The Journal of Systems & Software 191 (2022) 111379

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Fuzzing with automatically controlled interleavings to detect concurrency bugs☆

Youngjoo Ko [a], Bin Zhu [b], Jong Kim [a,*]

[a] Pohang University of Science and Technology, Pohang, South Korea
[b] Microsoft Research in Asia, Beijing, China

ARTICLE INFO

Article history:
Received 27 April 2021
Received in revised form 1 March 2022
Accepted 19 May 2022
Available online 28 May 2022

Keywords:
Fuzzing
Bug detection
Concurrency vulnerabilities
Multi-threading
Reliability

ABSTRACT

Concurrency vulnerabilities are an irresistible threat to security, and detecting them is challenging. Triggering the concurrency vulnerabilities requires a specific thread interleaving and a bug-inducing input. Existing methods have focused on one of these but not both or have experimented with small programs, which raises scalability issues.

This paper introduces AutoInter-fuzzing, a fuzzer controlling thread interleavings elaborately and providing an interleaving-aware power schedule to detect vulnerabilities in a multi-threaded program. AutoInter-fuzzing consists of static analysis and dynamic fuzzing. At the static analysis, the fuzzer extracts and optimizes the interleaving space to be explored and adds instrumentation to control thread interleavings. We apply the power schedule in the dynamic fuzzing to focus on the seeds that reveal the new interleaving space. The fuzzer records the interleaving information in a log when a crash occurs and uses it to reproduce and validate the crash.

Experiments with 13 real-world multi-threaded programs show that the interleaving-aware power schedule effectively enlarges the untested interleaving space, and AutoInter-fuzzing outperforms AFL and ConAFL in detecting interleaving-relevant vulnerabilities. AutoInter-fuzzing has detected six interleaving-relevant vulnerabilities, including two new vulnerabilities and four interleaving-irrelevant vulnerabilities.

1. Introduction

Multi-threaded programs have been popular in modern systems because most CPUs have multiple cores that can execute threads concurrently, significantly improving software performance. Many programs have been implemented with multiple threads to boost their performance by distributing their work to the different threads. As programs become increasingly complex, bug detection also becomes difficult. Detecting bugs in multi-threaded programs are much more complex than in sequential programs because a multi-thread program introduces non-deterministic thread interleaving. Non-deterministic interleaving makes it hard to repeat the abnormal outcomes or crashes. Concurrency vulnerability can cause critical security issues such as privilege escalation (CVE, 2018a,b,c). Therefore, detecting concurrency vulnerabilities in complex multi-threaded programs is critically important.

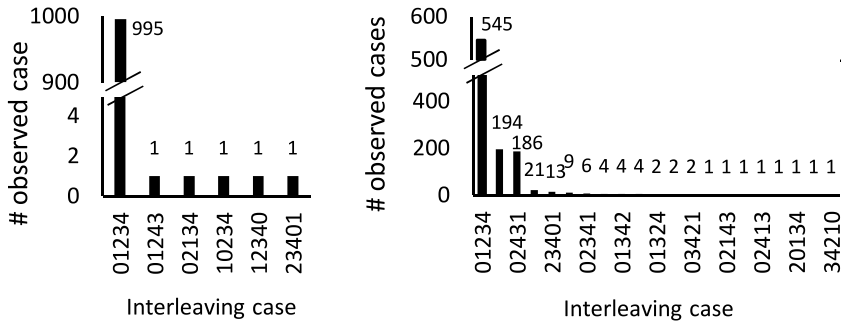
To find concurrency bugs, we have to find problematic operations to be interleaved, defined as interleaving space. Also, each concurrency bug requires a specific interleaving order and inputs to trigger it because the bug is hidden in sophisticated program flows. For an example of use-after-free, we need a thread interleaving at de-allocation and use operations (interleaving space), i.e., de-allocating the memory before using it (specific order), and an input creating multiple threads that execute de-allocation and use operations. The challenge is finding the interleaving space and fulfilling both requirements simultaneously. Since a thread interleaving in multi-threaded programs is non-deterministic at run-time, it is difficult to have the thread interleaving in a specific order naturally. In addition, the input needs to execute operations on multiple threads.

