to a *SMT solver* to check satisfiability. For unsatisfiable results, we ignore the trace. In terms of satisfiable ones, which indicate that specific event ordering can pick an unexplored path of the branch, we generate a solution to the constraints and leverage the *Schedule fuzzer* to enforce the computed schedule of the solution.

## 2.2 Key Insight

To hunt deep concurrency bugs and make ConFu practically useful, our design adopts the following key optimizations and insights.

**Combined Analysis Engine.** ConFu employ a dynamic transition between schedule fuzzing and symbolic trace analysis modes, which enables bug detection in a relatively lightweight manner (schedule fuzzing) and at the same time stays sensitive to control flow structure as much as possible (symbolic trace analysis).

**Stochastic Schedule Fuzzing.** In the schedule fuzzing mode, ConFu enables stochastic optimizations on schedule fuzzing. Memory accesses from a thread are put into groups such that the number of scheduling points get reduced. A memory access group does not allow write-read or write-write to the same memory location. To schedule the program at a group granularity, ConFu uses a bool vector $\mathcal{X} = \langle x_1, x_2, \cdots, x_k \rangle$ to specify whether group $i$ generates a context switch. At each iteration, ConFu leverages Markov Chain Monte Carlo (MCMC) to propose and accept a new vector based on the current $\mathcal{X}$. The proposal randomly flips a small number of values in $\mathcal{X}$. Using the coverage to encode the objective function, ConFu accepts the proposal with a specific probability.

**Symbolic Trace Analysis.** In the symbolic trace analysis mode, the goal of ConFu is to identify schedule sensitive branches [7] and steer the execution to find unexplored paths. To this end, ConFu constructs a set of constraints based on runtime traces, *i.e.*, program order (PO), synchronization order (SO), memory dependency (MD) and branch condition (BC). Specifically, PO requires operations from the same thread are ordered as in traces. SO defines partial orders on synchronizations, *e.g.*, `fork`, `lock`, `unlock`, `wait`, `notify` *etc.*. MD says that a read memory access gets the value from the most recent write. BC is the set of predicates of branches. For explicit branches as `if-else`, ConFu uses the branch condition as BC. For implicit branches as exception handling, ConFu predefined patterns to indicate the condition of exception, *e.g.*, divide by zero, out-of-bound access *etc.*. Based on the collected constraints, ConFu leverages an SMT solver to check whether they are satisfiable. If so, ConFu generates a feasible solution as the new schedule.

# 3 USING CONFU

## 3.1 Key Features

Currently, ConFu is a command-line based tool. we introduce the key features of ConFu with the example shown in Figure 2,

**Detector Configuration.** Three types of bug detectors in ConFu can be manually enabled, including `FT2` (data races), `DL` (deadlock), `NPD` (null pointer dereference). Detector of assertion failures is activated by default. The following command detects data races.

```java
public class Account{
  public static int Bank_Total = 0;
  public static BankAccount[] accounts;
  ...
  public void go(String[] args) {
    accounts = new BankAccount[NUM_ACCOUNTS];
        for(int i = 0; i< NUM_ACCOUNTS; i++) {
          accounts[i] = new BankAccount(i);
          accounts[i].start();
        }
        ...
        if(Bank_Total == Total_Balance)
          throw new RuntimeException("ERROR: don't match!");
        ...
```

**(a) `Account.java`**

```java
class BankAccount extends Thread {
  public int Balance = 0;
  ...
  public void run() {
    for(int i = 0; i< loop; i++) {
          int sum = random.nextInt()%MAX_SUM;
          Balance += sum;
          Account.Bank_Total += sum;
          ...
```

**(b) `BankAccount.java`**

**Figure 2: A simple program**

```
./confu -classpath=tests/ -tool=FT2 account.Account
```

```
======================================================================
FastTrack Error

        Thread: 2
         Blame: account/Account.Bank_Total_I
         Count: 1    (max: 100)
  Shadow State: [W=(1:3) R=(1:3) V=[]]
 Current Thread: [tid=2    C=[(0:5) (1:0) (2:3) (3:0)]   E=(2:3)]
         Class: account/Account
         Field: null.account/Account.Bank_Total_I
       Message: Write-Read Race
  Previous Op: Write by  Thread-0[tid = 1]
  Currrent Op: Read by  Thread-1[tid = 2]
        Stack: Use -stacks to show stacks...
    Race Pair: account/Account.java:98 --- account/Account.java:98
======================================================================
```

