Fuzzing for concurrent programs under C/C++ weak memory model

Master's Thesis

Luan Li

Fuzzing for concurrent programs under C/C++ weak memory model

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Luan Li
born in Liaoning, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Fuzzing for concurrent programs under C/C++ weak memory model

Author: Luan Li

Abstract

Fuzzing has been a popular approach in the domain of software testing due to its efficiency and capability to uncover unexpected bugs. Fuzz testing was originally developed in the days of sequential programs. With the rise of multi-core devices and increasing demand for computational efficiency, the prevalence of concurrent programming has led to a new wave of research applying fuzz testing techniques. In recent years, several fuzzers have been proposed for sequentially consistent multi-threading programs, a subset of concurrent programs, using thread interleaving semantics. However, exploration of fuzzing techniques for weak memory concurrency remains limited.

This thesis presents a novel fuzzing approach for programs under weak memory models. It generates test cases as execution graphs instead of thread schedules, and performs mutations on the execution graphs to generate new test cases.

We implement the fuzzer based on two state-of-the-art testing tools: C11Tester and GenMC. Different mutation strategies are explored for comparison. Benchmark results demonstrate that our fuzzer explores a broader range of execution graphs compared to naive random testing, resulting in improved bug detection.

Key Words: fuzz testing, weak memory model, execution graph, concurrency bug

Thesis Committee:

Chair:	Dr. Prof. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Burcu Kulahcioglu Ozkan, Faculty EEMCS, TU Delft
External supervisor:	Dr. Ori Lahav, School of Computer Science, Tel Aviv University
Committee Member:	Dr. Jeremie Decouchant, Faculty EEMCS, TU Delft
External Committee Member:	Dr. Michalis Kokologiannakis, ETH Zurich

Preface

First of all, I would like to thank Dr. Burcu Ozkan, Dr. Ori Lahav, and Dr. Michalis Kokologiannakis, who have provided me with extensive guidance. Without their supervision and support, I would not have been able to conduct this research project. It has been a great honor to learn from and work with such outstanding individuals. I would also like to express my sincere thanks to Dr. Prof. Arie van Deursen and Dr. Jeremie Decouchant for their invaluable support to this thesis project.

I am grateful for my two years as an MSc student at TU Delft. I would like to thank the professors and teaching assistants for providing incredible lectures and learning materials. I understand that "it takes ten minutes of class time for every ten hours of preparation." I would also like to thank Dr. Jonas Thies for supervising my research internship, which was a truly memorable experience. I am thankful to the EEMCS faculty for offering such a supportive platform with so many kind people and fascinating research opportunities.

I would like to thank my family for their constant support, both mentally and financially. I wish them a joyful and fulfilling life. I also want to thank my friends; it has been a great fortune to have them with me throughout this journey.

Last but not least, knowing oneself is as difficult as knowing others. Learning to get along with and befriend myself has always been an important lesson. "Be the water clear, it can wash my tassels; be the water muddy, it can wash my feet." I hope I never stop exploring this wonderful world.

Luan Li
Delft, the Netherlands
September 11, 2024

Contents

List of Figures

Chapter 1

Introduction

As software systems continue to grow in size and complexity, the occurrence of software bugs becomes an inevitable challenge. Industry experience shows that software often contains 1-25 bugs per thousand lines of code[51] and the number of bugs increases quadratically with the size of the codebase[11]. Software bugs can lead to critical program errors or even crashes, which, in turn, may result in significant financial losses[50, 5] or pose serious risks to safety and life[21, 30, 45]. To address the threat posed by bugs or vulnerability of programs, researchers have investigated a variety of bug detection techniques.

There are two major ways to detect bugs: formal verification methods and testing methods. Formal verification techniques, such as axiomatic approaches, use mathematical deduction to prove the absence of bugs. These approaches heavily rely on the expertise of developers and normally require a significant amount of time and effort[33]. Formal methods can sometimes produce false positives, which further increases the complexity and time cost of verification. Automated testing, on the other hand, has for long been of great importance for its scalability and efficiency. Although testing cannot prove full correctness of a program, it is very effective in finding real bugs. Various testing methods have been developed over the years, including static analysis[12, 9] and dynamic testing tools[69, 68]. However, due to the complexity of the programs being tested, static analysis tools do not always report bugs comprehensively or correctly. Dynamic testing, on the other hand, often requires high-quality test cases to cover a large portion of program behaviors, which may demand a deep understanding of the program under test, can be time-consuming, and requires significant effort to complete.

Fuzz testing has become increasingly popular in recent years. It repeatedly executes the tested program by generating random inputs and monitors for any observed buggy behaviors. These approaches are usually easy to apply and scale well. Fuzzers can be classified into three categories based on the level of knowledge they use from the tested programs: black-box, grey-box, and white-box fuzzers. A fuzzer typically has a feedback loop. It maintains a set of seeds as program inputs to execute the program. The information about the execution is collected to determine whether a seed is interesting, which means the seed has triggered new interested behaviors, such as covering new code branches. The interesting seeds will be used for generating new seeds during repeated executions. One of the most popular fuzzing tool is AFL[78], a coverage-guided mutation-based grey-box fuzzer

which achieves superior efficiency than previous black-box fuzzers. Since then, researchers have developed various techniques to improve the code coverage and accelerate the bug detection.

Entering the multi-core era, concurrent programming has gained increasing significance. The need for testing concurrent programs has also grown considerably. Researchers have developed various testing techniques, and fuzzing for concurrent programs has gained increasing attention. Traditional coverage-guided approaches face challenges in detecting concurrency bugs, as code coverage information does not reflect thread interleavings, which can lead to such bugs. Therefore, thread-relevant instrumentation is needed to provide concurrency feedback information in the fuzzing loop. Another problem is that, assuming sequential consistency, both the program input and the thread interleavings (or schedules) determine the program's behavior. Hence existing concurrency fuzzers can be classified into two types: fuzzers aiming for generating seeds[16] and thread interleavings[54, 29].

However, current concurrency fuzzers mainly focus on testing for programs under sequential consistency memory model. Modern computer architectures, such as Power[67] and ARM[22, 64], often allow speculative and out-of-order executions and introduce cache hierarchies to reduce memory access latency. On the one hand, although sequential consistency is easy to understand by programmers, achieving it is very expensive. On the other hand, by relaxing the memory order and allowing for weak memory behaviors, the efficiency of execution can be significantly improved. However, both developing and testing programs under weak memory have been notoriously hard. Given the success that fuzzing has achieved on sequential programs and multi-threading programs under SC memory, it is reasonable to believe fuzzing can also be helpful for weak memory testing.

In this thesis, we propose a novel fuzzing approach designed to support weak memory models. Unlike existing fuzzers that rely on random seeds or thread schedules, our approach mutates execution graphs to generate test cases. Due to the generality of execution graphs, our fuzzer also supports programs under the sequential consistency memory model. We then present two implementations in both C11Tester and GenMC, which are state-of-the-art platforms for testing weak memory programs.

The rest of this thesis is structured as follows: Chapter 2 provides the background information on fuzzers, weak memory models, execution graphs, and the C/C++11 memory model. Chapter 3 presents the intuition and a high level overview of the fuzzing algorithm. Chapter 4 describes the implementation on C11Tester, with the evaluation results and discussion. Chapter 5 describes the implementation on GenMC and the evaluation of three mutation strategies. Chapter 6 briefly summarizes some other related work on fuzzing, memory models and model checking, etc. Chapter 7 concludes this thesis and discusses possible future work.

Chapter 2

Background

2.1 Fuzzers

A fuzzer is a program that performs fuzz testing, or fuzzing. The idea of fuzzing was proposed by Miller et al. [55] in the late 1980s. They developed a program called "Fuzz" that generates random input strings for testing programs that have special requirements on inputs. If the generated input string can pass the program's input check but results in unexpected errors, a bug is detected. The fuzzing technique can automate software testing procedures and has been advanced significantly over the past years.

Fuzzers can be classified into three categories: black box, gray box, and white box fuzzers. Early fuzzers that came after Fuzz were primarily black box fuzzers. Black box fuzzing is I/O-driven fuzzing, which only tracks the input and the corresponding output data from a program, without knowledge of its internal states and their relationships with the input. Therefore, it is relatively simple to design and deploy black box fuzzers for a wide range of programs, especially for those that are not open-sourced because black box fuzzing can be performed non-intrusively. The drawback of black box fuzzing is that, because it does not have internal information about the program, it may expend significant effort generating irrelevant inputs and achieve low testing coverage. On the other hand, white box fuzzing usually has sufficient knowledge of the program's internal information. The first white box fuzzer was SAGE[25], which starts from a well-formed input and executes the program while collecting alternative branches along the path. These branches can be used to constrain the input generation and guide to cover new execution paths. Since white box fuzzing has full knowledge of the tested program, it can generate high-quality inputs that cover a large fraction of the execution paths. However, such approaches typically use symbolic execution techniques and constraint solvers, which usually consume large computation resources and face the challenge of state explosion. To balance the benefits of these two approaches, gray box fuzzing utilizes a small amount of the internal information and has become popular since the success of AFL[78]. Instead of using symbolic execution, it performs program instrumentation to collect the coverage information. It uses a genetic search algorithm to pick seeds and mutations that yield positive edge coverage. Since then, a large number of fuzzers based on AFL have been proposed[57, 39, 71, 23].

From another perspective, we could also categorize fuzzers with respect to their target programs. Consider the following three categories: sequential programs, sequentially consistent (SC) concurrent programs and concurrent programs under weak memory models. Traditional fuzzers are usually designed for sequential programs. For a single-threaded, deterministic program, a fixed input will produce a fixed output. Therefore fuzzers only generate and mutate program inputs. Take AFL for example, it generates program inputs, or seeds, that trigger interesting execution paths. For SC concurrent programs, program behaviors are determined by both program input and thread interleavings, hence fuzzers for such programs can have two respective targets. For example, MUZZ[16] targets program inputs, especially those that can cover thread-relevant execution paths of the program. It conducts static analysis on the program and instruments it in a biased manner on the concurrent parts of the code, such as the code between the thread creation and joining and outside critical sections, instead of uniformly instrumenting the code like AFL does. The biased instrumentation can guide the fuzzer to generate more thread-relevant inputs that can be used to detect concurrency bugs, such as data races. Conzzer[29], on the other hand, searches for thread schedules that cause bugs. It collects pairs of function call stacks as seeds and picks adjacent functions to generate new function call pairs. It proactively controls the scheduling and forces the selected call pair to be executed concurrently. RFF[54] uses reads-from pairs, consisting of a read instruction and its corresponding write, as seeds and enforces selected read-from pairs by prioritizing the read thread. After the read is executed, it then prioritizes the write thread. Both Conzzer and RFF are targeted at thread interleavings. Table 2.1 summarizes the aforementioned fuzzers and their definitions of fuzzing concepts, such as seeds and mutations.

Fuzzers	target (seed)	mutation
AFL	program input	xor, bit shift, hashing etc
MUZZ	thread-relevant program input	xor, bit shift, hashing etc
Conzzer	function call pairs	pick adjacent functions
RFF	reads-from pairs	changing rf pairs

Table 2.1: Fuzzers with their seeds and mutations

As illustrated in Table 2.2, to the best of our knowledge, while many fuzzers exist for sequential programs, there are only three fuzzers specifically designed for concurrent programs. However, none of these have been developed to address concurrent programs under weak memory models. In the field of weak memory concurrency research, program behavior is usually modeled by execution graphs. In this project, we develop a fuzzing approach based on execution graph semantics, using the graph prefix as seeds and changing reads-from relations as mutations. The details of the fuzzing algorithm and its implementation are provided in later chapters.

program	fuzzers
single-threaded	SAGE, AFL, TriforceAFL, kAFL, Driller, CollAFL, etc
SC multi-threaded	MUZZ, Conzzer, RFF
weak multi-threaded	??

Table 2.2: Fuzzers with their application domains

2.2 Weak Memory Models

In concurrent programming, shared memory is used to share data and pass messages among threads. Memory models are essential for programmers to reason about their code and for compilers and hardware manufacturers to implement low-level support infrastructure. The earliest memory model, proposed by Lamport[41] in 1979, is the Sequential Consistency Model (SC) . Under the SC model, intra-thread instructions are executed following their program order and threads can interleave in any order. A read operation can only read the most recent value written to the same memory location. The SC model is also known as the strong memory model, while other memory models are referred to as weak memory models.

Consider the store buffer (SB) 2.1 example, where `x`, `y` are shared variables, and `r1`, `r2` are local variables, all initialized with 0. Under SC, none of the possible thread interleavings (i.e. abcd, acbd, acdb, cadb, cdab, cabd) result in both `r1` and `r2` reading the value 0.

```
x = 0;
y = 0;
void thread1() {
    x  = 1;  // (a)
    r1 = y;  // (b)
}
void thread2() {
    y  = 1;  // (c)
    r2 = x;  // (d)
}
```

Listing 2.1: Store Buffer (SB) program

However, this behavior may be allowed by some weak memory models provided by hardware architectures and programming languages. For example, consider TSO (Total Store Order) [70], which is supported by x86 architectures. In the TSO model, each thread has a local store buffer. Values written to shared memory are first stored in the buffer and at some time in the future, will be flushed to the shared memory. The store buffer has the FIFO property, hence the ordering of all writes in the same thread will not be broken.

In the SB example, if the memory model is TSO, it is possible that after executing assignments `a` and `b`, the values are buffered, followed by `r1` and `r2` reading 0, and finally the buffered values flushed to the shared memory.

Some weak memory behaviors can be forbidden by one weak memory model but allowed by another. In the following message passing (MP) 2.2 example, after `data` is set to 1, the sender thread initializes the pointer, `p`, with the address of `data`, hoping that the

receiver thread uses the data only after the pointer is initialized (indicating that `data` is set). Under TSO, due to the FIFO property of store buffers, the shared variable `p` is initialized only after the data update is complete. However, this is not guaranteed under the PSO (Partial Store Order) model [72]. In PSO, each memory location has a separate FIFO store buffer in a thread. In this case, the ordering of moving the values of `data` and `p` from their buffers to the shared memory is not restricted. The receiver thread may read `y=1` when `data` has not been updated yet.

```
p = nullptr;
data = 1;
// sender thread
void sender() {
    data = 1;
    p = &data;
}
// receiver thread
void receiver() {
    while(p == nullptr) {;}
    use(*p);
}
```

Listing 2.2: Message Passing (MP) program

There are a variety of other weak memory models, such as the ARMv8 [64] memory model, supporting out-of-order executions and speculative executions, and language-level memory models, including the Java memory model[48] and C++ memory model. The rest of this paper primarily discusses the C/C++11 memory model[8], which provides weakly-ordered atomic operations to support weak memory behaviors.

2.3 Execution graphs

Execution graphs[7, 56, 6] are directed graph representations of program executions. Under the SC model, program executions can always be represented by a single sequence of events, as SC enforces that operations execute as if they occur sequentially by definition. While under weak memory models, programs can exhibit more complicated behaviors, where execution graphs are helpful in capturing a wider range of possible executions.

The nodes in an execution graph represent events or actions within a program, while the edges connecting them depict relationships as defined by the memory model. In this thesis, the terms ”event” and ”action” are used interchangeably. An event refers to a basic operation related to memory. For example, the line of code `y = x + 1` involves two events: a read event for the variable `x` and a write event for the variable `y`. Typically, not all variables are included in the graph—only those related to shared memory accesses, such as reads and writes to shared variables, since others may be irrelevant to understanding the program’s behavior. Reads and writes to local variables that do not affect other threads are usually omitted from the graph. For instance, if `a = x` reads the value of a shared variable `x` and assigns it to a local variable `a`, this is often represented in the graph as a single event: the read of `x`.

The edges in the execution graph encode the event relations specified by the memory model. For example, if two events, e_1 and e_2, have a *sequenced-before* (sb) relation, i.e. e_1 is sequenced before e_2, an arrow will be drawn in the graph from e_1 to e_2. As with events, not all relations are always depicted in the graph. For instance, if two events have multiple relations in the same direction, some less important ones may be omitted for simplicity or clarity. In the execution graphs presented in this thesis, edges of the same color represent the same type of relation when their names are omitted. Further details about these relations are discussed in the next section.

We use an example to introduce the notations used in execution graphs in this thesis. Figure 2.1 shows a possible execution graph of the SB program2.1. The five nodes represent five different events in the program. The directed edges are annotated with the names of the relations between these events. For instance, consider the node `W(x, 1)`: `W` stands for a write operation, and `W(x, 1)` represents writing the value 1 to the variable x (a). In the case of `R(x)`, `R` stands for a read operation. This node represents a read event for x (d), with the value being omitted because it can be obtained from the node from which the *read-from* (rf) edge originates.

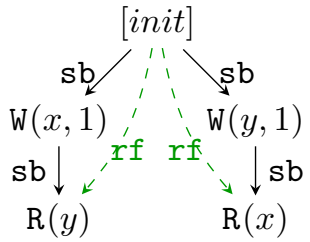

Figure 2.1: Execution graph example

The semantics of a program is a set of consistent graphs. Events within a graph cannot be connected arbitrarily by relations. Instead, the graphs should be consistent to the memory model. Consistency is specific to each memory model. An execution graph that is consistent to one memory model may not be consistent to another. In the SB example, the execution graph shown in Figure 2.1 is consistent under the SC model, but not under TSO. The next section will introduce the consistency specifications under the C/C++11 memory model.

2.4 C/C++11 Memory Model

C/C++11 provides additional concurrency primitives, including atomics, mutex, threads and fences, along with an extensive specification of its memory model. The first C/C++11 memory model was described in a proposal[10] in 2008, which was refined and formalized by [8]. The following contents use the notations and definitions in [8], unless otherwise specified.

The memory model can be defined as a function, taking a set of candidate executions X as input. These executions must be allowed by the operational semantics and are consistent,

denoted as pre-executions. The function returns ”NONE” if any executions have undefined behaviors; otherwise, it returns ”SOME” pre-executions.