Fig. 1 shows the observed executing orders of five threads by creating them 1000 times in the same order of creation from zero to four with/without running a resource contenting program. Most executions have the same thread interleaving without contention, but various thread interleaving cases have been observed with contention. Different thread interleaving patterns are shown while programs run, and certain interleavings rarely occur. If a bug is triggered with a thread interleaving that rarely occurs, it is

☆ Editor: Earl Barr.

* Correspondence to: Pohang University of Science and Technology, PIAI 441. 77 Cheongam-Ro, Nam-Gu, Pohang, Gyeongbuk, 37673, South Korea.
E-mail addresses: y0108009@postech.ac.kr (Y. Ko), binzhu@microsoft.com (B. Zhu), jkim@postech.ac.kr (J. Kim).

https://doi.org/10.1016/j.jss.2022.111379
0164-1212/



(a) without contention (b) with contention

Fig. 1. The number of observed execution orders of five threads. We create them 1000 times in the same order of creation from 0 to 4 (a) without contention (b) with contention.

tough to expose this hidden bug. Moreover, since existent multi-threaded programs are more complex than this simple case, it is pretty challenging to control the thread interleaving that triggers vulnerabilities in the real world.

Studies using both static and dynamic methods have been reported detecting concurrency vulnerabilities. Static methods (Voung et al., 2007; Engler and Ashcraft, 2003; Blackshear et al., 2018; Pratikakis et al., 2006; Sui et al., 2016) analyze context-sensitive correlations and detect potential bugs in multi-threaded programs without actual execution. Static methods generally have a high percentage of false alarms (Hong and Kim, 2015) and require additional steps to validate bugs detected. Dynamic methods (Park et al., 2010; Zhang et al., 2010) monitor the execution of programs to detect concurrency errors based on program behaviors such as error patterns. In addition, studies (Burckhardt et al., 2010; Musuvathi et al., 2004; Godefroid, 1997; Yu et al., 2012) have leveraged interleaving characteristics and generated tests to cover as many interleavings as possible. They usually focus on reproducing bugs using a given input. Thus, they are unsuitable for testing multi-threaded programs with concurrency vulnerabilities, which requires a specific execution order and inputs to activate the buggy path.

Among methods generating bug-inducing inputs, grey-box fuzzing (GBF) is a representative technique for detecting program bugs. GBF generates efficient inputs to cover as many paths as possible using coverage metrics because exploring many paths can increase the possibility of finding bugs. GBF has received significant attention from researchers and the industry and has successfully detected thousands of vulnerabilities in real-world programs with a lightweight coverage feedback instrumentation. However, existing GBF tools (Rawat et al., 2017; Zalewski, 2020; Peng et al., 2018) cannot effectively detect concurrency vulnerabilities in multi-threaded programs because they do not proactively explore thread interleavings.

Recently proposed GBF tools (Liu et al., 2018; Chen et al., 2020) aim to trigger the concurrency vulnerabilities of multi-threaded programs. ConAFL (Liu et al., 2018) applies a static analysis to locate potential bugs and vulnerabilities, their triggering interleaving, and fuzzing inputs to trigger an interleaving. However, it is unscalable to large-sized programs because it requires a heavy static analysis. MUZZ (Chen et al., 2020) selects multi-threading-relevant seeds by distinguishing execution states in multi-threading contexts. However, it does not control a specific thread interleaving. We need a fuzzing tool that considers both a specific thread interleaving and a bug-inducing input.