**Figure 3: Output of race detection**

Figure 3 shows a data race bug found. In the `Blame` row, the report shows the shared memory involved in the data race, *i.e.*, `Balance_Total` in this case. The `Message` indicates the type of the current race. `Write-Read Race` means the race includes a pair of concurrent operations, *i.e.*, a read follows a write to the same memory in the observed runtime trace. `Race Pair` shows the source locations involved in the race.

**White List.** This feature allows ConFu to ignore specific classes in the analysis. In practice, users may need such a feature when they are analyzing a complex application where considering all classes is time-consuming and resource-expensive, or when they do not want results from specific classes get mixed with others. The command below is used to enable the white-list feature.

```
./confu -classpath=tests/ -tool=FT2
        -classes="-.*Account.*" account.Account
```

In the `classes` option, the white-list specifies classes In this case, any classes containing "Account" will be ignore. As a result, both `Account.java` and `BankAccount.java` escape.

**Go Fuzzing.** The schedule fuzzing and symbolic trace analysis feature as in §2 can be enabled by the following command.

```
./confu -fuzz 20 -classpath=tests/
        -tool=PCT:FT2 account.Account
```

```
Bank records = 49, accounts balance = 125.
java.lang.RuntimeException: ERROR: records don't match !!!
        at account.Account.__$rr_checkResult__$rr__Original_(Account.java:81)
        at account.Account.checkResult(Account.java:74)
        at account.Account.__$rr_go__$rr__Original_(Account.java:70)
        at account.Account.go(Account.java:34)
        at account.Account.__$rr_main__$rr__Original_(Account.java:30)
        at account.Account.main(Account.java:29)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:497)
        at rr.RRMain$3.run(RRMain.java:179)
[main: PCT tool finished!]
[RR Shutdown: Total Time: 1111]

[RR Shutdown: Time = 1111]
++++ date-confu fuzzing  19 ends
```

**Figure 4: Output of fuzzing**

The `fuzz` option enables the backend engine which combines stochastic schedule fuzzing and symbolic trace analysis of ConFu. This options needs a parameter to specify the number of iterations of the fuzzing. Furthermore, we add the `PCT` tool in the tool chain to generate and enforce thread schedules in a guided manner. Figure 4 shows the output by enabling the fuzzing feature. Specifically, a runtime exception was triggered in the detection, which did not occur without fuzzing as in Figure 3. Compared to the single-run bug detection, the fuzzing feature aims at uncovering as many bugs as possible (we can add more tools in the tool chain to capture other types of bugs), especially deeply rooted ones.

**Coverage Collection.** To collect runtime coverage, ConFu uses the command below.

```
./confu -fuzz 20 -classpath=tests/
        -tool=COV:PCT:FT2 account.Account
```

The `COV` tool helps collect the coverage information[1] during the fuzzing period and generate a coverage report when finished. We can use `java -jar coverage/draw.jar coverage/` command to generate a graph of the accumulative coverage. The coverage information gives users a quantitive estimation on how much state space is covered.

[1] *Def-use* coverage [10] is collected since it is suitable for finding concurrency bugs.

**Table 1: Preliminary results**

| Benchmark | PCT | | CalFuzzer | | ConFu | |
|---|---|---|---|---|---|---|
| | #bug | prob | #bug | prob | #bug | prob |
| montecarlo | 5 | 1.00 | 5 | 1.00 | 5 | 1.00 |
| moldyn | 2 | 1.00 | 2 | 1.00 | 2 | 1.00 |
| Tomcat | 21 | 0.72 | 15 | 0.51 | 28 | 0.81 |
| Log4j | 33 | 0.83 | 17 | 0.69 | 41 | 0.92 |
| DBCP | 17 | 0.81 | 11 | 0.42 | 26 | 0.92 |
| DeltaDB | 4 | 0.27 | 0 | 0.00 | 8 | 0.38 |

## 3.2 Typical Use Cases

Typical use cases of ConFu include analyzing stand-alone programs and server-side applications. We briefly describe the two cases.