A candidate execution X contains two components, $X = (X_{opsem}, X_{witness})$, where X_{opsem} is determined by the operational semantics and $X_{witness}$ is an existential witness of some further data. Both components are composed of memory actions (or actions for short) and relations. An action can be a non-atomic read or write, atomic operations, mutex operations and fences, represented by ¡aid, tid, type, location, value¿. The X_{opsem} contains three types of relations:

- *sequenced-before* (sb): A relation between intra-thread actions given by C/C++ language specifications, also referred to as program order. When two separate actions are written in two separate statements, the former is sequenced before the latter.
- *data-dependency* (dd): The dd is provided by the operational semantics, primarily used for release/consume atomics. For example, a store to a pointer and the use of the pointed data have a dd relation.

In the SB example, assuming that x and y are atomic variables, the X_{opsem} of a candidate execution can be drawn as in Figure 2.2.

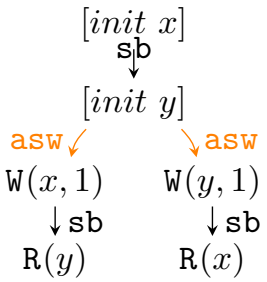

Figure 2.2: X_{opsem} of SB

The $X_{witness}$ part contains three additional relations. These relations are not uniquely determined by the operational semantics. Therefore, given a program p, the candidate execution X can have only one X_{opsem}, but multiple possible $X_{witness}$ configurations.

- *read-from* (rf): An rf edge is established from a write action (non-atomic write, atomic write, or read-modify-write) to a read action (non-atomic read, atomic read, or read-modify-write) if the read action retrieves a value from the write action. Additionally, an rf edge is established between a lock action and its immediately preceding unlock action for the same mutex. The rf reads-from map is a function that includes all these rf relations in the execution.
- *modification-order* (mo): This represents a total order of all writes to the same atomic location. Each location can have its own independent mo ”chain,” which is unrelated to chains for other locations.

- *sequentially-consistent* (sc): This totally orders all mutex actions and actions with `mo_seq_cst` memory order.

In the SB example, assuming the initializations are non-atomic and other writes and reads are `mo_seq_cst`, a possible $X_{witness}$ for the SB example can be shown in Figure 2.3

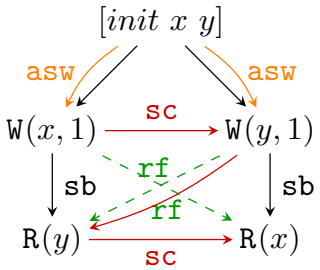

Figure 2.3: $X_{witness}$ of SB

There are some derived relations defined based on the above six relations. These derived relations will help to define the memory model and rule out illegal executions.

- *synchronizes-with* (sw): Every unlock action of a mutex has an sw edge pointing to the lock ordered after it in the sc order mentioned above. A read-acquire (read with `memory_order_acquire`) reading from a write-release gives rise to a sw relation. More generally, when the read-acquire R reads from a write W, it also synchronizes-with other write-release that is ordered before W in the modification order. However, not all write-releases preceding W can have sw relations with W, only those contained by the *release sequence* of W. The definition of *release sequence* is omitted here.

- *happens-before* (hb): If the execution has no consume operations, the hb relation is a transitive closure of $sb \cup sw$. More generally, hb is defined as the union of sb and *inter-thread-happens-before*, which includes the sw relation.

The three relations in $X_{witness}$ (rf, mo and sc) cannot be arbitrarily composed to make an execution. Instead, they have to satisfy some constraints, called *coherence*. The coherence constraints have the form "A-B Coherence", or CoAB, where both A and B are either reads or writes, and $A \xrightarrow{\text{hb}} B$. As illustrated previously, the hb is derived from sb and sw, where sw is derived from sc and rf. The constraints on hb, mo and rf will ultimately constrain the combinations of rf, mo and sc. The coherence constraints are listed below:

- *Read-Read Coherence* (CoRR): Two reads ordered by hb cannot read from two writes ordered by mo in the other direction.

- *Write-Read Coherence* (CoWR): When a write, w, happens before a read r, r cannot read from a write that precedes w in mo.

- *Write-Write Coherence* (CoWW): The mo and hb relations of two writes, w_1 and w_2, should have same directions. For instance, $w_1 \xrightarrow{\text{mo}} w_2 \wedge w_2 \xrightarrow{\text{hb}} w_1$ is not allowed.
- *Read-Write Coherence* (CoRW): When a read happens before a write w, it cannot read from a write that is ordered after w in mo. This forbids the $rf \cup hb \cup mo$ to be cyclic.

Consider the SB example. If the reads and writes are SC atomic operations, the execution shown in Figure 2.4, where both reads read the value 0, is not allowed because this execution violates the CoWR constraint. However, if the reads and writes are relaxed, then both reads reading the value 0 from the initializations is permitted.

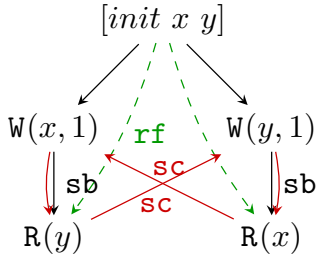

Figure 2.4: Not-allowed of SB

The C/C++11 memory model described above is an axiomatic model that specifies which executions are allowed and which are not. However, this model has some flaws. Firstly, the data-dependency relation and those associated with consume-release atomics are specified but are not implemented on most platforms, where they are typically treated as acquire-release atomic operations. Furthermore, there are some executions that are allowed by the model but should be ruled out in principle. For instance, consider a load buffer (LB) program where two threads load from different locations and store the loaded values to the other location. If all the atomic operations have a relaxed memory order, any loaded value is allowed by the model. This issue is known as the *out-of-thin-air* (OOTA) problem. Since the model was proposed, numerous efforts have been made to revise it, which will be summarized in Section 6.

Chapter 3

Fuzzing

In this chapter, we describe our high-level algorithm for fuzzing weak memory programs. The next two sections will implement mutations for test generation differently.

3.1 Motivation

As discussed before, weak memory programs can produce nontrivial set of possible executions and testing the program behavior under weak memory models is challenging. Here is a summary of several characteristics associated with random-based testing for weak memory programs, which we will illustrate with examples:

- The program behavior depends not only on scheduling decisions but also weak memory behaviors.
- Different scheduling decisions made by the test generator can lead to the same executions.
- The search space of execution graphs is typically large, often larger than in strong memory models.
- The probability of finding specific executions is not uniform, with some execution graphs being infrequent.

Firstly, weak memory models usually allow more behaviors which are not permitted in the SC model. For example, under a certain schedule, a read event may have more than one write events to read from. This presents challenges for testing weak memory programs, as their behaviors are not uniquely determined by scheduling decisions. Take SB for example, assuming all operations are relaxed, suppose a tester's scheduler adds events to an execution graph in the following order: `[`*init*`]` $\rightarrow$ `W(`x`, 1)` $\rightarrow$ `W(`y`, 1)` $\rightarrow$ `R(`y`)`. When `R(`y`)` is added, it has two reads-from options: `[`*init*`]` and `W(`y`, 1)`. This is not possible in the SC model, because under this schedule, `[`*init*`]` happens before `W(`y`, 1)`, which in turn happens before `R(`y`)`. Hence, `R(`y`)` can only read from the most recent write in this happen-before order. In

the weak memory model, however, since both `W(y, 1)` and `[init]` are relaxed, there is no happen-before relationship between them to constrain the read options for `R(y)`.

Secondly, different orders of adding events and forming relations can result in the same executions. For example, consider Listing 3.1, where two groups of threads are updating their flags. The test generator might first add events from threads in `group 1` and then from those in `group 2`, or it might choose to add events from `group 2` first. Since the threads in the two groups do not interact with each other, both cases can result in same execution graphs.

```
atomic<int> x = {};
atomic<int> y = {};

// group 1
void thread1() { x = 1; }
void thread2() { if(x == 1) {x = 0; /*...*/} }

// group 2
void thread3() { y = 1; }
void thread4() { if(y == 1) {y = 0; /*...*/} }
```

Listing 3.1: P6

Thirdly, the search space, which is the set of all possible execution graphs, can become very large or even infinite due to various combinations of relations, control flow branches, and loops. For simplicity, assume a single-threaded random test generator is used, which takes one step at a time—either adding a node to the graph or forming a relation between two nodes. Additionally, the test generator has a mechanism to ensure that the steps taken are valid according to the memory model. For example, consider the SB example: if all atomic operations are relaxed, the four execution graphs shown in Figure 3.1 are all allowed. In contrast, under the SC model, only the first three executions are permitted.

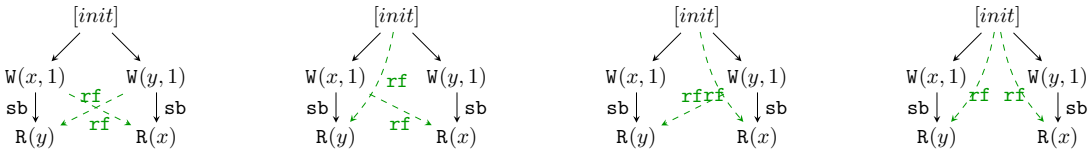

Figure 3.1: Valid execution graphs of SB

Lastly, the probability of encountering different executions is not uniformly distributed. Some execution graphs are more likely to be found, while others may have a lower probability of being discovered. Consider the example shown in Listing 3.2, where two threads update their corresponding flags `x1` and `x2`, and a third thread reads them. Suppose the test generator makes random decisions uniformly when adding events. The program printing 'A' requires one of the following orders shown in Table 3.1, with a total probability of $\frac{7}{18}$, while printing 'B' has a probability of $\frac{11}{18}$.

```
atomic<int> x1 = {};
```

order	probability
w1 → w2 → r1 → r2	$\frac{1}{3} \times \frac{1}{2} = \frac{1}{6}$
w2 → w1 → r1 → r2	$\frac{1}{3} \times \frac{1}{2} = \frac{1}{6}$
w1 → r1 → w2 → r2	$\frac{1}{3} \times \frac{1}{3} \times \frac{1}{2} = \frac{1}{18}$

Table 3.1: Probabilities of each order

```
atomic<int> x2 = {};

void thread1() {
    x1 = 1;      // w1
}
void thread2() {
    x2 = 1;      // w2
}

void thread3() {
    if(x1 == 1 /* r1 */ && x2 == 1 /* r2 */) {
        print("A");
        assert(false);
    }
    else {
        print("B");
    }
}
```

Listing 3.2: P2

The process of the random walk test generation procedure is similar to the Galton board experiment. Consider the following n level decision tree, from top to bottom, each step has two choices, either going left or right. Each node represents a state and the edges represent the decisions. There are some states that can be reached from multiple paths. Take the middle state in the second level for example, it can be reached by first choosing left then right, or by first choosing right then left. There are also some other states that have more strict requirements on the decision making. For example, the state circled in red requires always select the left choices to reach, with the probability of $\frac{1}{2^n}$.

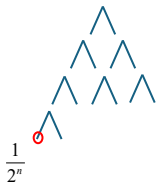

Figure 3.2: Random decision tree

If each time we start from the top and randomly make decisions, those states in the middle will be reached more frequently than those in the corners. However, if we can start from some middle states, it would be easier to reach some corner states, and hence the states reached in the end will be more diverse. This is where our fuzzer come to help. An intuitive description of the fuzzer is: whenever the fuzzer reaches a new state (i.e. finds an interesting execution graph), it mutates one of the decisions made. For the next iteration, the fuzzer replays the decision making until the mutated point, change the decision as mutated, and continues randomly afterwards. As shown in Figure 3.3 (a), suppose the red path is the previous exploration and the fuzzer mutates the second decision from going right to left (b). Then for the next exploration, it replays until the mutated choice and the probability of reaching the left corner state becomes $\frac{1}{2^{n-2}}$ (c).

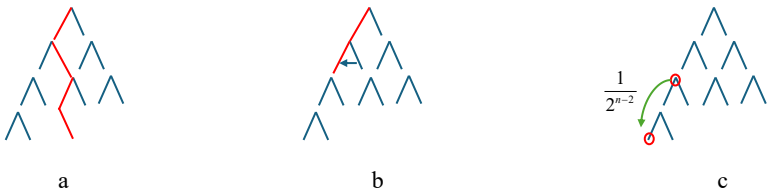

Figure 3.3: Mutation

Relating this to the randomized testing procedure, each parent node in the decision tree represents an intermediate state in the process of constructing a graph, while the leaf nodes represent completed graphs. The edges correspond to the scheduling or rf decisions made during exploration. Each time, the random walk tester restarts from the beginning (top of the tree) and randomly generates execution graphs independently. However, due to the previously listed challenges, it often tends to generate a subset of frequently occurring execution graphs, sometimes repeatedly, leaving hard-to-find executions unexplored. We consider this approach inefficient and utilize fuzzing techniques to improve this.

The fuzzer aims to improve the performance of randomized testing approaches. Some random testers, such as C11Tester or GenMC in its estimation mode, use random walk testing. In contrast, exhaustive model checkers like GenMC explore all possible executions. Exhaustive checkers are useful when the search space is limited, typically when the program under test is not too large. Randomized testers are usually employed for testing larger programs. However, the drawback of this approach is that it does not retain state information between explorations, leading to redundant efforts. In our fuzzing algorithm, we track the explored execution graphs and mutate infrequent ones to guide the tester in covering a larger fraction of the graph search space.

3.2 Example

In this section, we present an example of the fuzzing approach on the execution graphs, using the 3.3 program with release-acquire pairs.

```
atomic<int> x = {};

void thread1() {
    x.store(1, release);     // w1
    x.store(2, release);     // w2
}
void thread2() {
    auto r1 = x.load(acquire);     // r1
    auto r2 = x.load(acquire);     // r2
}
```

Listing 3.3: Fuzzing example

Suppose during exploration, one execution graph is constructed as shown in Figure 3.4, where the labels of events and relations are omitted. The read values in `thread2` are both 2. In this execution, `r2` has only choice to read from, which is `w2`, since `r1` has already read from `w2` and `r2` reading from w1 will introduce a cycle in the graph. However, `r1` can have two choices: `w1` and `w2`. If the fuzzer consider this execution as interesting and change the rf choice for `r1`, a new execution can be revealed in the next iteration, as shown in Figure 3.5.

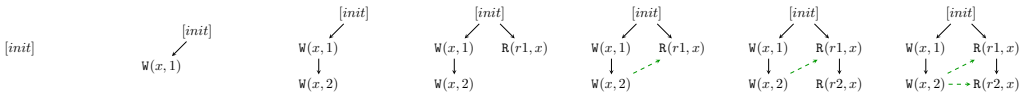

Figure 3.4: Construction of an execution graph

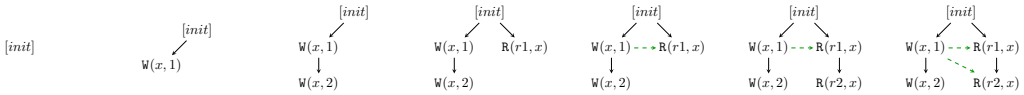

Figure 3.5: Mutate the previous execution graph and re-explore

3.3 Overview

This section provides a high-level overview of the fuzzing algorithm. The implementation details and evaluation results will be presented in later chapters.

The fuzzer uses partially constructed graphs, represented by ”prefixes” in the following context, as guidance for further explorations. As shown in Algorithm 1, the fuzzer executes the program P a total of N times (line 1), generating an execution graph each time after exploring the search space. It maintains a set of prefixes, initially empty (line 3). At the beginning of each exploration, it picks a prefix from the prefix set (line 7). If no prefixes are available in the set, the exploration will be entirely random (line 10). With a selected prefix, the fuzzer replays the execution up to the end of the prefix and randomly constructs the remaining part of the execution graph (line 8). When the exploration is finished, an execution

graph is constructed, and the fuzzer determines whether this graph is interesting according to certain metrics (line 12). A graph is considered interesting if it is a new execution graph or contains new relations or events. The interesting graph is then mutated to generate new prefixes (line 13). These new prefixes are added to the prefix set for future use (line 14). For example, the fuzzer might modify a reads-from relation in the graph and remove the invalid portion following that relation to create a prefix. The fuzzer may also dynamically discard some prefixes based on their effectiveness in finding new interesting graphs. New graphs are added to the graph set (line 16) and are ultimately outputted at the end (line 18).

Algorithm 1 Fuzzing algorithm

```
 1: Input: Program P and number of explorations N
 2: Output: N execution graphs
 3: prefix_set ← ∅
 4: graphs ← ∅
 5: for i ← 1 to N do
 6:     if prefix_set ≠ ∅ then
 7:         p ← get_prefix(prefix_set)
 8:         g ← explore_from_prefix(p)
 9:     else
10:         g ← explore_randomly()
11:     end if
12:     if is_interesting(g) then
13:         p′ ← mutate(g)
14:         prefix_set ← prefix_set ∪ p′
15:     end if
16:     graphs ← graphs ∪ g
17: end for
18: return graphs
```

The following are a few notes regarding this algorithm.

***N* as an input parameter** The number of unique execution graphs in the outputted graph set is a major metric for evaluating the fuzzer's searching ability. The algorithm takes a fixed number of iterations, N, as an input parameter. This facilitates the comparison of algorithmic efficiencies regardless of implementation details. In practice, execution time is also an important consideration. In our fuzzer implemented with GenMC, described in Chapter 5, we also evaluate the fuzzer by running it for a fixed time budget and count the number of unique executions found.

Counting the number of unique graphs In order to count the number of execution graphs, we will also need a hash function that maps the set of graphs, G, to a subset of integers, H. This hash function, $h : G \to H$ should be a bijection. That is, it should satisfy the following properties:

- h is surjective: The same execution graphs should always have the same hash values. This requires the hash function to exclude irrelevant information, such as the timestamps of events, which is related to specific explorations.
- h is injective: Different execution graphs should have different hash values. This requires the hash function to include enough information from the graphs, such as the rf relations. Ideally, there should be no hash collisions, but as this is a low-probability event, it can be ignored within the scope of our work.

If h is bijective, the size of the set of hashes, $|H|$, equals that of execution graphs (i.e. they are equinumerous), and can thus be used to count the number of distinct graphs. In our implementations, we use different hash functions due to the differing internals of the C11Tester and GenMC, but both are considered bijective functions.