In this paper, we propose *AutoInter-fuzzing* to detect interleaving-relevant vulnerabilities in multi-threaded programs. We refer to *interleaving-relevant vulnerabilities* to indicate vulnerabilities caused by a specific interleaving. AutoInter-fuzzing consists of lightweight static analysis and dynamic fuzzing. The lightweight static analysis extracts the interleaving space — vulnerable operation pairs, where two operations of a pair access the same memory region and are executed in different threads concurrently. Since we cannot test all thread interleavings operation by operation, we reduce the thread interleaving space to be controlled by applying optimization to remove false positive pairs that cannot run concurrently through thread-aware analysis. We instrument the target program for the dynamic fuzzing step.

The dynamic fuzzing generates test cases with the inputs to the target program. We develop an interleaving-aware power schedule that generates test cases discovering untested interleaving space to increase the chance of having bug-inducing inputs. While fuzzing, the fuzzer checks the execution trace from the instrumentation in the target program. If there are pairs needing thread interleavings, the fuzzer controls the target program to have a specific execution order to run untested thread interleaving. The dynamic fuzzing utilizes context-aware IDs and scheduling code instrumentations to control the execution order of pairs. When a crash occurs, our fuzzing records a log (the input used and a specific execution order) to validate the crash at a later time.

In evaluation, we show that the static analysis can extract the reasonable interleaving space by removing pairs that cannot run concurrently. We demonstrate that AutoInter-fuzzing outperforms AFL and ConAFL in detecting interleaving vulnerabilities. We confirm that the power schedule effectively generates test cases that reveal untested interleavings. We demonstrate that logged crash information (i.e., inputs and interleaving orders) can help identify detected bugs.

This paper includes the following significant contributions:

- We design an end-to-end testing tool, AutoInter-fuzzing, to detect interleaving-relevant vulnerabilities in multi-threaded programs. AutoInter-fuzzing automatically creates test cases and controls the thread interleaving without user interaction.
- AutoInter-fuzzing determines the interleaving space to be controlled through the lightweight static analysis. The optimization phase in the static analysis can reduce the false-positive pairs by 49.6% on average.
- The interleaving-aware power schedule in AutoInter-fuzzing generates test cases that expose untested interleaving space. It can help generate bug-inducing inputs that can test the buggy thread interleaving.
- We evaluated AutoInter-fuzzing on 13 real-world multi-threaded programs with various sizes from 1.1K to 11.5M lines of code (LoC). AutoInter-fuzzing detected six interleaving-relevant vulnerabilities, including two new vulnerabilities. In addition, AutoInter-fuzzing found four interleaving-irrelevant vulnerabilities.

Code (1) T1

```
1  int *global
2  void Thread_1()
3  {
4      ...
5      global=NULL;
6
7  }
```

Code (2) T2

```
1
2  void Thread_2(int i)
3  {
4      ...
5      if (i==0)
6          *global++;
7  }
```

Fig. 2. An example of NULL-pointer dereference.

This paper is organized as follows. We briefly describe the background of bugs in multi-threaded programs and grey-box fuzzing test in Section 2. An overview of the design is provided in Section 3. Section 4 presents the static analysis of AutoInter-fuzzing and Section 5 describes the dynamic fuzzing with elaborated thread interleaving. The implementation of Auto Inter-fuzzing is presented in Section 6. The evaluation results are provided in Section 7. Discussion and the related work are presented in Sections 8 and 9, respectively. The paper concludes with Section 10.

2. Background

2.1. Bugs in multi-threaded programs

Bugs and vulnerabilities make programs behave abnormally and may cause security issues. Multi-threaded programs are more complex than single-threaded programs, so they have hard-to-find bugs. The interaction between threads can trigger hidden bugs, called concurrency bugs.

Concurrency bugs usually occur when multiple threads run concurrently and affect other threads. There are many concurrency bugs, such as accessing invalid memory regions, data races, atomic violations, and deadlocks. Bugs related to accessing invalid memory regions include NULL-pointer dereference, use-after-free, and double-free. A data race and atomic violation occur when lock and unlock are incorrectly implemented. Moreover, a random interleaving order may trigger a deadlock, causing the program to be stuck without providing a service. These bugs can cause incorrect outputs of programs or a crash in program execution.