**Analyze Stand-Alone Programs.** To analyze stand-alone programs as in Figure 2 is quite straightforward. Users can directly pass the main class as an argument and configure ConFu. We do not need manually modify or recompile the programs. This use case commonly happens in the early stage of development when independent small modules have been developed.

**Analyze RESTful Microservices.** Another use case of ConFu is analyzing RESTful microservices. To do that, we need to start the microservice through ConFu. For example, given a spring-boot based microservice packaged in a jar file, we use the `JarLauncher` class as the main entry of execution. Then, we need to run the microservice by sending requests to its REST APIs, *e.g.*, `GET`, `POST` *etc.*. ConFu monitors the execution of the microservice and reports bugs at runtime.

```
1  @RequestMapping(method = RequestMethod.POST)
2    public List<ToDo> post(@RequestBody ToDo todo) {
3      this.todos.add(todo.withId(this.id++));
4      return this.todos;
5  }
```

**Figure 5: A buggy spring-boot microservice**

Figure 5 shows a spring-boot microservice bug found by ConFu. Specifically, if two `POST` requests arrive at a specific timing, the REST API may create two `ToDo` objects with the same id (line 3), thus leads to an error.

# 4 PRELIMINARY EVALUATION

We have been using ConFu to analyze open-source applications and real-world software. Preliminary results are shown in Table 1 (compared to PCT [3] and CalFuzzer [9] in terms of bug detection capability and bug hitting probability). On average, ConFu found 40% more bugs and hit bugs with higher probability. In analyzing an industrial time-series database called DeltaDB [1], ConFu found a serious exploitable bug which can fail the database server connection and in turn lead to a Deny-of-Service vulnerability.

# 5 RELATED WORK

Finding concurrency bugs has been a long-standing research problem. Targeting at deadlocks, a class of techniques have been proposed [4, 13]. Non-deadlock bugs include data races, ordering and atomicity violations. To detect data race bugs, both static techniques [12] and dynamic techniques [5] were proposed based on algorithms such as happen-before [5] and lockset [15]. Order and atomicity violations are also discussed in [14]. The ConFu is not limited to any specific type of concurrency bugs and provides a state space exploration framework for efficient bug detection.

# 6 CONCLUSION

In this demo paper, we present ConFu, a dynamic analyzer for concurrent applications. The key insight is a potent combination of guided schedule fuzzing and symbolic trace analysis techniques. We have instantiated ConFu for JVM applications. In the preliminary evaluation, ConFu showed its potential in finding bugs more efficiently and digging out deeply rooted ones. ConFu is available at https://github.com/njaliu/ConFu-release. A demo video is available at https://youtu.be/R2YCyOrMPd4.

# REFERENCES


[1] 2017. DeltaDB. https://github.com/thulab/tsfile. (2017).

[2] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 81–96.

[3] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 167–178.

[4] Yan Cai and WK Chan. 2014. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering* 40, 3 (2014), 266–281.

[5] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133.

[6] Klaus Havelund. 2000. Using runtime analysis to guide model checking of Java programs. In *International SPIN Workshop on Model Checking of Software*. Springer, 245–264.

[7] Jeff Huang and Lawrence Rauchwerger. 2015. Finding schedule-sensitive branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 439–449.

[8] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 389–400.

[9] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An extensible active testing framework for concurrent programs. In *Computer Aided Verification*. Springer, 675–681.

[10] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A study of interleaving coverage criteria. In *The 6th joint meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*. ACM, 533–536.

[11] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 329–339.

[12] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 308–319.

[13] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *ICSE 2009. IEEE 31st International Conference on Software Engineering, 2009*. IEEE, 386–396.

[14] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 25–36.

[15] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.