The is_interesting Function This function evaluates execution graphs as a feedback metric in the fuzzing loop. Conceptually, it serves as a connecting point between the beginning and end of the exploration procedure in two folds:

- Backwards: It assesses the performance of mutated prefixes based on their ability to find interesting graphs. Prefixes that lead to more interesting graphs should be prioritized for use.
- Forwards: It determines whether an execution graph is interesting, which helps the fuzzer decide whether to mutate it. An interesting execution graph indicates the potential for discovering other interesting graphs through mutation.

In the following chapters, we present two fuzzers based on C11Tester and GenMC, both of which implement the fuzzing algorithm, with different implementations for functions such as is_interesting, mutate, explore_from_prefix and the hash function.

Chapter 4

Fuzzing with C11Tester

C11Tester is an automatic testing tool that supports a large fragment of the C/C++ weak memory model. This chapter presents an overview of C11Tester and customization points of its pluggable framework. It then describes the fuzzer's implementation. Finally, we show the evaluation results on several benchmarks.

4.1 Overview of C11Tester

C11Tester contains the following basic constructs: instrumentation, scheduler, consistency checker and race detector.

C11Tester uses an LLVM pass, CDSPass, to instrument all atomic operations, non-atomic accesses to shared memory locations, thread functions, mutex operations, and fence operations by inserting corresponding function calls. The compiled program is linked with a dynamic library containing these function calls. C11Tester implements a thread library that supports the same APIs of C++'s standard library and POSIX thread library. The user's thread function calls will be mapped into user space fibers, instead of kernel space threads. Thus, all threads are executed sequentially, with scheduling managed by a central scheduler provided by C11Tester. A context-switching approach is used to simulate thread interleavings and improve efficiency. Each atomic operation, thread creation and joining, mutex locks and unlocks, and memory fences will create an Action object, containing its type, value, memory order, thread ID and other runtime information. Since all threads are sequenced during execution, each action has a unique sequence number. The relations between actions, such as read from and synchronized with, are also maintained by the Action class.

Although C/C++ programs under weak memory models do not use scheduling semantics, C11Tester does contain a central scheduler. It is worth noting that the scheduler does not simulate schedulings of threads, instead, it is designed to check actions of different threads in a total order. This is based on the implementation of mapping user's threads into fibers and checking them sequentially. All actions within the same thread will be checked following their sequence order, as a step, and the scheduler decides whether to check actions of other threads in the next step. Hence, two decisions will be made at each step, the

behavior of the current action and the next thread to select. If the current action is a read, C11Tester will choose a write action for it to read from. Since the read-modify-write operations are instrumented with two functions, a read and a write, the scheduler ensures these two actions are atomic, i.e. they are checked without switching to other threads in between.

Memory locations are divided into two types: atomic locations and non-atomic locations. Non-atomic access to shared memory locations is the source of data races. C11Tester implements a race detector to check data races. The race detector allocates a chunk of shadow memory for each of the non-atomic memory locations. Loading from or writing to non-atomic shared variables is instrumented with specific functions, which will register the current thread's state to the shadow memory. When multiple concurrent accesses to some shared non-atomic memory with at least one of them being a write is detected, a data race will be reported.

Here, we use an example to demonstrate the process of C11Tester's random testing procedure. In the following content, we use the word 'consume' to mean that the tester executes an action without pausing to make scheduling and rf decisions.

```
#include "librace.h"
#include <atomic>
#include <thread>
uint8_t data = {};
std::atomic<int> flag = { 1 };

// thread 2
void t2() {
    store_8(&data, 0);          // cds_store8
    flag.store(1, mo_release);
}

// thread 3
void t3() {
    if (flag.load(mo_acquire))
        store_8(&data, 1);
}

// thread 1
int main() {
    std::thread thrd2(t2);
    std::thread thrd3(t3);
    thrd2.join();
    thrd3.join();
}
```

Listing 4.1: P3

In this program, there are two shared variables: a non-atomic variable `data` and an atomic flag. The `librace.h` header is provided by C11Tester and contains the instrumented function `store_8` for non-atomic variables. The initialization of `flag`, which is non-atomic, deliberately uses 1 as the initial value. Since the C++ standard thread library internally

calls POSIX APIs, the thread-related function calls will be hooked by C11Tester's dynamic library.

After compilation, the random tester launches the program. After initializing the global variables, the checker creates the first thread, which is the `main` thread, and assigns an tid, 1, to it. The first event is the creation of `thrd2`. After `thrd2` is created and assigned an tid of 2, two threads, `main` and `thrd2`, are active. The tester will randomly pick a thread to continue. Suppose the `main` thread is selected; the next event of it is the creation of `thrd3`. After creating `thrd3`, the next event of main is joining `thrd2`, making `main` inactive. Now, the current active threads are: `thrd2` and `thrd3`.

If the tester selects `thrd2`, the two events in it, writing to `data` and `flag`, will be consumed, and the `thrd2` is finished. Now, `thrd2` becomes joinable and the currently active threads are: `main` and `thrd3`. Suppose `main` is selected and after joining `thrd2` it becomes inactive again. Then, `thrd3` becomes the only active thread so it will be executed without the need to make a choice. The first event in `thrd3` is loading the value of `flag`. Since the store of `flag` in `thrd2` has already been explored, the non-atomic initialization of `flag` is not the latest in the modification order. Hence, only the atomic store is visible to this load, and the reads-from relation also forms a synchronization-with relation between the two thread. When writing to the non-atomic `data`, the race detector checks previous access to that location. The store in `thrd2` happens before the read in the current thread, which happens before the current store, so all events are totally ordered and no data race will be reported.

Considering another case, suppose the tester selects `thrd3` after after both threads have been created. The load of `flag` will be executed and only the initialization of `data` is visible to it. After reading the initial value, the store of `data` is consumed. When `thrd3` is finished, the stores in `thrd2` will be executed. When writing to `data`, the race detector checks previous actions related to that memory location. The store in `thrd3` has already been explored, and it has no synchronization with the current store. Therefore, the two stores are concurrent, and a data race is reported.

One optimization implemented by C11Tester is that it will consume writes with release and relaxed memory orders, only pausing on SC writes. This is because the sc order is part of the C/C++11 memory model, so different sc order will result in different executions. Moreover, consuming weak writes does not prevent covering possible executions. Consider the following example4.2,

```
std::atomic<int> var = {};        // (w0)
void t2() {
    auto _ = var.load(acquire); // (r1)
}
void t3() {
    var.store(1, sc);             // (w1)
    var.store(2, release);        // (w2)
    var.store(3, release);        // (w3)
}
```

Listing 4.2: P4

after creating two threads, if t2 is selected first, r1 can only read from the initial value. If t3 is selected, the first event, w1, in this thread will be executed. The next two events are release writes, so the tester will not re-select threads after executing w1. Instead, it will execute w2 and w3 until t3 is finished. Then, it finds that the only active thread is t2, so it selects t2. Now, the set of available writes for r1 to read from becomes: w1, w2 and w3, and r1 is free to randomly select one of them. Although making thread decisions after each event is also feasible, the optimized version is more efficient and still able to cover all four possible executions.

Another optimization performed by C11Tester is that it avoids backtracking during exploration. To achieve this, it performs consistency checking at each step to ensure that it is valid. It makes decisions based on the implications of the memory model and the already explored events. In 4.2, for example, if t3 has been explored, when choosing the rf relation for r1 in t2, the rf-set only contains w1, w2 and w3, where w0 is excluded. If r1 had selected w0 and the random tester continued until some point where the execution graph is found to be infeasible under the memory model, it would have to perform backtracking to make a different choice for r1. Consider another example 4.3,

```
std::atomic<int> var = {};        // (w0)
void t2() {
    auto _1 = var.load(acquire); // (r1)
    auto _2 = var.load(acquire); // (r2)
}
void t3() {
    var.store(1, sc);             // (w1)
    var.store(2, release);        // (w2)
    var.store(3, release);        // (w3)
}
```

Listing 4.3: P5

if r1 has already selected w3 to read from, only w3 will be included in the rf-set for r2, because w3 $\xrightarrow{\text{rf}}$ r2 implies that w0-2 are modification-ordered before w3 and thus should not be visible to r2, which is sequence ordered after r1. Otherwise, the RR coherence will be violated. Excluding infeasible writes from the rf-set ensures the rf's to be valid, which in turn avoids the backtracking efforts.

To summarize, C11Tester will execute the program from beginning to end, randomly picking threads and writes, with the optimizations of consuming weak writes and constraining rf-sets.

4.2 Customization points of C11Tester

In C11Tester the scheduler only controls where reads can read from, and which thread to run next. These two functions can be overwritten to implement new fuzzers. The provided set already excludes invalid choices, so it is safe to select from the remaining choices in the set. Thus, we can define two types of mutations: changing the next thread to be added or changing the write from which a read event reads.

4.3 Fuzzer implementation

To implement a fuzzer as described in Algorithm 1, several specific functions need to be defined:

- A hash function that computes a unique identifier for an execution graph, which is used to indicate whether the execution graph has been encountered before and to count the number of unique executions that are found in the end of N iterations.
- A function that returns a boolean indicating whether the execution graph is interesting.
- A function that mutates the previous execution graph and produces a prefix of the mutated graph.
- A function that enforces the prefix, i.e. replays until the mutated choice.

The hash function for execution graph encodes: event types, memory orders, and reads-from relations. It first iterates through threads by their thread IDs, and for each event in each thread, it computes the hash of its event type and memory order (for atomic operations). If the event is a read event, it also combines the hash of the write event it reads from. Note that in this hash function, the modification order is not included, although it is part of the execution graph under many memory models. This is because the hash function only cares about observable behaviors, such as read values, which may be influenced by modification orders. However, changing some modification orders may or may not change the observable behaviors. The hash function is defined in Algorithm 2 as follows:

Algorithm 2 Hash of execution graphs

```
1:  Input: Execution graph g
2:  Output: Hash value h
3:  h ← 0
4:  for each i from 0 to g.max_tid() do
5:      events = g.get_events_in_thread(i)
6:      for each e ∈ events do
7:          h ← hash_combine(h, e.type)
8:          h ← hash_combine(h, e.memory_order)
9:          if e is a read event then
10:             h ← hash_combine(h, e.rf)
11:         end if
12:     end for
13: end for
14: return h
```

The is_interesting function returns true if the hash of the current execution graph is not covered in previous explorations. This is the least restrictive metric, alternatively, it can be defined as returning true if new rf relations has been covered. Another possible definition is

to check whether the execution graph reveals a new bug, which is biased toward searching for bugs. In our definition, we aim to find more new executions, with more new buggy executions being a by-product.

The mutate function uniformly picks a fixed number of decisions, including threads and rf's, from multiple choices in the provided set. For each of them, it mutates the selected decision and discards the rest decisions to produce a prefix. Since randomness is used only in the functions for controlling rf and selecting the next thread, the choices of threads and writes will be uniquely mapped to an execution graph. Hence, the prefix of a decision trace also defines a prefix of an execution graph.

The mutated prefixes will be added to a prefix set. The replaying function chooses a prefix from the set and enforces C11Tester to make the same decisions in that prefix. After enforcing the prefix, C11Tester switches to random mode and continues random exploration until the graph is completed.

4.4 Benchmarks

The fuzzer and C11Tester are tested under the benchmarks described below. Some of them are collected from open source libraries, others come from the original C11Tester's benchmark set, which are also taken from open source libraries, internet discussions and papers. Here is a list of descriptions for these benchmarks:

barrier It comes from a solution of StackOverflow discussion. It implements a spinning barrier that halts a fixed number of threads and releases the barrier when all threads are waiting. It maintains a counter for the number of threads that are waiting currently and a step state that counts the number of barrier synchronizations. It was injected with a bug of using the relaxed memory order of the counter.

chase-lev-deque An implementation of the Chase-Lev deque data structure using C11 primitives. It was published in [42] but was found to have a bug in its implementation, due to the use of relaxed operations of fences.

mpmc-queue A multi-producer, multi-consumer queue implementation from a blog post [14].

linuxrwlocks A reader-writer lock implementation from the Linux kernel. An rwlock only allows one writer at a time but can allow multiple readers to access the protected data.

mcs-lock A list-based contention-free lock originally proposed by Mellor-Crummey and Scott[53]. The implementation comes from [13]. In this data structure, each thread maintains a node of a queue. When a thread wants to acquire the lock, it asks the mutex to set its tail of the queue to be the thread's node. When other threads lock, they have to wait for the tail to be removed by the mutex.

dekker A Dekker's critical section algorithm implemented with fences [74]. This algorithm ensures only one thread can enter critical section at a time. A shared turn variable records which thread is taking its turn. Each thread has a flag variable to indicate its state. Before entering the critical section, a thread should first raise its own flag and then check whether the turn is set. All atomic operations on the turn and flags are relaxed but fences are used to synchronize concurrent accesses.

rwlock Another rwlock implementation similar to linuxrwlocks.

seqlock A sequence lock implementation. The lock protects some shared data using an atomic counter, initialized to 0. A writer increments the counter twice, both at the beginning and end of writing. Hence whenever the counter is odd, there must be some other thread modifying the data, so other threads have to wait.

bipartite-buf A single-producer-single-consumer test for a bipartite buffer implementation written in C++11, customazed from [58]. The bipartite buffer is a variation of the ring buffer which allows in-place processing with linear space guarantee.

left-right A generic implementation [65] of the LiftRight algorithm[66]. The algorithm functions similarly to the reader-writer lock but ensures non-blocking for reads. It uses two instances for the protected data, one of which is used by all reader threads. The writer thread will work on the other instance and after writing is finished, two instances are switched.

ring-buf A ring buffer that supports adding or removing multiple objects simultaneously [58]. It maintains two indexes, a read index and a write index of the buffer. After adding or removing, these two indexes will be updated to appropriate values.

4.5 Evaluation and discussion

Research questions The following research questions are listed to evaluate the fuzzing algorithm.

RQ1 Does the fuzzer detect the bug ealier than the random tester?

RQ2 Is the fuzzer able to cover a larger range of execution graphs compared to other random-based exploration strategies?

RQ3 When the program has bugs, does the ability to detect more bugs come as a byproduct of covering more execution graphs?

RQ4 How does the fuzzer introduce overhead in C11Tester in real-world applications?

To address the research questions, we use the benchmarks to test our fuzzer and other approaches with same number of iterations, N, and compare their results. Unless otherwise stated, $N = 10000$.

4.5.1 RQ1: Fuzzer vs C11Tester

One important metric for a fuzzer is whether it is able to detect the bug effectively. In this section, we use several programs with some hard-to-find bugs and examine the iterations taken to find a bug for the first time, denoted as N_{bug1}. The smaller N_{bug1} is, the more effective the fuzzer or the tester is. We run both the fuzzer and the random tester for 20 times and record their average iterations taken to find the bugs. Table 4.1 shows N_{bug1} for the fuzzer and the random tester, where the dash indicates that the bug was not detected in all tests.

Benchmarks	long-race	mp	P1	reorder_10
C11Tester	-	686	678	176
C11Fuzzer	453	210	48	81

Table 4.1: Iterations taken to detect the first bug

Take the long-race benchmark, for example. This benchmark is taken from the rff's repository, customized with relaxed memory operations, as shown in Listing 4.4.

```
std::atomic<size_t> sum{ 0 };
std::atomic<size_t> dif{ 0 };

// thread 2
void* sub_worker(void* arg) {
  if (sum.load(relaxed) == 1) {
    dif.fetch_sub(1, relaxed);
    if (sum.load(relaxed) == 2) {
      dif.fetch_sub(1, relaxed);
      if (sum.load(relaxed) == 3) {
        dif.fetch_sub(1, relaxed);
        if (sum.load(relaxed) == 4) {
          dif.fetch_sub(1, relaxed);
        }
      }
    }
  }
  return NULL;
}

// thread 3
void* add_worker(void* arg) {
  sum++;
  if (dif.load(relaxed) == -1) {
    sum.fetch_add(1, relaxed);
    if (dif.load(relaxed) == -2) {
      sum.fetch_add(1, relaxed);
      if (dif.load(relaxed) == -3) {
        sum.fetch_add(1, relaxed);
        if (dif.load(relaxed) == -4) {
```

```
            fprintf(stderr, "Bug found\n");
            abort();
          }
        }
      }
    }
    return NULL;
}
```

Listing 4.4: long-race

Hitting the bug requires two workers to alternately modify the shared variable in a very strict order (we also tested the long-race benchmark with rff, and it took 15 thousand iterations to detect it). There are a total of 9 different possible executions of this program. Figure 4.5.1 shows how many times each execution graph are explored in N iterations. It can be observed that the random tester is heavily biased towards the first and second executions, which together take up 96.5% of N. In contrast, the fuzzer exhibits a more ”flattened” frequency plot on executions and covers all 9 execution graphs. This more even distribution is due to mutations that target those infrequent executions during exploration.

Figure 4.2 shows the number of unique executions and bugs found in first 1000 iterations. It can be observed that the fuzzer detects the bug earlier than C11Tester and also covers a higher range of executions.