Triggering a concurrency bug is difficult even when such bugs exist because it requires an interleaving space, a specific execution order, and a bug-inducing input. For example, as shown in Fig. 2, the interleaving space is line 5 of T_1 and line 6 of T_2, where the thread interleaving has to occur. When the NULL assignment operation to variable "global" (line 5 of T_1) is executed before the dereference operation (line 6 of T_2) (specific order), it triggers NULL-pointer dereference. In addition, the input is needed to pass a control to the branch (line 5 of T_2) to execute the dereference operation.

A real-world multi-threaded program typically has an extremely large interleaving space where thread interleavings can occur. This makes it difficult to test the whole thread interleaving space to expose concurrency bugs. Moreover, even when a concurrency bug occurs at run-time, it is difficult to re-create the bug because it needs to repeat the same thread interleaving in a large thread interleaving space. In addition, the input must drive a path that executes operations in different threads in such an order that makes a bug occur. In conclusion, it is not easy to satisfy the conditions to trigger and reproduce a concurrency bug, even when there is such a bug.

2.2. Grey-box fuzzing test

Grey-box fuzzing consists of several components: seed selection, power schedule, energy assignment, seed mutation, and coverage feedback. We briefly describe these components used in the fuzzing as follows.

- **Seed and seed queue.** A seed is a test case to generate other test cases with mutation. A test case is added to the seed if the test case is favored. For example, if a test case drives to execute new basic blocks in the target program, then the test case is added as a seed.
- **Testcase.** A test case is an input to the target program. It is created by mutating a seed.
- **Seed selection.** Seed selection is the process of choosing a seed for mutation. There are several approaches to selecting seeds. For example, AFL (Zalewski, 2020) selects the seed that has explored new execution paths or triggered a new crash.
- **Seed mutation.** Mutation generates test cases by modifying a seed. Examples of mutation strategies are bit-flip, arithmetic, and token.
- **Power schedule and energy assignment.** Power schedule determines how many test cases will be generated with a given seed. The number of test cases is the energy, and power schedule assigns energy to each seed. The higher the energy, the more the number of test cases.
- **Coverage feedback.** After testing the target program with input, grey-box fuzzing obtains the execution trace (i.e., coverage) to discern the quality of test cases. Various coverage metrics include path, edge, and basic block coverage.

Fuzzing (Miller et al., 1990) is a software testing method that automatically generates random or designated inputs to test a target program. Grey-box fuzzing (GBF), a kind of fuzzing, leverages lightweight instrumentation to extract the coverage information such as path and code coverage to determine the quality of test cases. GBF purposes of detecting bugs as well as to increase the coverage. GBF first selects a seed from the seed queue and decides how many test cases are to be generated from the seed based on the power schedule. GBF mutates the seed using mutation strategies (e.g., bit-flip and arithmetic) to generate test cases. After testing the program with the test case as an input, GBF obtains coverage feedback from the program. The test cases are added to the seed queue if the new coverage is discovered. GBF repeats the input generation, testing the program, and evaluating the input using coverage. If all test cases from the seed are tested, GBF selects another seed and continues the above steps.

A representative GBF tool is American-fuzz-lop (AFL) (Zalewski, 2020), which is widely used. Many GBF techniques (Böhme et al., 2017; Lyu et al., 2019; Manès et al., 2018) have also been proposed to improve the bug and vulnerability detection capability from simple to complex programs. For example, MOPT (Lyu et al., 2019) selects efficient mutation operators based

on a customized Particle Swarm Optimization to create meaningful test cases. AFLFAST (Böhme et al., 2017) concentrates on generating inputs that maximize the coverage by solving magic-byte, a specific value for entering the branches. GBF techniques have found numerous bugs in real-world programs and have proven their effectiveness in several studies (Lyu et al., 2019; Chen et al., 2020).

Although GBF techniques have improved the quality of the fuzzing by maximizing the coverage and generating crashes, they have usually handled single-threaded programs. A deep understanding of the characteristics of multi-threaded programs, such as shared memory and interleaving, has yet to be achieved. Such techniques are ineffective in discovering and reproducing detected concurrency bugs because the thread interleaving in a multi-threaded program is non-deterministic. Moreover, obtaining thread interleaving information may be difficult because a program creates many threads dynamically at run-time. These dynamically created threads are unknown before the program's execution and can change from one run to another. To reveal vulnerabilities in multi-threaded programs, GBF needs to know the thread interleaving information, control execution orders, and generate effective test cases efficiently.

3. Design

Our main goal is to discover interleaving-relevant vulnerabilities by triggering them if they exist. To achieve this goal, we utilize the fuzzing technique to generate bug-inducing inputs and automatically control thread interleavings without user intervention.

3.1. Terminologies

We define the following terminologies in advance to avoid confusion.

- **Context-aware ID.** Depending on the loop, function call stack, or multiple threads, the operation in the code can be executed multiple times. We have to distinguish these operations in a dynamically executing environment. We utilize the executed threads and the executed operations sequence as the context to distinguish these operations. We generate an ID considering the context, i.e., the operation sequence and the thread number, and assign it to the executed operation. This ID is called context-Aware ID.
- **Thread-aware pair.** Thread-aware pair is two operations that access the same memory address in different threads and can run concurrently.
- **Interleaving space.** Interleaving space represents all thread-aware pairs with their possible thread interleavings.

3.2. Design goal

We set the following design goals.

Automatic fuzzing to trigger a concurrency vulnerability. Our fuzzing aims to detect a concurrency vulnerability. We need a buggy input that reaches vulnerable operations and a specific execution order to trigger a concurrency vulnerability. Therefore, our fuzzing tool generates inputs executing operations vulnerable to concurrency vulnerabilities. Moreover, we control thread interleavings to test the vulnerable execution order without user interaction.

Effective and sophisticated control of thread interleavings. We have to define the interleaving space to trigger concurrency vulnerabilities for the effective control of thread interleaving because only some operations cause concurrency vulnerabilities

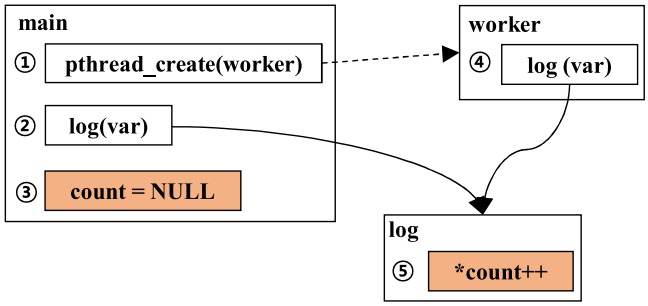


Fig. 3. The log function is called in both the main and worker threads. A solid arrow means a function call and a dashed arrow means thread creation. The orange boxes cause NULL-pointer dereference. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

such as de-allocation, store, and load operations. We also need sophisticated control of a thread interleaving between operations because it is difficult to fully control the interleaving of operations at the static analysis time due to the reuse of many lines of code. Fig. 3 shows an example of the desired interleaving in which the operation (③) of the main thread is executed before the execution of the operation (⑤) by the worker thread. Both the main thread and worker thread call the *log* function (② and ④, respectively). In this case, the fuzzing has to distinguish which thread has called the *log* function. Otherwise, the fuzzing may attempt to interleave two operations that cannot be interleaved (i.e., impossible interleaving) (between ③ and ⑤), both from the main thread. Therefore, we have to discern precisely which operations are interleaved at every run.