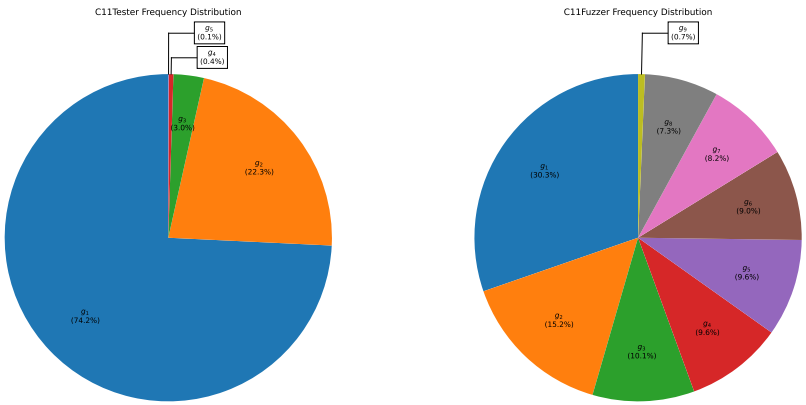

Figure 4.1: Frequencies of execution graphs

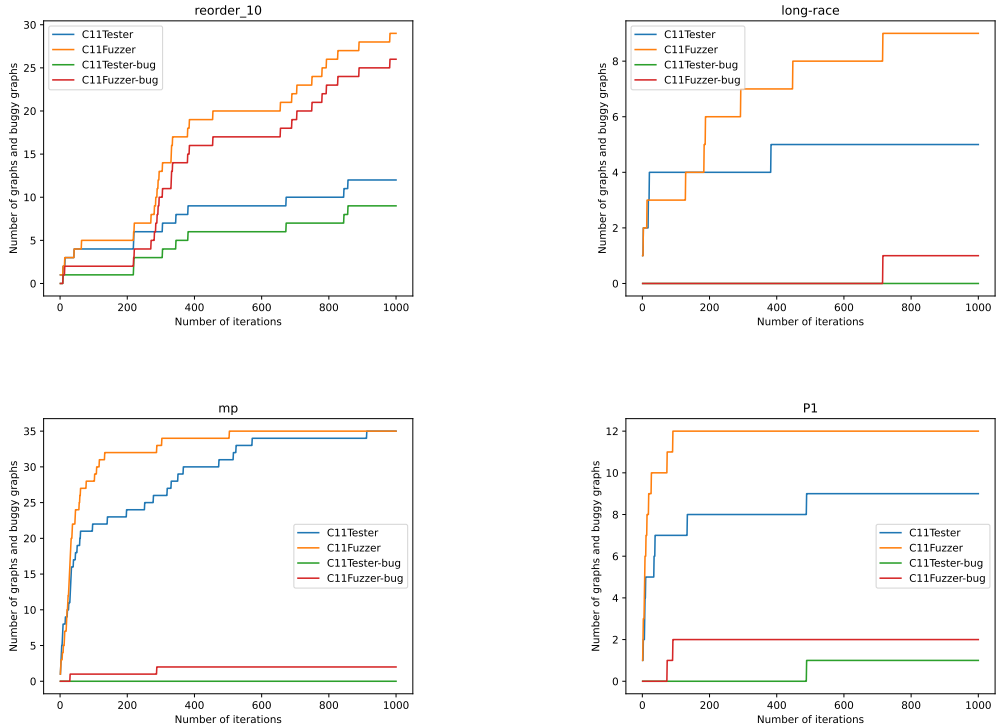

Figure 4.2: Buf finding plots

4.5.2 RQ2: Fuzzer vs C11Tester

Using the 11 benchmarks, we evaluate the ability of both the fuzzer and C11Tester to find unique execution graphs. The metric used is the number of different execution graphs discovered over N iterations. Table 4.2 and Table 4.3 show the number of unique executions found by two approaches. It can be seen that the fuzzer is able to find a larger number of different execution graphs in a fixed number of iterations than C11Tester with the random-based searching strategy in most of the benchmarks, with the average improvement of 27.0%. The improvements are calculated by:

$$R_{improvement} = \frac{N_{c11Fuzzer} - N_{c11Tester}}{N_{c11Tester}} \times 100\%,$$

where $N_{c11Fuzzer}$ and $N_{c11Tester}$ are the number of unique execution graphs found by C11Tester and C11Fuzzer, respectively.

Figure 4.4 and Figure 4.4 draw the coverage plots for each benchmark, with orange lines representing the fuzzer and blue lines representing the random tester. It can be observed that in all of these cases, the fuzzer is able to find more unique executions. In addition, in some benchmarks, such as chase-lev-deque or mpmc-queue, the fuzzer's speed of finding executions (the slop of coverage plost) does not significantly slow down as the number of found executions grows. Some coverage plots of the fuzzer, such as those in bipartite-buf

Benchmarks	barrier	chase-lev-deque	mpmc-queue	linuxrwlocks	mcs-lock
C11Tester	6969	6244	4185	981	9703
C11Fuzzer	7741	8505	6373	1225	9994
Improvement	11.1%	36.2%	52.3%	24.9%	3.0%

Table 4.2: Benchmarks (1)

Benchmarks	dekker	rwlock-test	seqlock-test	bipartite-buf	left-right	ring-buf
C11Tester	395	9998	6137	297	5378	328
C11Fuzzer	494	9997	7962	340	6678	576
Improvement	25.1%	-0.0%	29.7%	14.5%	24.2%	75.6%

Table 4.3: Benchmarks (2)

and ring-buf, also have some ”bumps” which the random tester do not have. Such bumps are caused by some prefixes that guide to a group of new interesting executions.

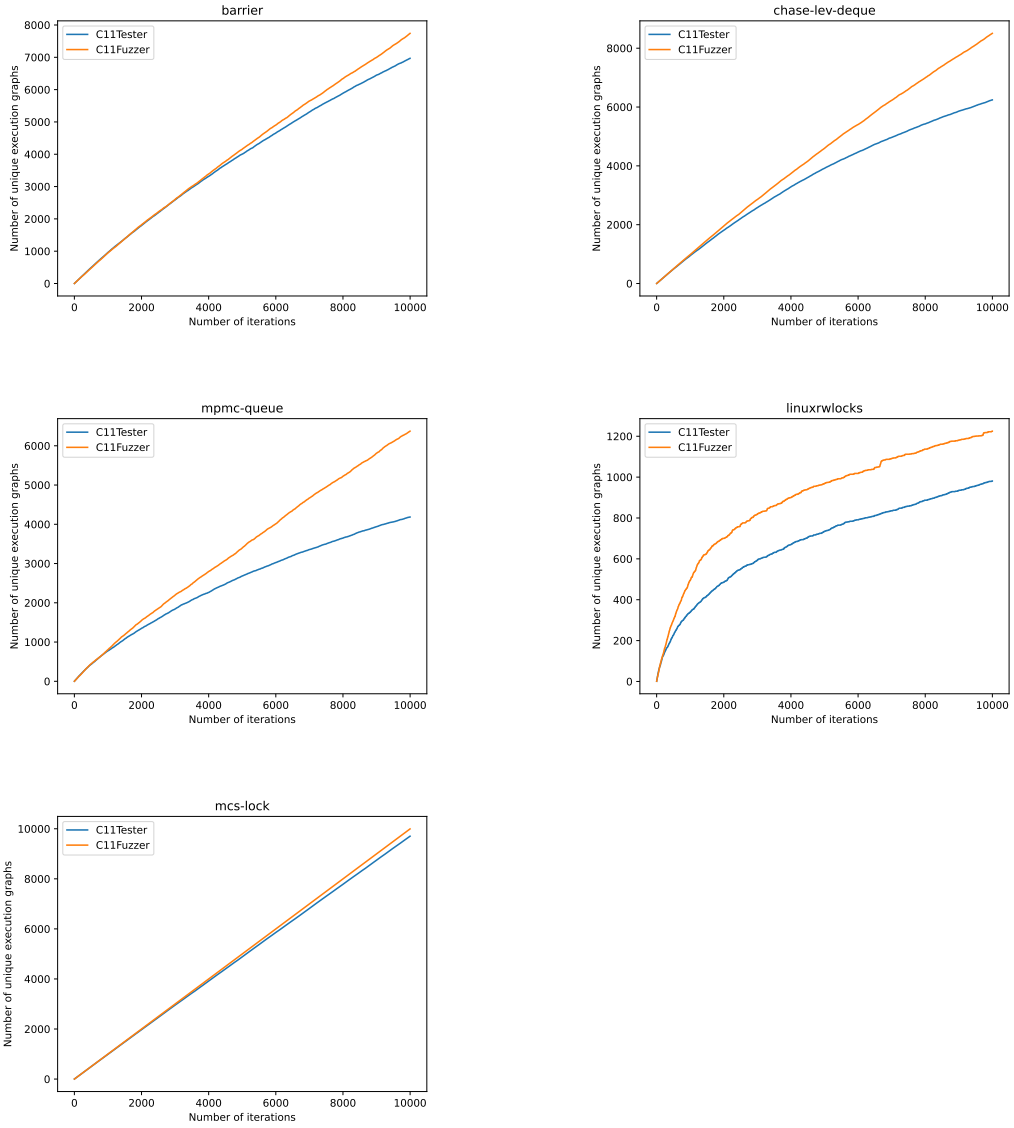

Figure 4.3: Coverage plots (1)

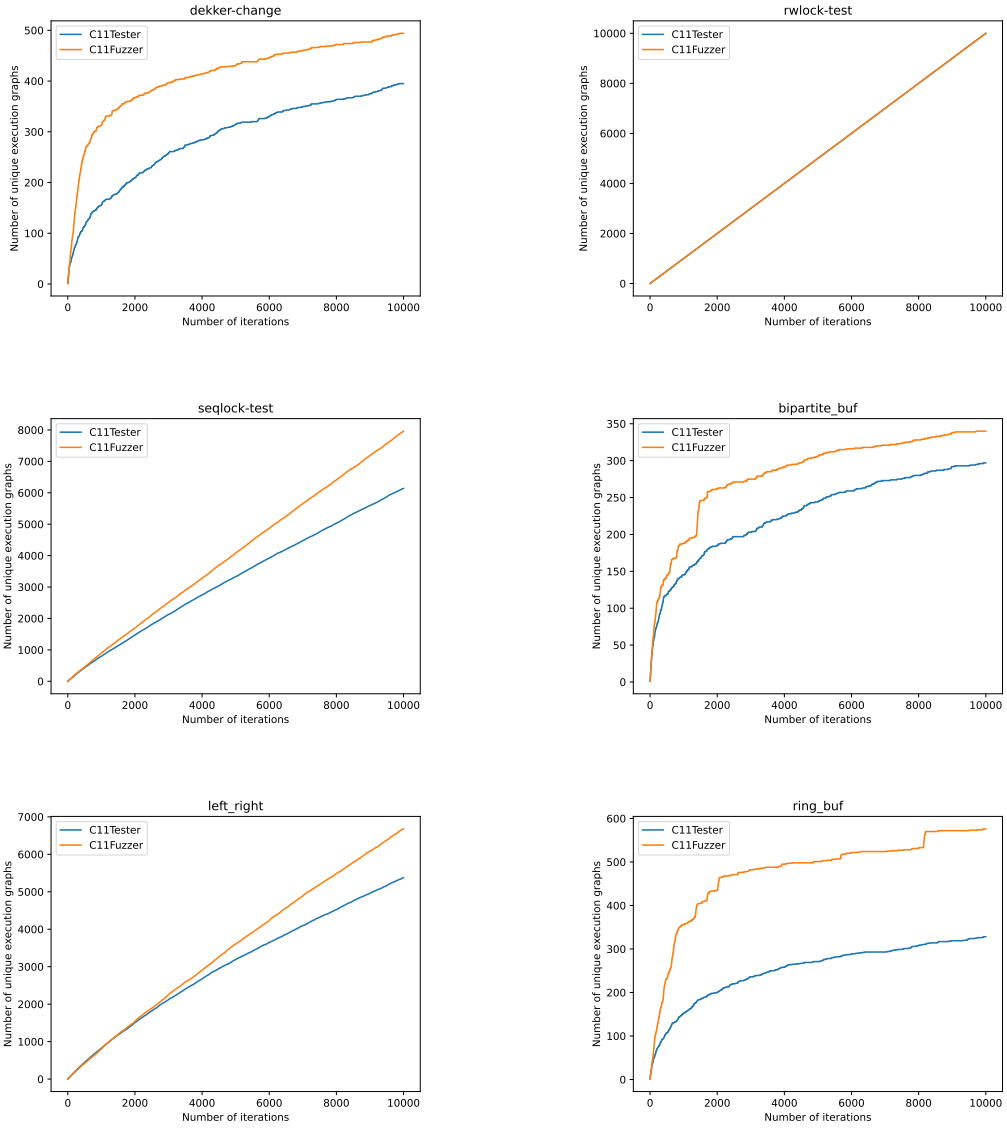

Figure 4.4: Coverage plots (2)

4.5.3 RQ2: C11Fuzzer vs PCTWM

PCTWM [24] is a state-of-the-art weak concurrency tester that expands the idea of PCT, which constrains the cope of exploring executions. It is expected that PCTWM will cover a smaller range of execution graphs than C11Tester, which performs an unbounded random search. A subset of the benchmarks described above is tested with the same configurations of bug depth and communication events, as shown in Table 4.4.

Figure 4.5 shows the coverage plots of number of unique executions found by PCTWM, C11Fuzzer and C11Tester. It can be observed that PCTWM's bounded searching scope is

Benchmark	bug depth (d)	communication (k)	history (h)
barrier	1	10	2
chase-lev-deque	2	56	1
mcs-lock	1	16	1
seqlock-test	4	18	1
linuxrwlocks	5	100	10
mpmc-queue	2	17	2

Table 4.4: PCTWM parameters

usually smaller than C11Tester's unbounded random searching scope, which is smaller than C11Fuzzer's.

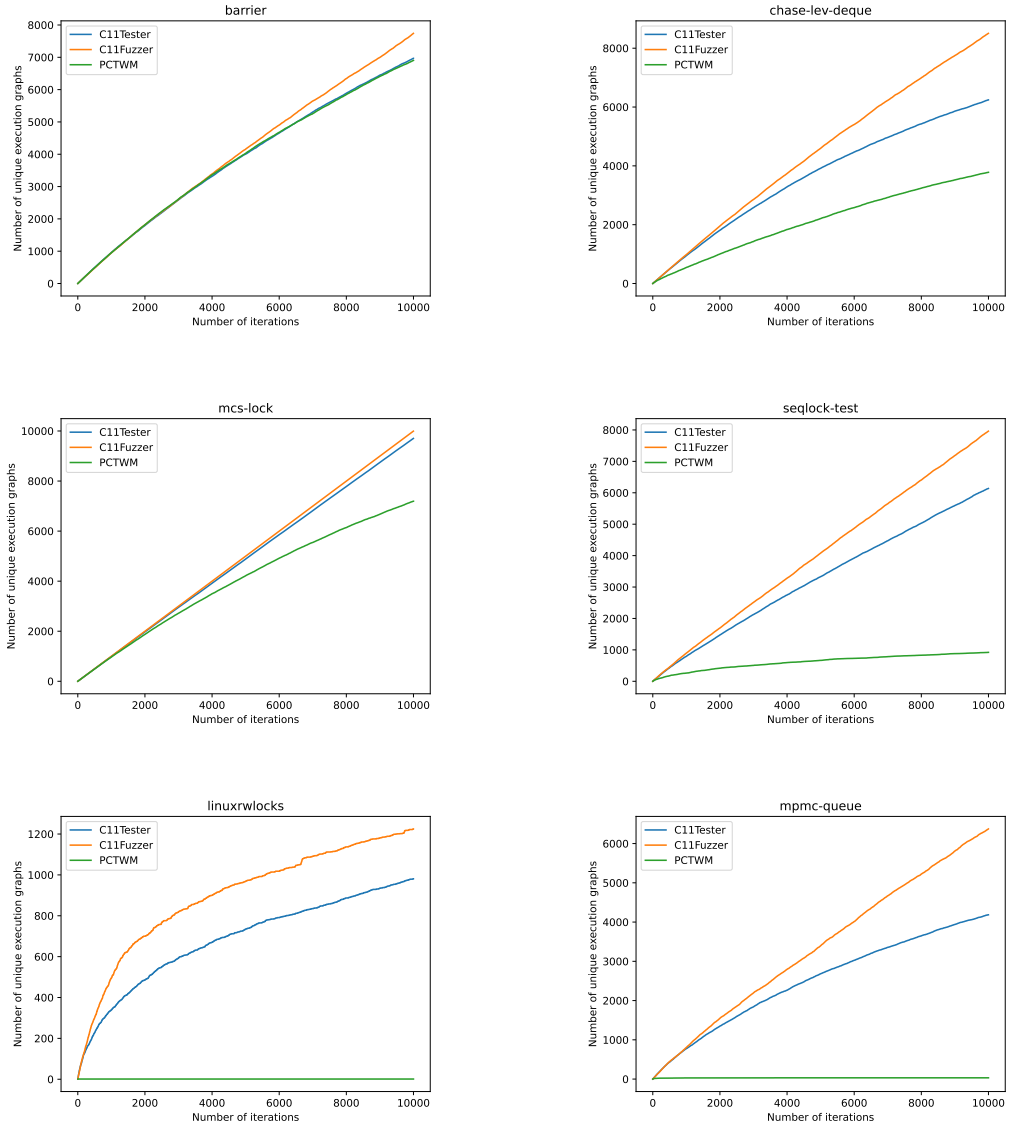

Figure 4.5: C11Fuzzer vs PCTWM

4.5.4 RQ3: Distinct bug hitting rate

The design of the fuzzer is not biased towards finding bugs, but it is also interested whether covering a wider range of execution graphs will improve the bug hitting rate. Some of the above benchmarks contain injected bugs, such as the inappropriate use of synchronization primitives, which can introduce either data races or assertion failures. The distinct bug hitting rate is computed as:

$$R_{bug} = \frac{N_{bug}}{N},$$

where N_{bug} is the number of distinct buggy executions found over N iterations. Since all benchmarks are executed N times, a higher number of distinct bugs found indicates a higher distinct bug hitting rate. We focus on this distinct bug rate instead of a naive bug rate because the latter does not reflect the fuzzer's ability to discover more bugs. It could repeatedly hit the same bug and still show a high bug hitting rate. Table 4.5 and Table 4.6 show the results from the benchmarks.

Benchmarks	barrier	chase-lev-deque	mpmc-queue	linuxrwlocks	mcs-lock
C11Tester	5789	5895	2341	976	8525
C11Fuzzer	6993	5858	5858	1220	9418

Table 4.5: Distinct bug hitting rate (1)

Benchmarks	dekker	rwlock-test	seqlock-test	bipartite-buf	left-right	ring-buf
C11Tester	5	4697	3872	297	1478	328
C11Fuzzer	5	4736	5009	340	1488	576

Table 4.6: Distinct bug hitting rate (2)

It can be observed that except for dekker benchmark, which in total has 5 different bugs, C11Fuzzer is able to find more distinct bugs in other benchmarks.

4.5.5 RQ4: Real-world applications

The fuzzer is composed of a python script that performs mutation and file IO's and a compiled binary that is hooked to the executed program. For overhead evaluation, several things are measured:

- The scripting overhead of the random version and the fuzzing version. Substracting them yields the overhead of mutation and IO's.
- The time used by the binary to load the decisions for replay.
- The time used for the binary to execute.

We test our fuzzer using a real-world application, iris, which is an asynchronous logging library that makes extensive use of atomic operations. Both C11Tester and C11Fuzzer are tested $N = 100$ times and an average of the overhead per execution is taken. Table 4.7 shows the measurement results. It can be seen that the mutation takes up most of the overhead, 7.4 seconds for each execution. A future improvement could be writing the fuzzer in a compiled language and transfer the heavy file IO into memory.

Items	script	binary	mutation	load-replay
C11Tester	1.50s	39.51s	0s	0s
C11Fuzzer	8.9s	40.5s	7.4s	2.9s

Table 4.7: Overhead (per execution)

Chapter 5

Fuzzing with GenMC

In this chapter we first present an overview of GenMC. Then we present three different mutation strategies and show their effectiveness in the end.

5.1 Overview of GenMC

GenMC is an model checker for C programs, supporting a variaty of memory models, including RC11[40], IMM[63] and LKMM[60] memory models. It uses Kater[35] to automatically generate axiomatic memory models that provides the specified interfaces. The memory models to be checked can be selected by the user via command line arguments, with RC11 being the default model. It incorporates an LLVM-based interpreter that compiles the target program into LLVM-IR (intermediate representation) and generates execution graphs in accordance with the specified memory model. Data races, assertion failures and other errors will be reported when detected. GenMC has two modes: estimation mode and verification mode. In estimation mode, the GenMC driver randomly collects a sample of execution graphs, independently, to get an estimation of the size of the search space and time to finish verification. After estimation, the driver performs an exhaustive enumeration of execution graphs in the verification mode and halts when errors are encountered. The estimation mode can be disabled by command line options, too.

Both the estimation and verification modes share the same set of interfaces, with some functionality turned off during estimation. Since the fuzzer aims to improve randomized testing, here we mainly describe GenMC's estimation mode and show its customization points for our fuzzer.

The core component of GenMC is its driver. The driver is responsible for calling the interpreter to transform the target program into LLVM-IR, constructing execution graphs, checking consistency, and reporting errors. The interpreter is used to interpret the source code and keep relevant execution information. The interpreter will ask the scheduler of the driver to fetch the next instruction. Normally, the scheduler randomly picks the next thread and fetches the next instruction of that thread, with some special cases such as RMW instructions, prioritized threads, or reads that need to be rescheduled. Then the interpreter handles each instruction following the visitor pattern.

The execution graph is composed of events, each having a label indicating its position in the graph and other information about the event itself. An event can be looked up using its position, which is a pair of thread ID and its index in that thread. Both the stamp and the position uniquely identify an event in a single graph; however, the stamp is determined by the order of adding events to the graph and hence will vary across explorations, while the position is determined by the source code of the tested program.

The driver has a stack of executions, called execStack. Each execution has an execution graph instance and a workqueue. The workqueue stores the exploration operations, called Revisit, to be conducted, on the corresponding graph. The driver fetches an item each time from the workqueue and revisits it. When the workqueue is empty, the driver pops out the current execution from the execStack and continues with other executions. In estimation mode, only one kind of Revisit, RerunForwardRevisit, is used, which indicates the driver to reset the execution graph to its initial state and start over the next iteration.

The above mentioned exploration procedure is listed in Algorithm 3.

Algorithm 3 GenMC driver explore

```
1:  EE ← getInterpreter()
2:  execStack ← []
3:  while not isHalting() do
4:    /* Continue with the current graph */
5:    EE.run()
6:    r ← RerunForwardRevisit
7:    stamp ← 0
8:    pushRevisit(execStack[last], r, stamp)
9:    validExecution ← false
10:   while not validExecution do
11:     [stamp, item] ← getNextItem(execStack[last].workqueue)
12:     if item is empty then
13:       execStack.pop()
14:       if not execStack.empty() then
15:         continue
16:       else
17:         return
18:       end if
19:     else
20:       g ← execStack[last].graph
21:       cutToStamp(g, stamp)
22:       validExecution ← isConsistent(g) /*always true for graphs cut from RerunFor-
          wardRevisit*/
23:     end if
24:   end while
25: end while
```

5.2 Customization points of GenMC

In the estimation mode, the driver pushes a RerunForwardRevisit and a zero stamp to the workqueue at the end of each execution, so the graph will always be reset to an empty state, which stays at the end of execStack. It is also viable to push other Revisit objects to the workqueue and the driver will cut the graph accordingly. In addition, we could also cut the graph manually and push it together with a latest stamp so the driver will not cut it again. If the manually cut graph is consistent, the interpreter will continue and finish exploration with it. Both pushing other Revisit and manually cutting the graph serve as the mutation part of our fuzzer. The driver has a function, getRfsApproximation, that can provide a list of stores that a read can read from, so the fuzzer can pick a different store from that list.

5.3 Fuzzer implementation

Similar to what is discussed in section 4.3, several functions need to be implemented.

- A hash function that computes a unique identifier for an execution graph.
- A function that mutate the previous execution graph and produces a prefix of the mutated graph.
- A function that judges whether an execution is interesting.

5.3.1 Hash function for execution graphs

Firstly the hash function for a single event should be defined, as shown in Algorithm 4

Algorithm 4 Hashing an EventLabel

```
1:  Input: EventLabel lab
2:  Output: Hash value h = hash(lab)
3:  h ← 0
4:  pos ← lab.getPos()
5:  hash_combine(h, pos.thread)
6:  hash_combine(h, pos.index)
7:  if lab is a ReadLabel then
8:      if lab.getRf() is not empty then
9:          slab ← lab.getRf()
10:         hash_combine(h, hash(slab))
11:     end if
12: end if
13: return h
```

Then the events are iterated by thread ID and indices to compute the hash value of the graph, as listed in Algorithm 5.

Algorithm 5 Hashing an ExecutionGraph

```
1: Input: ExecutionGraph g
2: Output: Hash value h = hash(g)
3: h ← 0
4: for i ← 0 to g.getNumThreads() − 1 do
5:    for j ← 0 to g.getThreadSize(i) − 1 do
6:       lab ← g.getEventLabel(Event(i, j))
7:       hash_combine(h, hash(lab))
8:    end for
9: end for
10: return h
```

5.3.2 Mutation methods

The mutation process is composed of two steps: changing an rf relation and cutting the graph. The driver has provided a function, getRfsApproximation, that calculates a list of possible stores given a read event. It first collects a list of coherent stores restricted by the memory model. In RC11, it selects all concurrent stores and the latest store in mo before the provided read. The fuzzer first picks out all read events that have multiple store choices and pairs each read with each of its alternative stores. Then the fuzzer randomly selects one of these pairs for mutation. Here we denote the selected read event as R, its original store as S_{old}, and the newly selected store as S. In accordance to GenMC's terminology, the word "view" is used to represent a subset of events in an execution graph. Here a "cut view" represents the view of the current graph to be kept in the following cutting strategies, which serves as a prefix defined in Algorithm 1. Additionally, we use the following notations in the descriptions below:

- $preds_e$: All events in a graph that has smaller stamps than event e.
- $pporf_e$: All events in a graph that are porf predecessors of event e.

The fuzzer implements three different cutting strategies, described as follows:

Revisit cut It constructs the ReadForwardRevisit and BackwardRevisit objects and pushes them to the workqueue directly. These two kinds of Revisit's are defined in GenMC, used in its verification mode. We first compare the timestamps of R and S. If R has a greater stamp, a ReadForwardRevisit will be constructed. When the driver retrieves a ReadForwardRevisit from the workqueue, it removes all events whose stamps are greater than R. Since S's stamp is smaller, it will be kept. Additionally, the read becomes the latest event added to the graph, hence the events that may no longer be valid due to the change in R's rf will not be retained. This cut view can be denoted as $preds_R$. On the other hand, if S has a greater stamp, a BackwardRevisit will be constructed. The driver first collects all events that has smaller stamps than R, i.e. $preds_R$, similar as did in ReadForwardRevisit. Since S has a greater stamp this time, it will not be included in $preds_R$. Then the driver computes

all events that are porf predecessors of S, denoted as $pporf_S$. The cut view is the union of the two sets of events, $preds_R \cup pporf_S$ and the rest of the graph will be cut.

Minimal cut This cut strategy aims to retain as little events as possible. It only keeps those events that are both porf predecessors of R and S. The events in unrelated threads (concurrent events) will be dropped. The minimal cut view can be written as $pporf_R \cup pporf_S$

Maximal cut This cut strategy aims to retain as many events as possible. In maximal cut, the unrelated events, which are removed in minimal cut, will be kept. It only removes the events that are porf successors of the read. To get this set of events, the fuzzer iterates through all events in the graph. For each event, if the R is not a porf predecessor of it, it will be added into the set. Because the maximal cut will include more events into the cut view, some special attention needs to be paid to the "pair relations". For example, if a thread-join event is to be added into the view, both itself and its corresponding thread's thread-finish event should not be porf successors of R. The maximal cut view can be represented by $\{e \in G \mid R \notin pporf_e,\ R \notin pporf_{e'spair}\} \cup R$, where G is the current graph to be mutated.

We use an example to illustrate the above mutations. Suppose the execution graph shown in Figure 5.1 is the graph to be mutated, assuming all operations are relaxed. The numbers on the shoulder of R's and W's are the timestamps given to the events. The select read event is $\mathrm{R}^7(x)$, reading from event $\mathrm{W}^5(x,1)$.

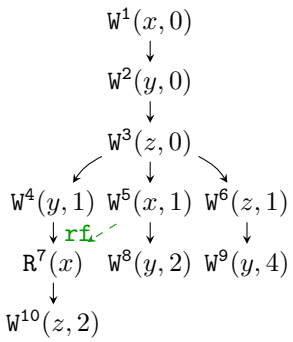

Figure 5.1: The execution graph to be cut

For revisit cut, since the read event has the greater stamp, the graph will be simply cut up until stamp = 7, shown in Figure 5.2.

For minimal cut, the fuzzer counts the porf predecessors of $\mathrm{R}^7(x)$ and $\mathrm{W}^5(x,1)$. The events in the third thread (the rightmost column) will be removed. $\mathrm{W}^{10}(z,2)$ and $\mathrm{W}^8(y,2)$ are removed since they porf successors of porf successors of $\mathrm{R}^7(x)$ and $\mathrm{W}^5(x,1)$, respectively. The resulting graph is shown in Figure cut:minimal.

For maximal cut, only the porf successors of the read, $\mathrm{R}^7(x)$, will be removed. In this graph, $\mathrm{W}^{10}(z,2)$ will be removed with all other events remained, shown in Figure 5.4.

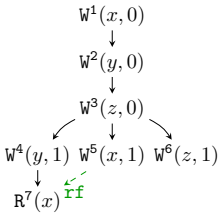

Figure 5.2: Revisit cut output

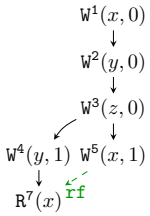

Figure 5.3: Minimal cut output

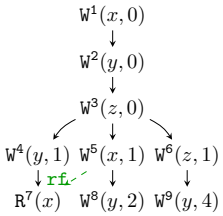

Figure 5.4: Maximal cut output

5.3.3 The is_interesting function

The `is_interesting` function is defined as follows: we first compute the relative frequency for a graph:

$$f_{\text{rel}}(g) = \frac{f_g^2}{\sum_{g_i \in G} {g_i}^2 / |G|},$$

where f_{g_i} is the frequency of occurrences of the execution graph g_i in the set of explored graphs G. For example, if the fuzzer explores graphs g_1, g_2, and g_3 2, 3, and 5 times respectively in 10 total explorations, the relative frequency of g_1 is:

$$\frac{2^2}{(2^2+3^2+5^2)/3} \approx 0.32,$$

and for g_3, it is:

$$\frac{5^2}{(2^2+3^2+5^2)/3} \approx 1.97.$$

If the relative frequency is below a certain threshold, the corresponding execution graph is considered "interesting." For example, with a threshold of 0.4, g_1 is considered interesting, while g_3 is not.

This computation provides more granularity than simply checking if a graph is new. If we only flagged a graph as interesting when it's new, it could only be considered interesting once. By using frequencies, we refine this process. However, just using frequency is not always sufficient. For example, with a threshold of 0.2 in the above case, none of the graphs would be considered interesting, even though g_1 has a lower frequency than the others. In this case, the threshold of 0.2 is too low. In another scenario, if the fuzzer discovers 100 different graphs with frequencies like 0.01, 0.02, 0.015, etc., all lower than 0.1, they would all be considered interesting. But it would be reasonable to only prioritize those with relatively lower frequencies. In other words, the threshold is too high in this case. To address these issues, we use the relative frequency, which takes into account both the graph's frequency and those of others. Squaring the frequencies amplifies differences and avoids discarding the sum, which is always 1.

5.4 Benchmarks

We use the following benchmarks to evaluate the fuzzer.

ring-buffer A ring buffer implementation in FreeBSD 8.0.0. Each thread enqueues a message and dequeues one from the buffer. The program checks the correctness and integrity of the messages. The ring buffer uses an array to store data. The enqueue and dequeue operations use compare-and-swap loops to update the queue head and tail pointers.

mpmc-queue A multi-producer multi-consumer bounded queue implementation. It maintains three state variables that keeps track of the number of reads and writes that have been started and finished. Each writer obtains an index in the queue's array buffer using cas loop and write to that position. The readers will increment the reading index and read the data. In the test, 2 readers and 4 writers are spawned.

ttaslock A spinlock called Test-and-Test-and-Set (TTAS) lock. The lock has an atomic state variable shared by multiple threads. Before locking it, a thread first loads the flag and wait until it is not set. Locking is implemented using a loop that exchanges the state value until the old value of it is not set. In the benchmark, two threads are luanched to update a non-atomic shared variable and asserts they read their values in the critical section.

treiber-stack A treiber stack[15] implementation using compare-and-swap for pushing and popping nodes.

ms-queue An ms-queue implementation similar to that in the previous chapter, written in C.

linuxrwlocks The reader-writer lock implementation of linux kernel, similar to that in the previous chapter but written in C.

P1 One thread continues to modify a shared variable and the other thread checks whether it is equal to a certain value once.

stack_bug A treiber stack with an injected bug.

long_race The same benchmark to that used in the previous chapter, in C.

mp A message passing program where multiple threads update a shared flag variable. The bug is triggered when the flag ends up with a certain value.

stack_oddeven A concurrent stack benchmark with threads pushing odd and even numbers, injected with a bug by using store operation to replace a cas loop.

stack_oddeven2 The same stack used in stack_oddeven but injected with a different bug, by using exchange operation to replace the cas loop.

buggy_queue An ms_queue benchmark injected with a bug while pushing items to the queue.

buf_ring_bug A concurrent ring buffer benchmark injected a bug by disabling the memory barrier during enqueuing.

long-assert A test case that has a complicated assertion that is hard to trigger. One thread checks the assertion involving two shared variables and the other thread modifies one of them after doing some exponential work.

5.5 Evaluation and discussion

Research questions Below lists the research questions we want to explore.

RQ5 Is the fuzzer able to explore more distinct execution graphs in a fixed number of iterations than the random tester?

RQ6 Among the three mutation strategies, which performs best? Why?

RQ7 What's the runtime performance of the fuzzer with different mutations, comparing with random testing?

RQ8 Can the fuzzer detect bugs faster compared to the random walk tester?

RQ9 Is the fuzzer able to find bugs faster than the model checker?

5.5.1 RQ5: Fuzzer vs random tester on search space coverage

To address this question, we run both the fuzzer and the GenMC in its estimation mode for 10 thousand iterations. Using the hash function defined in section 5.3.1, we count the number of distinct execution graphs they found. Table 5.5.1 shows the results of each benchmark. The improvements are computed by:

$$R_{improvement} = \frac{\frac{1}{3}(N_{revisit} + N_{minimal} + N_{maximal}) - N_{random}}{N_{random}} \times 100\%.$$

It can be observed that for all of the above benchmarks, the fuzzer is able to explore higher number of execution graphs than the random tester does. The average improvement, computed by

$$\overline{R_{\text{improvement}}} = \frac{1}{n}\sum N_{improvement}$$

is 181.62%.

Benchmark \ Strategy	random	revisit cut	minimal cut	maximal cut	$R_{improvement}$
ring-buffer	6953	16449	30803	49290	362.83%
linuxrwlocks	9658	10699	27459	20666	103.02%
mpmc-queue	23145	49506	53003	66422	143.29%
ms-queue	13746	31404	23078	32792	111.63%
treiber-stack	9029	28005	41201	53892	354.45%
ttaslock	8633	8982	10430	10245	14.51%

Table 5.1: Number of unique execution graphs of each exploration strategy

5.5.2 RQ6: Revisit cut vs minimal cut vs maximal cut

Figure 5.5 shows the execution graph coverage plots of random testing and fuzzing with three mutation strategies. It can be seen that in 5 of these benchmarks, the maximal cut finds the greatest number of execution graphs. The revisit cut performs the best in 2 benchmarks.

It is hypothesized that the performance is related to the number of events that remain in the mutated graph. Generally, the more events that are preserved in the mutated graph, the closer their relations tend to be. The random strategy, for example, can be seen as a method that cuts out the whole portion of the graph, resulting in the fewest remaining events. To better understand this relationship, we measured the average number of events in execution graphs for each benchmark, as well as the average number of events that remain after applying each cutting strategy, as shown in Table 5.5.2. The data shows that, in the benchmarks linuxrwlocks and ttaslock, the revisit cut strategy retains the highest number of events in their mutated graphs and performs best in discovering more execution graphs. Similarly, in other benchmarks, the maximal cut strategy retains the highest number of events and explores the largest number of distinct graphs. Overall, the results indicate that the number of explored execution graphs is positively correlated with the number of events remaining in the mutation strategies.