3.3. Overview

AutoInter-fuzzing is used to test a multi-threaded program with two steps: static analysis and dynamic fuzzing, as shown in Fig. 4. We first conduct a points-to static analysis to find the interleaving space that may trigger concurrency bugs in a given program. The static analysis aims at finding all pairs of two operations accessing the same memory address in different threads. We analyzed a target program with an inclusion-based pointer analysis (Andersen, 1994) to handle pointer operations. The points-to analysis does not handle the control flow, and the static analysis does not know which operations are executed in run-time. As a result, the pairs obtained may include false positives such that the two operations in a pair belong to different threads but never run concurrently or do not affect each other. We remove such false-positive pairs by checking thread relationships and reachability in the optimization phase. We also instrument the target program for the dynamic fuzzing step. We insert code to assign context-aware IDs to discern the threads and operations in run-time. We also instrument scheduling code to control a thread interleaving and record an execution trace to obtain created threads and executed operations.

After the static analysis and instrumentation, we run a dynamic fuzzing mechanism using the pairs obtained from the static analysis. The dynamic fuzzing has two execution modes: the normal and interleaving modes. First, the fuzzer runs the target program in normal mode with a given input. Each thread and operation is assigned with a dynamic context-aware ID by the instrumentation code during run-time. While executing the target program, the target program records a trace of the executed operation IDs, thread IDs, and the memory address that the operations access. The trace is delivered to the fuzzer as feedback. After executing the normal mode, the fuzzer decides whether to

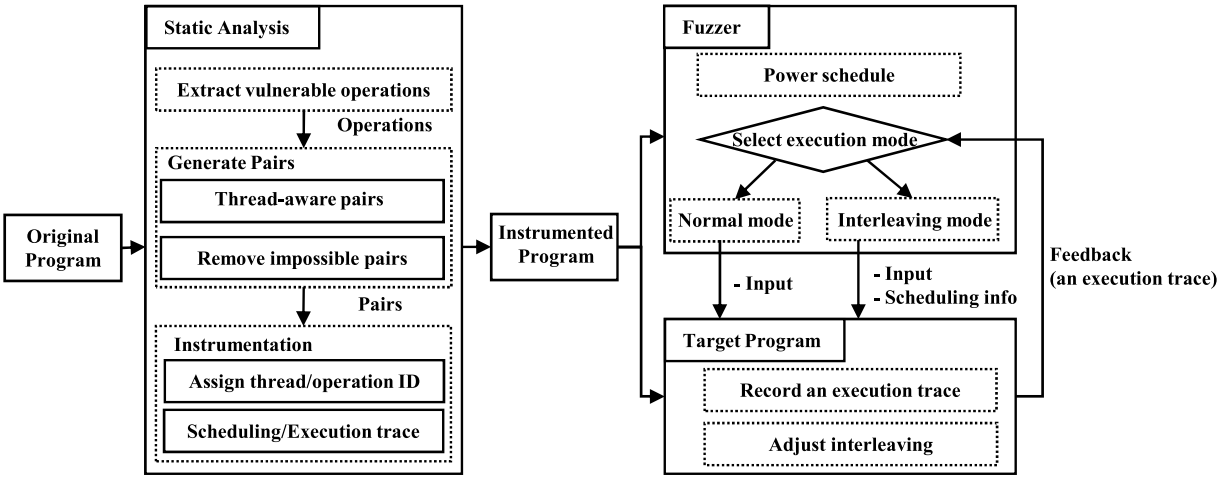


Fig. 4. Overview of AutoInter-fuzzing.

enter the interleaving mode using the feedback (operation IDs, thread IDs, and memory addresses) obtained from the normal mode. If two operations access the same memory address in different threads and are the pair found through the static analysis, the fuzzer enters the interleaving mode. The fuzzer executes two operations in both orders in the interleaving mode with the same test case used in normal mode. Suppose the fuzzer does not find two operations that access the same memory region and run in different threads as the result of static analysis. In that case, the fuzzer runs the normal mode continuously to generate test cases and feed test cases to the target program without controlled interleaving. When a crash occurs, the fuzzer records the test case and the execution order of the two operations used to execute the target program. The recorded test case and the execution order help to validate and reproduce the crash.

In the following sections, we use Code 3 to explain how static analysis works, what result it produces, and how static analysis results are used in dynamic fuzzing. In addition, we describe how the dynamic fuzzing works to test Code 5 program.