Benchmark \ Strategy	total	revisit cut	minimal cut	maximal cut
ring-buffer	102.18	81.88	41.15	94.11
linuxrwlocks	77.26	59.79	30.79	54.03
mpmc-queue	85.24	59.57	31.75	73.95
ms-queue	172.85	136.00	66.13	159.68
treiber-stack	113.50	83.98	53.51	99.49
ttaslock	65.35	59.41	21.76	51.88

Table 5.2: Number of events in original graphs and mutated graphs, on average

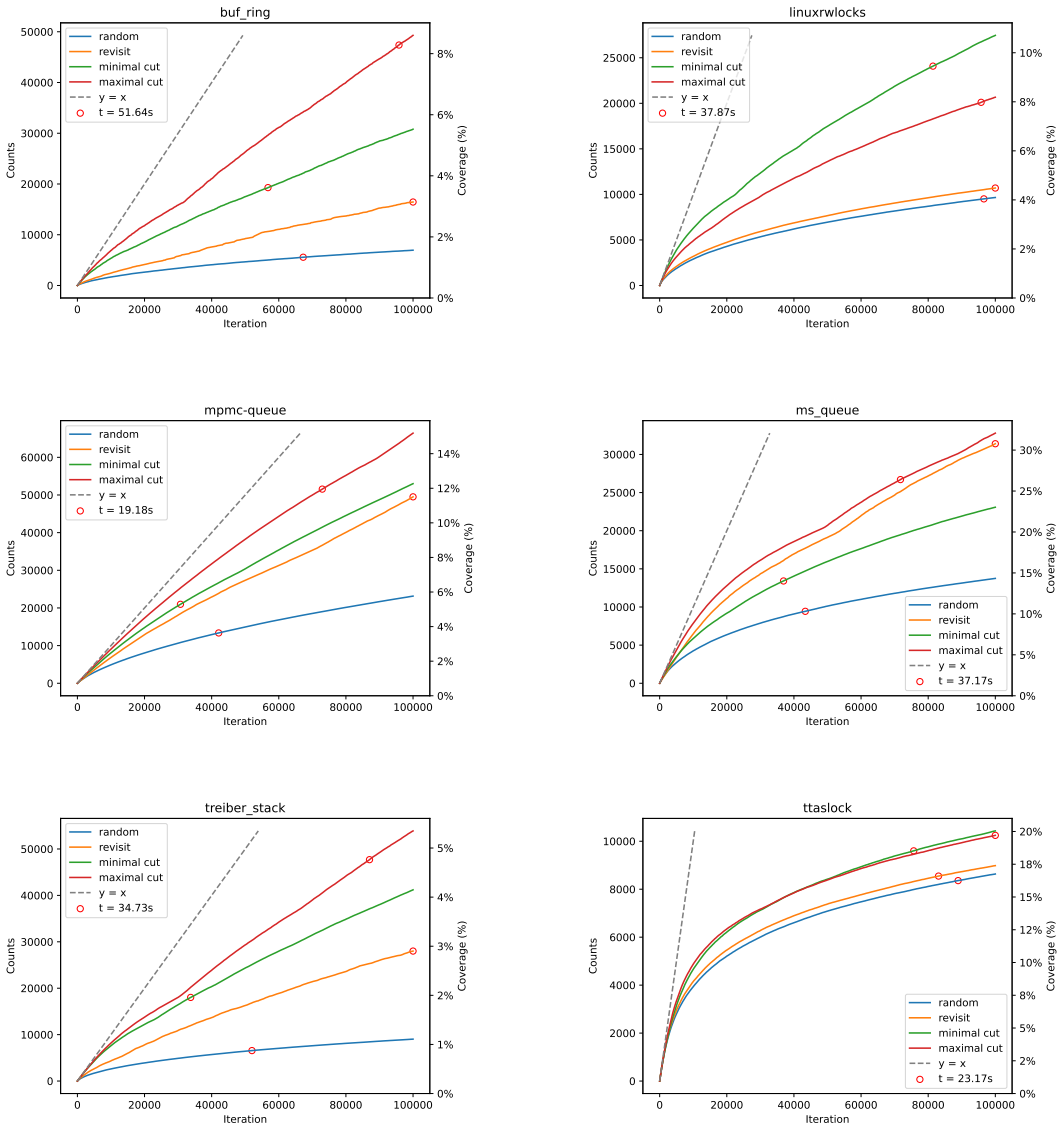

Figure 5.5: Coverage plots

5.5.3 RQ7: Runtime overhead and efficiency of 3 mutations

We examine the total time elapsed for exploring 100 thousand iterations, shown in Figure 5.6. The random testing is taken as a baseline for evaluating the overhead. It can be seen that random testing does not always take the shortest time. This is due to the mechanism of GenMC. When adding events to the graph, the GenMC driver first inspects whether there is already an event in that position. If so, it continues without the need to check consistency, pick rf's, or arrange it in the coherence order. Although the maximal cut takes many steps to compute the cut view, it still takes the shortest time in one of the benchmarks

since the driver can save effort on those remaining events. Overall, all three mutations have insignificant effects on the runtime performance, with affordable overheads.

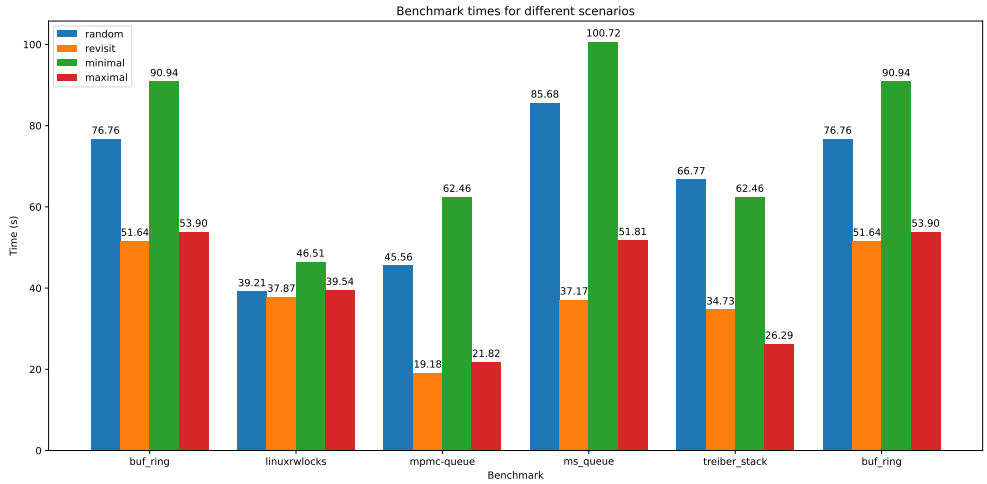

Figure 5.6: Time elapsed by various strategies

In addition, it is possible that one strategy may find more execution graphs in a fixed number of iterations than another but takes more time to complete. Therefore another concern is the efficiency of various mutations, defined as $\sigma_T = \frac{N_{graph}}{T}$, where N_{graph} is the number of distinct execution graphs, and t is a fixed time. Note that this ratio is not a constant value over time, as it becomes increasingly difficult to explore new execution graphs as time progresses. Here we set T to be 1 minute and count the number of different execution graphs explored. The results are shown in Figure 5.7. Our results indicate that the maximal cut has the highest efficiency in a majority of benchmarks, while the random tester generally has the lowest efficiency.

5.5.4 RQ8: Fuzzer vs random tester on bug detection

We use several buggy benchmarks to test the bug detection capabilities of both our fuzzer and the random tester. For each benchmark, we run the fuzzer and the random tester for a fixed number of iterations, N. When a bug is detected for the first time, we stop the testers and record the current iteration number and the elapsed time. For ease of comparison, if the bug is not detected within N iterations, we assign a placeholder iteration number equal to N. Each technique is run 20 times on each benchmark, and we calculate the averages of iterations and time, as shown in Figure 5.8 and Figure 5.9. The results indicate that the fuzzer detects bugs faster and requires fewer iterations in the majority of benchmarks.

5.5.5 RQ9: Fuzzer vs model checker

Model checkers are also used to verify the correctness of programs and detect bugs. The GenMC model checker implements an optimal algorithm that ensures no duplication while

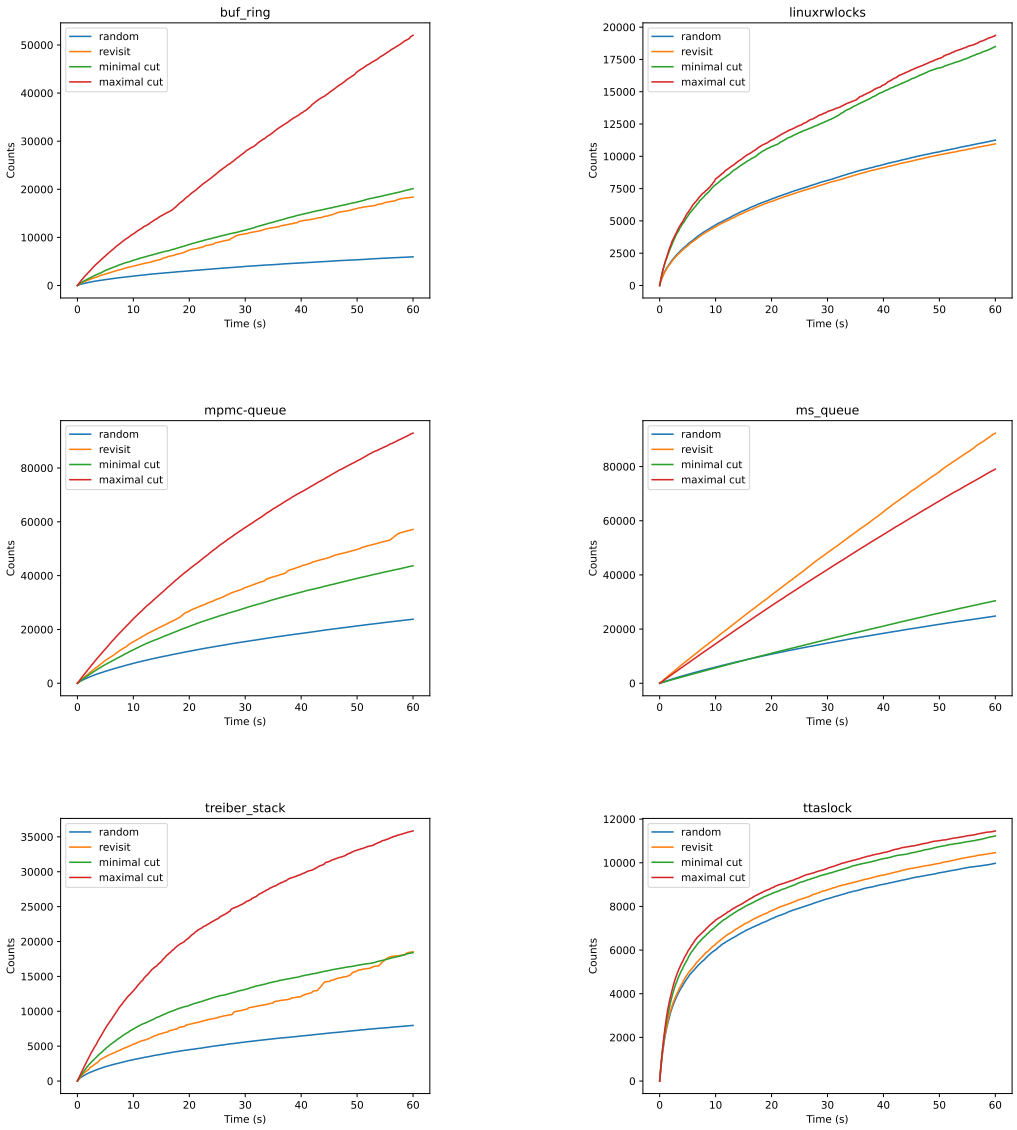

Figure 5.7: Number of executions found in 1 minute

enumerating execution graphs. However, we would like to demonstrate that in some cases, introducing randomness can be more efficient in terms of the number of iterations taken and the time elapsed for detecting a bug. Take the long-assert benchmark for example, as shown in Listing 5.1. In order to violate the assertion, `x` should read from its initial value while `y` reads the values 1, 2, 3, 4, and 5, sequentially. In the GenMC's verification process of this program, when the read of `x` event is added, the driver picks a value for it and pushes other alternative choices to the workqueue, which will be revisited later. Suppose the driver picks a non-zero value, the zero value will be tried again only after the drivers finish exploring

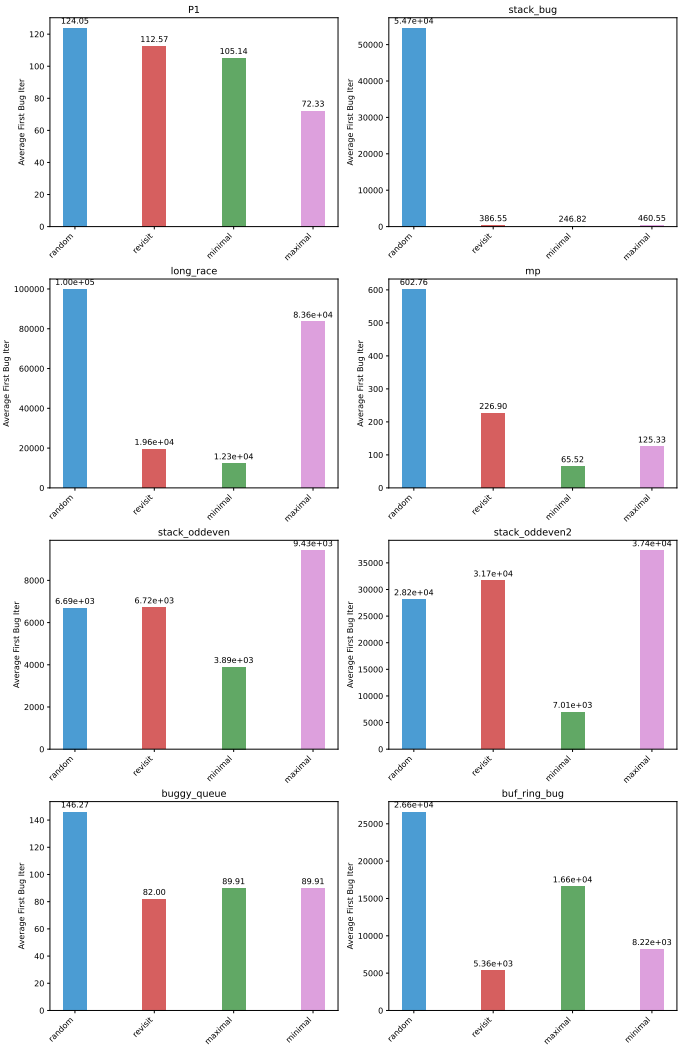

Figure 5.8: Average iterations to detect the bugs

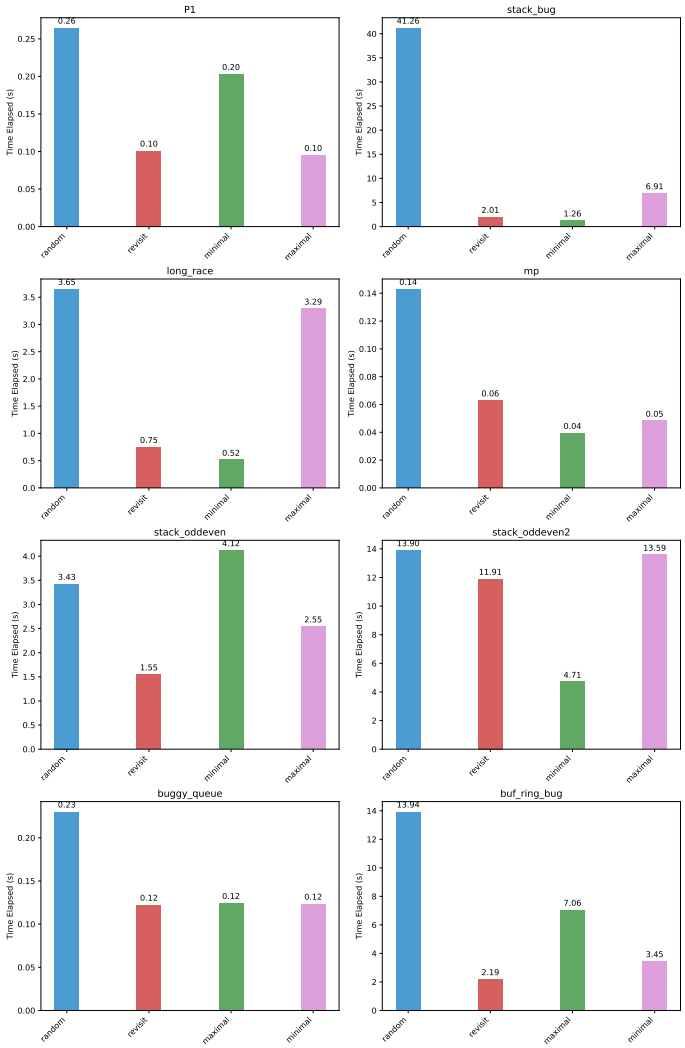

Figure 5.9: Average time to detect the bugs

graphs with the non-zero choice.

```
atomic_int x, y;

void* thread_0(void* ) {
  x = 42;
}

void* thread_1(void* ) {
  // fails when x==0, y==1, y==2, y==3, ...
  assert(x != 0 || y != 1 || y != 2 || y != 3 || y
      != 4 || y != 5);
}

void* thread_n(void* unused) {
  atomic_fetch_add(&y, 1);
}

void* thread_2(void* unused) {
  pthread_t t[N]; // N is a hyper parameter
  for (int i = 0U; i < N; i++) {  // y++, ++, ++...
    pthread_create(&t[i], NULL, thread_n, NULL);
  }
  for (int i = 0U; i < N; i++) {  // join threads
    pthread_join(t[i], NULL);
  }
  // cas on x after exponential work on y
  atomic_fetch_add(&x, 1);
}

int main() {
  pthread_t t0, t1, t2;
  pthread_create(&t0, NULL, thread_0, NULL);
  pthread_create(&t1, NULL, thread_1, NULL);
  pthread_create(&t2, NULL, thread_2, NULL);
  // join threads...
}
```

Listing 5.1: long-assert

Comparing the model checker, the random tester, and the fuzzer with revisit, minimal and maximal cuts, Figure 5.10 and Figure 5.11 show the average number of iterations (for the model checker, this is the number of graphs prior to the buggy one) and the time taken to find the bug, respectively, with `N` taken from 5 to 9. It can be observed that the random tester and the fuzzer require fewer iterations to find the bug in all cases. When `N` is small, the model checker is faster, but as `N` increases, the fuzzer and random tester become faster.

If we change the assertion to `assert(x!=0||y!=1||y!=2||...||y!=N)`, it becomes increasingly difficult to find the bug as `N` gets larger. As shown in Figure 5.13 and Figure 5.12, the fuzzer with different mutations can take less time and iterations than the ran-

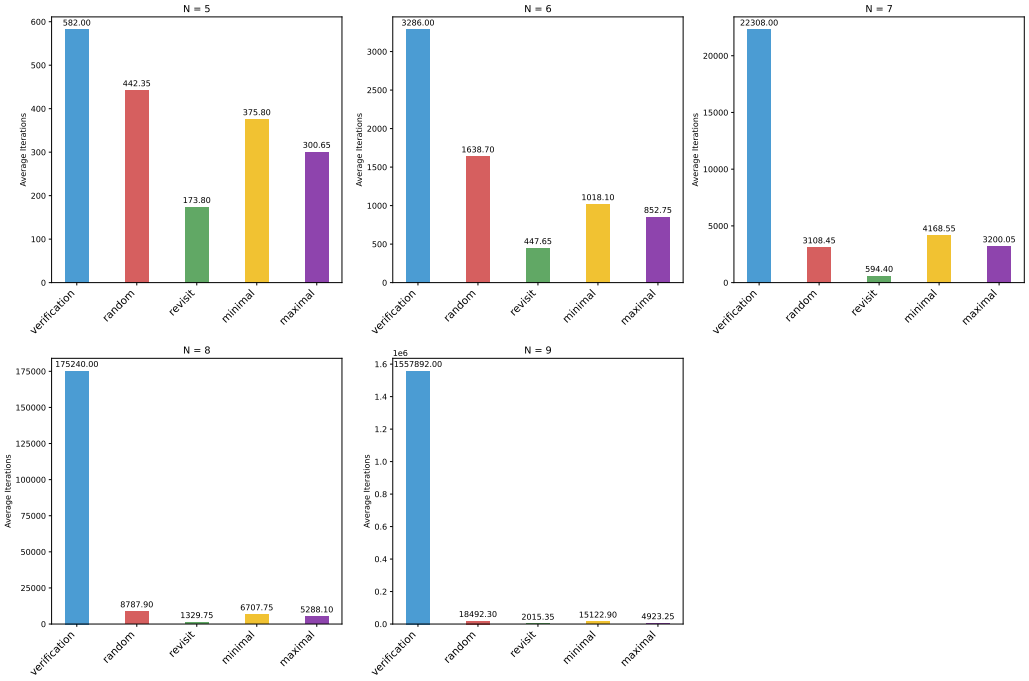

Figure 5.10: Average iterations to detect the bug

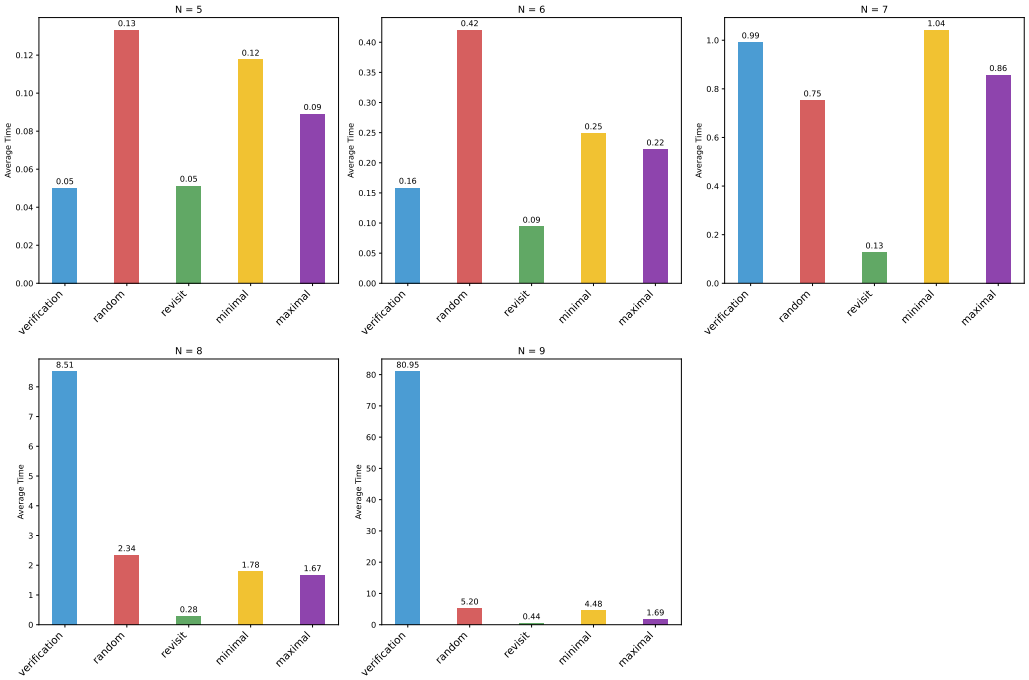

Figure 5.11: Average time to detect the bug

dom tester. The model checker is not the most time consuming strategy this time but still the revisit cut can take less time to find the bug.

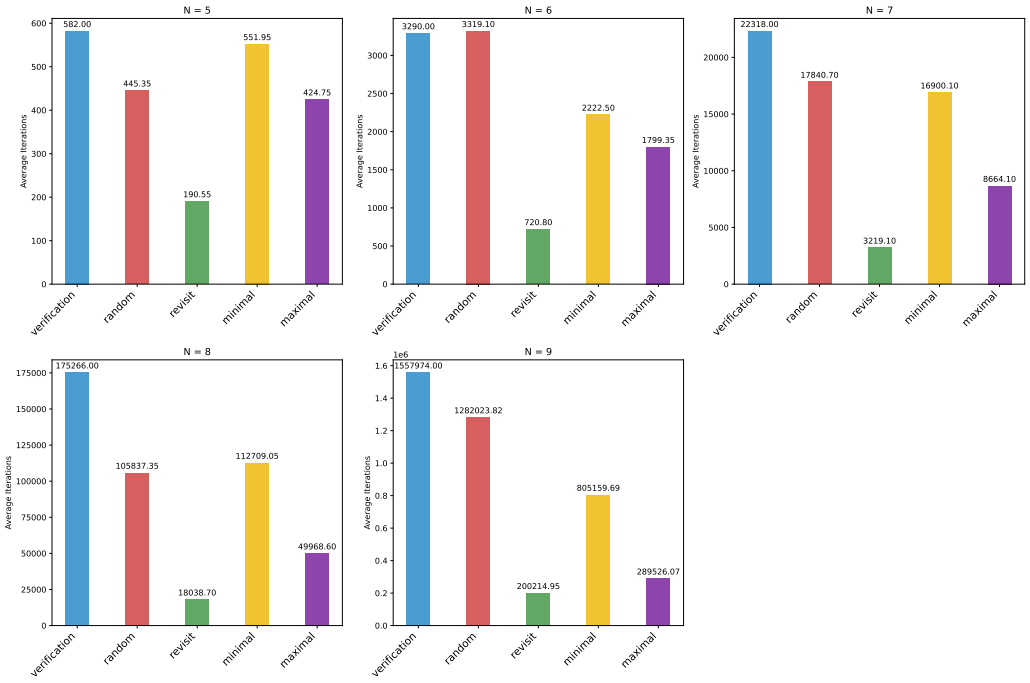

Figure 5.12: Average iterations to detect the bug (varying assertion)

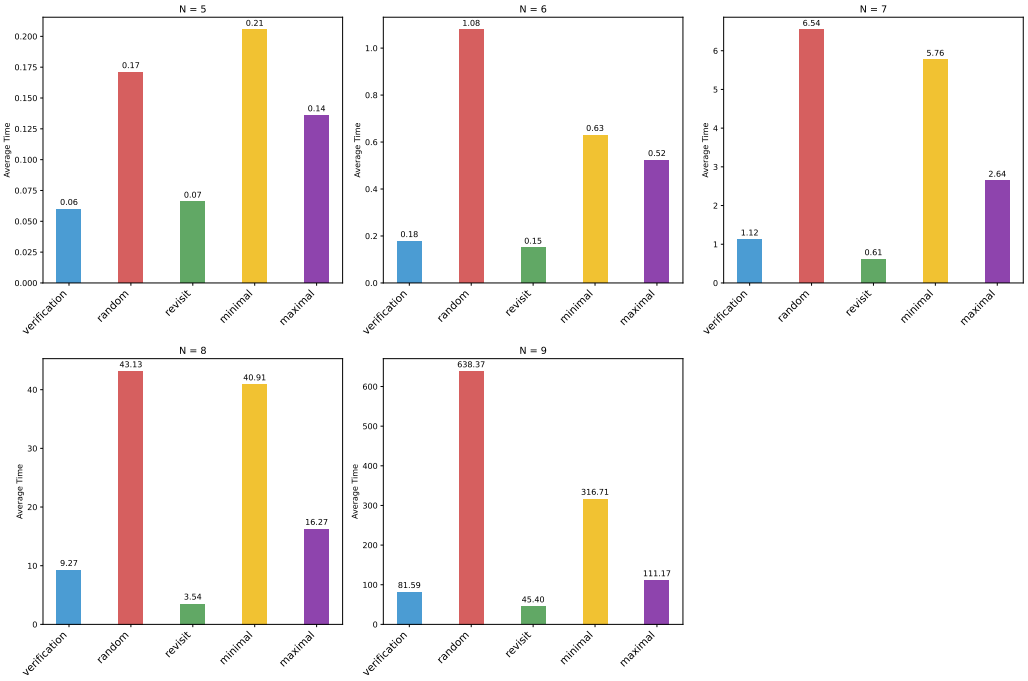

Figure 5.13: Average time to detect the bug (varying assertion)

Chapter 6

Related Work

Related work falls into three categories: memory models, fuzzing and model checking.

6.1 Memory models

In 1979 Lamport[41] proposed a memory model that models the behaviors of concurrent programs, called sequentially consistent (SC) model. In SC model, multiple tasks or processes can be deemed as if executing in a sequence, one after another. The SC model is usually called the strong memory model, with other more relaxed models called weak memory models. The total store order (TSO) model[70] enforces the total order restrictions on all write events, relaxing the the read event orders. The C/C++11[8] is a language-level memory model that specifies the semantics of the threading library constructs and atomic operations. The RC11 model[40] repairs the flawness of the semantics of SC atomic operations in the original C11 memory model which has too strong restrictions that forbids the Power compilation and too weak SC fences. [32] introduces a denotational semantics model that provides a replacement of execution graphs and axiomatic approaches. It defines the denotations for programs based on pomsets of memory actions, which are then used to define the semantics of data races. [75] presents a operational model that performs logical deduction on programs under RC11 memory model following the Owicki-Gries method. [62] proposes a operational model that defines the thread states using event structures, incorporating a storage subsystem that communicates and synchronizes with the threads. The promising semantics model (PS)[31] introduces the concepts of promises and timestamps. The write events are modeled as promises made by threads that should be fullfilled in the future. The threads maintain timestamps for memory locations, restricting their read events to read from writes with larger timestamps. Promise 2.0 (PS 2.0)[43] redesigns and refines the PS 1.0 semantics to prevent the OTOA problem by adding a stronger restricton on certifying promises.

6.2 Fuzzing

AFL[78] is a coverag guided grey-box fuzzer. The user program is intrumented with deputy functions at the entry of basicblocks evenly. The deputy functions can collect runtime information for code coverage analysis. AFL mutates the input for the program to cover new branches and execution paths. Since AFL is initially designed for sequential programs, which has deterministic results when the input and random seed are fixed, it is unaware of thread interleaving contexts. AFLFast[49] is an an extention of AFL, improving the efficiency and speed using power schedules. It incorporates search strategies that orders the seeds and controls their powers, i.e. the number of inputs generated by seeds, to reveal the low frequency paths. FairFuzz[44] is a coverage guided fuzzer based on AFL that improves the uniformity of covered code paths. It places fairness as the core criteria when generating seeds, preventing code paths being biasedly explored. MOPT[47] proposes a automated parameter optimization approach to improve mutation based fuzzers. It implements a dynamic mutation scheduling strategy to optimize the parameter settings, such as probability distribution, for selecting mutation operators. Angora[17] is a coverage guided fuzzer that aims to increase the branch coverage. It improves mutation based fuzzing, which is more difficult to generate high quality seeds than symbolic execution, by dynamically tracking program execution and counting branches to solve constraints for different paths, guiding the generation of seeds in later executions. SlowFuzz[61] is aimed at algorithmic complexity vulnerabilities, which are triggered when user input causes the program to exhibit worst-case algorithmic behaviors. It searches worst-case inputs using the binary Muzz[16] thread-aware grey-box fuzzing tool for multithreaded programs. Instead of instrument the user program evenly, Muzz conducts static analysis first to locate set of specific program statements on which thread-interleavings may only happen, identified as suspicious interleaving scopes. Then Muzz will instrument more deputy instructions inside these scopes. During repeated execution, the inserted functions can collect thread context information, hence the fuzzer is able to select interesting seeds that triggered unique thread interleavings. Conzzer[29] is a context-sensitive and directional concurrency fuzzing tool for data-race detection. It is motivated by the fact that some concurrent bugs only happen when two specific functions with callstacks are executed concurrently. Conzzer uses context sensitive concurrent function call pairs as the mutation target. Krace[76] fuzzes against data races in kernel file systems. It sets the coverage metric at the concurrency dimension, generating and mutating syscall sequences to detect possible data races, using a race detector based on lockset and happens-before relations. Razzer[28] applies the static analysis technique to guide the fuzzer to reach potential racy states. It uses controlled scheduling generate deterministic executions, by enforcing certain thread interleavings. RFF[54] uses the reads-from pairs as seeds to explore thread schedules for bug detection. It analyze program executions using abstract schedules to identify interesting seeds.

6.3 Model Checking

As the complexity of software systems grows, the demand for checking their correctness and robustness also increases. In 1986, Clarke and Emerson [19] proposed a automated ver-

ification approach for finite state concurrent systems satisfying some certain temporal logic specifications. Due to its efficiency over manually proving, model checking has been widely applied in various domains, including computer architecture, control systems, concurrent software and communication protocols. SPIN[27] is a model checker that formally verifies concurrent and distributed systems. It uses a domain specific language, called Promela, to describe the models, with the properties checked written in assertions or linear temporal logic formulas. SMV[52] is a symbolic model checking tool for verifying finite state space systems. It uses the specialized modeling language, named SMV, to model the system behaviors. It uses boolean formulas instead of state graphs to verify the programs with the help of binary decision diagram manipulations. NuSMV[18] is a reimplementation and enhanced version of SMV. It inherits most of the basic functionalities of SMV with a number of extentions. It is developed for improving the scalability and flexibility of SMV. PRISM[38] is a problisitic model checker supporting Markov chains and Markov Decision Processes, modeling the undeterministic system behaviors. The tool provides both symbolic model checking based on binary decision diagrams and sparse matrices based model checking. It can be used for checking a wide range of applications, such as internet protocols, embedded systems and biological systems. CDSchecker[59] is an exhaustive model checker exploring the behaviors of concurrent code under the original C/C++11 memory model based on stateless model-checking. A technique of constraint-based treatment of modification order is introduced remove redundancy from the search space. A cycle in the modification order graph corresponds to infeasibility of the constraints. CDSchecker uses depth-first search to check for cycles by adding edges to the modification order graph and rolls back when the constraints are unsatisfiable. The scalability limits and memory model support are extended by the C11Tester. CBMC[20] is a bounded model checker for C and C++ programs. It verifies memory safety problems, undefined behaviors and assertion failures. It bounds the searching space of programs by bounding the number of iteration for loops, limiting the length of execution paths and restricting the variable values, etc. GenMC [34] is a stateless model checker for C/C++ programs under weak memory models. It suports a variety of memory models such as RC11, IMM, and LKMM, and is extendable for further axiomatic memory models. The program is compiled into LLVM-IR for incorporating optimizations including symmetry reduction and lock-aware partial order reduction, making it possible to support different programming languages provided that its memory model is axiomatic well defined. Civl[37] is a formal verification tool for concurrent programs, supporting C, C++ and Java. It staticly analyzes the program and perform a sequence of transformations that simplifies the program based on its internal invariants. PAT[77] is a framework used for automated system analysis based on process algebra CSP. It can be used for verifying systems such as concurrent systems, real-time systems, network service models and probabilistic systems. It verifies system properties including deadlock freedom, liveness, LTL properties and can perform probabilistic model checking.

Chapter 7

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

7.1 Summary

This project contributes a novel method of fuzzing C/C++ concurrent programs under weak memory models. We first reviewed the existing fuzzing techniques and categorized them according to their usage scenarios. As summarized before, fuzzing has been widely applied in sequential programs and sequentially consistent multi-threaded programs. Then we pointed out the lack of research in fuzzing for weak memory concurrency. We porposed a new fuzzing framework that utilizes execution graph semantics, as opposed to those using thread interleaving semantics. We set the metric to be the number of distinct execution graphs explored in a fixed number of iterations when comparing our fuzzer with naive random testers. We also proposed several different mutation strategies on a given execution graph. we implemented our new fuzzer on two state-of-the-art software testing platforms: C11Tester and GenMC. In both impelementations the fuzzing approach can explore a wider range of execution graphs compared to randomized testing. It is also shown that fuzzing is better at detecting rare executions while random testing tends to fall into ferquent executions more biasedly. In the implementation on GenMC, we compared the three mutation strategies according to their performance and efficiency. Our work has demonstrated that the feasibility of fuzzing for weak concurrency and proved its superiority over randomized testing.

7.2 Discussion/Reflection

The structure of our fuzzing framework differs slightly from some existing fuzzers, in the mutation function. For example, AFL uses program inputs as seeds and mutates existing interesting seeds to generate new ones. Similarly, RFF uses abstract schedules as seeds, mutating old schedules to create new ones. The mutation functions in such fuzzers can

be expressed as $f_{mutate} : S \to S$, where S represents the set of seeds. In our framework, however, since seeds are defined as execution graph prefixes, they are mutated directly from execution graphs, which are also treated as the programs' outputs. Hence the mutation function is of the form: $f_{mutate} : G \to S$, where G denotes the set of execution graphs. In some sense, our fuzzer "mutates the program outputs to generate inputs", much resembling a snake biting its own tail. This does not violate the principle of using fuzzing to guide new testing procedures with historical information, which is expected to outperform naive random testing that lacks any feedback and black-box fuzzing, as [25] suggests. Therefore, one corresponding question to ask is: should we keep track of graphs or prefixes, when they are connected in some way? Our design choice is to keep a list of prefixes, and evaluate their scores according to some metric, just like the way AFL maintains its seeds and performs power scheduling. However, it could be an alternative way to keep track of the execution graphs.

Besides, another design choice we made was about the interesting metic. In our fuzzing algorithm, we consider an execution to be interesting if its graph is new or rare. As a result, the fuzzer will explore as many distinct execution graphs as it can. A question can be asked: why not define interesting graphs as buggy execution graphs (or more interesting, at least), i.e. being biased towards those executions with bugs? Since in software testing we often care more about the bugs. One could argue that: 1, if no bugs have been found, the fuzzer still wants to mutate from previous executions; 2, if the tester finds a bug, the user can fix the bug and test it again, without the need of knowing more bugs. On the other hand, one could also suggest: 1, the more, the better; 2, a software testing tool should try harder to detect bugs in principle. In our fuzzer, we took the unbiased approach. Even it does not detect any bug in the end, it can still provide more confidence on the correctness of the user program due to its higher coverage of possible executions.

Thirdly, it is worth discussing whether to use model checking or fuzzing when testing programs. Model checkers tend to utilize systematic algorithms to explore the search space. GenMC, for example, implements an optimal algorithm that ensures each execution it explores is unique. However, model checkers often face the challenge of state space explosion, which can cause the search process to take a long time to complete. One might ask: why not run the model checker and stop it after a fixed amount of time or iterations? This approach would prevent infinite waiting times while still ensuring that each execution obtained is unique. Naively, suppose the checker uses a depth first search strategy. If the first step yields two choices and the checker selects the left branch, the right branch will only be explored after all subtrees of the left branch have been fully explored. If we stop the search midway, we would only obtain results from the left branch. On the other hand, random testing can be thought of as both unbiased and biased. It is unbiased in that it does not have a predetermined preference for each exploration, but it is biased because some executions may have a lower probability of being explored than others. Our fuzzing approach, compared to random testing, aims to be more biased in the sense that it prioritizes those infrequent executions, and more unbiased in the sense of obtaining a more evenly distributed collection of results.

Regarding the prefix selection procedure, we currently maintain performance scores and perform a random selection weighted by these scores. This procedure can also be related

to feedback control methods. For example, denote the probabilities of generating execution graphs $[g_1, g_2, \ldots, g_n]$ starting with a prefix p_i as $[a_{1i}, a_{2i}, \ldots, a_{ni}]$. The fuzzing system can then be modeled as: $s = Au$, where $s = [N_{g_1}, N_{g_2}, \ldots, N_{g_n}]^T$ represents the number of each execution graph output, $A = \{a_{ij}\}$ is a matrix with a_{ij} denoting the probability of prefix p_j generating execution graph g_i, and $u = [N_{p_0}, N_{p_1}, \ldots, N_{p_m}]^T$ is the number of usages for each prefix, with p_0 representing an empty prefix, which is equivalent to a completely random execution. By differencing both sides of the equation, we obtain: $\Delta s = A\Delta u$. By setting the desired output $s_d = [1, 1, \ldots, 1]^T$ and performing an online estimation of A [26], one may apply linear control methods, such as LQR, to determine which input prefix should be selected in the next iteration.

Lastly, a few comments about the thread interleavings. It is known that in weak memory concurrency we use execution graphs, instead of thread interleavings used in SC concurrency, to model executions. Since the SC memory model is a stronger model than weak memory models, the executions allowed under SC should be also allowed by weak memory models, therefore should be captured by execution graph semantics. In other words, using only execution graphs is sufficient in a more general sense. However, many model checkers also have a scheduler in their implementations. This scheduler is not the scheduler used for determining thread interleavings under SC, but is used for scheduling the order of adding events to execution graphs. Note that given a prefix P and the next event e in thread t, the following two scenarios are equivalent: 1, set P as the prefix of next execution and force the scheduler to pick thread t first, e being the next event to be added; 2, add e to prefix P to compose a new prefix $P' = P \cup \{e\}$ and let the scheduler to add other events randomly. Hence, choosing which one of the two approaches is only an impelmentation issue, not an algorithmic difference. In our implementation on C11Tester, we chose the first approach, but that does not mean the fuzzer is still under the SC semantics.

7.3 Future work

In our current implementation, although the maximal cut performs best in a majority of benchmarks, still other mutations can outperform maximal cut in other cases. One continuation of this work could be develop a compound strategy that dynamically adjusts its preference on mutation methods. Besides the proposed mutation strategies, one could also develop other language or model specific mutations. The current implementation mutates the reads-from relation, including read-modify-writes. There could be other mutations targeting on relations such as modification-orders or primitives such as atomic fences, cas loops and atomic pointers. Besides, the fuzzing procedure relies on feedback from previous executions. As discussed before, the fuzzing approach can improve the random testing, but it would be also possibel to derive some theoritical analysis on estimating how far it improves from the baseline on a given benchmark. The current fuzzer considers execution graphs to be interesting when they are unseen or rare. An alternative approach could be to record certain relations, such as the reads-from relation, and consider a graph interesting when it contains certain interesting relations.

Bibliography

[1] Docker: Accelerated container application development. URL https://www.docker.com/.

[2] Extending genmc's usability and performance (replication package). URL https://zenodo.org/records/10018136.

[3] Vagrant community. URL https://www.vagrantup.com/.

[4] Oracle vm virtualbox. URL https://www.virtualbox.org/.

[5] External technical root cause analysis — channel file 291, 2024. URL https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf.

[6] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 512–532, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37036-6.

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), jul 2014. ISSN 0164-0925. doi: 10.1145/2627752. URL https://doi.org/10.1145/2627752.

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, jan 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926394. URL https://doi.org/10.1145/1925844.1926394.

[9] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276514. URL https://doi.org/10.1145/3276514.

[10] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, jun 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375591. URL https://doi.org/10.1145/1379022.1375591.

[11] Frederick P. Brooks. *The mythical man-month – Essays on Software-Engineering*. Addison-Wesley, 1975.

[12] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 459–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-20398-5.

[13] cbloom. Mcs list-based lock, . URL https://cbloomrants.blogspot.com/2011/07/07-18-11-mcs-list-based-lock_18.html.

[14] cbloom. A look at some bounded queues - part 2, . URL https://cbloomrants.blogspot.com/2011/07/07-30-11-look-at-some-bounded-queues.html.

[15] Thomas J. Watson IBM Research Center and R.K. Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986. URL https://books.google.nl/books?id=YQg3HAAACAAJ.

[16] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu.

[17] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *CoRR*, abs/1803.01307, 2018. URL http://arxiv.org/abs/1803.01307.

[18] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, apr 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. URL https://doi.org/10.1145/5397.5399.

[20] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

[21] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, mar 1997. ISSN 0163-5948. doi: 10.1145/251880.251992. URL https://doi.org/10.1145/251880.251992.

[22] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the armv8 architecture, operationally: concurrency and isa. *SIGPLAN Not.*, 51(1):608–621, jan 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837615. URL https://doi.org/10.1145/2914770.2837615.

[23] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018. doi: 10.1109/SP.2018.00040.

[24] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. Probabilistic concurrency testing for weak memory programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 603–616, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575729. URL https://doi.org/10.1145/3575693.3575729.

[25] Patrice Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, page 1, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938817. doi: 10.1145/1292414.1292416. URL https://doi.org/10.1145/1292414.1292416.

[26] G.C. Goodwin and K.S. Sin. *Adaptive Filtering Prediction and Control*. Dover Books on Electrical Engineering. Dover Publications, 2009. ISBN 9780486469324. URL https://books.google.nl/books?id=IIRCAwAAQBAJ.

[27] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. doi: 10.1109/32.588521.

[28] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019. doi: 10.1109/SP.2019.00017.

[29] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. 01 2022. doi: 10.14722/ndss.2022.24296.

[30] Phillip Johnston and R.L. Harris. The boeing 737 max saga: Lessons for software organizations. *Software Quality Professional Magazine*, 21, 2019. URL https://api.semanticscholar.org/CorpusID:195414546.

[31] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *SIGPLAN Not.*, 52(1):175–189, jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009850. URL https://doi.org/10.1145/3093333.3009850.

[32] Ryan Kavanagh and Stephen D. Brookes. A denotational account of c11-style memory. *ArXiv*, abs/1804.04214, 2018. URL https://api.semanticscholar.org/CorpusID:4839024.

[33] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

[34] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 427–440, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81685-8.

[35] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. Kater: Automating weak memory model metatheory and consistency checking. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571212. URL https://doi.org/10.1145/3571212.

[36] Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis. Enhancing genmc's usability and performance. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 66–84, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-57249-4.

[37] Bernhard Kragl and Shaz Qadeer. The civl verifier. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 143–152, 2021. doi: 10.34727/2021/isbn.978-3-85448-046-4_23.

[38] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In Tony Field, Peter G. Harrison, Jeremy Bradley, and Uli Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 200–204, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-46029-9.

[39] Intel Labs. Hw-assisted feedback fuzzer for x86 vms. URL https://github.com/IntelLabs/kAFL.

[40] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. *SIGPLAN Not.*, 52(6):618–632, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062352. URL https://doi.org/10.1145/3140587.3062352.

[41] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439.

[42] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *SIGPLAN Not.*, 48(8):69–80, feb 2013. ISSN 0362-1340. doi: 10.1145/2517327.2442524. URL https://doi.org/10.1145/2517327.2442524.

[43] Sung Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In Alastair F. Donaldson and Emina Torlak, editors, *PLDI 2020 - Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 362–376. Association for Computing Machinery, June 2020. doi: 10.1145/3385412.3386010. Publisher Copyright: © 2020 ACM.; 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020 ; Conference date: 15-06-2020 Through 20-06-2020.

[44] Caroline Lemieux and Koushik Sen. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238176. URL https://doi.org/10.1145/3238147.3238176.

[45] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, jul 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940. URL https://doi.org/10.1109/MC.1993.274940.

[46] Weiyu Luo and Brian Demsky. C11tester: a race detector for c/c++ atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 630–646, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446711. URL https://doi.org/10.1145/3445814.3446711.

[47] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[48] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, jan 2005. ISSN 0362-1340. doi: 10.1145/1047659.1040336. URL https://doi.org/10.1145/1047659.1040336.

[49] Böhme Marcel and Zalewski Michal. Aflfast, 2016. URL https://github.com/mboehme/aflfast.

[50] Arash Massoudi. Knight capital glitch loss hits $461m, 2012. URL https://www.ft.com/content/928a1528-1859-11e2-80e9-00144feabdc0.

[51] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, USA, 2004. ISBN 0735619670.

[52] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, USA, 1992. UMI Order No. GAX92-24209.

[53] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. *SIGARCH Comput. Archit. News*, 19(2):269–278, apr 1991. ISSN 0163-5964. doi: 10.1145/106975.106999. URL https://doi.org/10.1145/106975.106999.

[54] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 1615–1629, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3623097. URL https://doi.org/10.1145/3576915.3623097.

[55] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[56] Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis. Model checking for a multi-execution memory model. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. doi: 10.1145/3563315. URL https://doi.org/10.1145/3563315.

[57] Nccgroup. Afl/qemu fuzzing with full-system emulation. URL https://github.com/nccgroup/TriforceAFL.

[58] Djordje Nedic. lockfree. URL https://github.com/DNedic/lockfree.

[59] Brian Norris and Brian Demsky. Cdschecker: checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131–150, oct 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509514. URL https://doi.org/10.1145/2544173.2509514.

[60] McKenney Paul, Weigand Ulrich, Parri Andrea, Feng Boqun, and Stern Alan. Linux-kernel memory model, 2020. URL https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0124r7.html.

[61] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2155–2168, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134073. URL https://doi.org/10.1145/3133956.3134073.

[62] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 622–633, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837616. URL https://doi.org/10.1145/2837614.2837616.

[63] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290382. URL https://doi.org/10.1145/3290382.

[64] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158107. URL https://doi.org/10.1145/3158107.

[65] Manuel Pöter. xenium. URL https://github.com/mpoeter/xenium.

[66] Pedro Ramalhete and Andreia Correia. Brief Announcement: Left-Right - A Concurrency Control Technique with Wait-Free Population Oblivious Reads. In Yoram Moses and Matthieu Roy, editors, *DISC 2015*, volume LNCS 9363 of *29th International Symposium on Distributed Computing*, Tokyo, Japan, October 2015. Toshimitsu Masuzawa and Koichi Wada, Springer-Verlag Berlin Heidelberg. URL https://hal.science/hal-01207881.

[67] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 175–186, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993520. URL https://doi.org/10.1145/1993498.1993520.

[68] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587936. doi: 10.1145/1791194.1791203. URL https://doi.org/10.1145/1791194.1791203.

[69] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[70] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors.

Commun. ACM, 53(7):89–97, 2010. doi: 10.1145/1785414.1785443. URL https://doi.org/10.1145/1785414.1785443.

[71] Shellphish. Driller: augmenting afl with symbolic execution! URL https://github.com/shellphish/driller.

[72] CORPORATE SPARC International, Inc. *The SPARC architecture manual: version 8*. Prentice-Hall, Inc., USA, 1992. ISBN 0138250014.

[73] Colin Wheildon. *Type & Layout*. Strathmore Press, 1995. (ISBN 0 9624891 5 8).

[74] Anthony Williams. Implementing dekker's algorithm with fences. URL https://www.justsoftwaresolutions.co.uk/threading/implementing_dekkers_algorithm_with_fences.html.

[75] Daniel Wright, Mark Batty, and Brijesh Dongol. Owicki-gries reasoning for c11 programs with relaxed dependencies. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan, editors, *Formal Methods*, pages 237–254, Cham, 2021. Springer International Publishing. ISBN 978-3-030-90870-6.

[76] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020. doi: 10.1109/SP40000.2020.00078.

[77] LIU YANG. Model checking concurrent and real-time systems : the pat approach. URL https://scholarbank.nus.edu.sg/handle/10635/17326.

[78] Michal Zalewski. American fuzzy lop, 2014. URL https://lcamtuf.coredump.cx/afl/.

Appendix A

Source code

In this appendix we provide the source code repositories used in this thesis and instructions on how to use them.

A.1 Fuzzer implementation in C11Tester

The source code is available at GitHub: https://github.com/lililuanluan/c11tester-fuzz.git.

- Download and install the vagrantBox[3] and VirtualBox[4].
- Download the C11Tester artifact following the instructions of the C11Tester[46] paper.
- Download the fuzzer implementation from the GitHub link: https://github.com/lililuanluan/c11tester-fuzz.git and place the files in the `c11tester/` folder. Go to the `c11tester/benchmarks/` folder and run the `afuzzer.py` script.

A.2 Fuzzer implementation in GenMC

The source code is available at GitHub: https://github.com/lililuanluan/genmc-fuzz.git.

- Download and install Docker[1].
- Download the GenMC artifact from paper [36] following the instructions in its artifact[2].
- Download the `src/` and `debug-luan` folders and place them in the `/genmc-tool/` directory.
- Go to `debug-luan/` and run the `test.sh` script.

Appendix B

Requirements and Guidelines

This chapter details some requirements and guidelines for MSc theses submitted to the Software Engineering Research Group.

B.1 Requirements

B.1.1 Layout

- Your thesis should contain the formal title pages included in this document (the page with the TU Delft logo and the one that contains the abstract, student id and thesis committee). Usually there is also a cover page containing the thesis title and the author (this document has one) but this can be omitted if desired.
- Base font should be an 11 point serif font (such as Times, New Century Schoolbook or Computer Modern). Do not use sans-serif fonts such as Arial or Helvetica. *Sans-serif type is intrinsically less legible than seriffed type* [73].
- The final thesis and drafts submitted for reviewing should be printed double-sided on A4 paper.

B.1.2 Content

- The thesis should contain the following chapters:
  - Introduction.
    Describes project context, goals and your research question(s). In addition it contains an overview of how (the remainder of) your thesis is structured.
  - One or (usually) more "main" chapters.
    Present your work, the experiments conducted, tool(s) developed, case study performed, etc.
  - Overview of Related Work
    Discusses scientific literature related to your work and describes how those approaches differ from what you did.

  - Discussion/Evaluation/Reflection

    What went well, what went less well, what can be improved?
  - Conclusions, Contributions, and (Recommendations for) Future Work
  - Bibliography

B.1.3 Bibliography

- Make sure you've included all required data such as journal, conference, publisher, editor and page-numbers. When you're using BIBTEX, this means that it won't complain when running `bibtex your-main-tex-file`.
- Make sure you use proper bibliographic references. This especially means that you should avoid references that **only** point at a website and not at a printed publication.

  For example, it's OK to add a URL with the entry for an article describing a tool to point at its homepage, but it's not OK to just use the URL and not mention the article.

B.2 Guidelines

- The main chapters of a typical thesis contain approximately 50 pages.
- A typical thesis contains approximately 50 bibliographic references.
- Make sure your thesis structure is balanced (check this in the table of contents).

  Typically the main chapters should be of equal length. If they aren't, you might want to revise your structure by merging or splitting some chapters/sections.

  In addition, the (sub)section hierarchies with the chapters should typically be balanced and of similar depth. If one or more are much deeper nested than others in the same chapter this generally signals structuring problems.
- Whenever you submit a draft of your thesis to your supervisor for reviewing, make sure that you have checked the spelling and grammar. Moreover, *read it yourself at least once from start to end, before submitting to your supervisor.*

  Your supervisor is not a spelling/grammar checker!
- Whenever you submit a second draft, include a short text which describes the changes w.r.t. the previous version.