Code 3: Example of the target program having a use-after-free vulnerability

```
1  void* init (){
2      int* a = alloc(20);
3      return a;
4  }
5
6  void compute (void *p, int idx){
7      p[1] = p[idx] + 1;
8  }
9
10 void worker (void * p){
11     p[0] = 1;
12     compute(p,0);
13 }
14
15 void main(char * input){
16     int* memory = init();
17
18     if (input == 'A'){
19         pthread_create(worker,
               memory);
20     }
21     else if (input == 'B'){
22         compute(memory, 2);
23     }
24
25     free(memory);
26 }
```

4. Static analysis

The static analysis aims to find suspicious operation pairs that may trigger concurrency bugs. AutoInter-fuzzing will interleave the execution of operations in these pairs to expose potential concurrency bugs. A static analysis works in two steps: extracting vulnerable operations and generating pairs. We apply a points-to analysis to extract vulnerable operations that access or handle shared memory regions. Using these operations, we generate pairs in two phases. In the first phase, we naively pair the two operations in different threads that access the same memory address. Then, in the second phase, we remove false positive pairs where the two operations of the pair cannot be executed concurrently. Each step of the static analysis is described in detail as follows.

4.1. Extracting vulnerable operations

We need to extract vulnerable operations to find the interleaving space to be explore because not all operations create concurrency vulnerabilities. We target concurrency memory vulnerabilities such as use-after-free, NULL-pointer dereference, and double-free. The concurrency memory vulnerabilities are caused by accessing the invalid memory region. In a multi-threaded program, operations accessing shared memory between threads can make the valid memory region into the invalid memory region and create concurrency memory vulnerabilities. Therefore, we define operations that access the shared memory region between threads as vulnerable operations. The following list shows the types of memory operations that can cause concurrency vulnerabilities:

1. Store/Load operation accessing shared memory and global variables.
2. External Function calls with arguments used to access shared memory.
3. Memory allocation and de-allocation operations.

Memory access operations to global variables and shared memory regions may trigger concurrency bugs since they can affect other thread behaviors. We also handled the local pointers that access the shared memory region and global variable. Calling external functions with pointer arguments can potentially cause

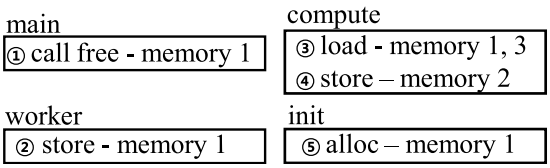


Fig. 5. The result of extracting vulnerable operations of Code 3 for each function. We know the memory regions that the operation may access in run-time.

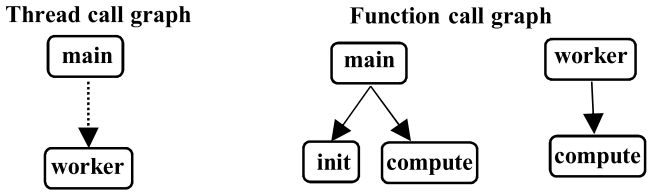


Fig. 6. The thread and function call graph of Code 3.

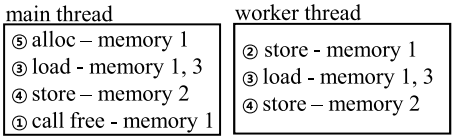


Fig. 7. Collecting relevant operations in Code 3 for each thread. The number in circle means the code location of the operation.

concurrency bugs. The pointer arguments can be used by store or load memory operations inside the called functions and generating bugs (Caballero et al., 2012) such as use-after-free and double-free. Memory de-allocation operations are closely connected with the concurrency bug such as use-after-free. Checking of memory allocation operations is necessary to know the memory size to be de-allocated for operations that access this de-allocated memory region.

We utilize a points-to analysis to extract these operations to determine the memory region where operations may access. We identify an operation with its code location because it is analyzed statically. As a result, we obtain the code location of operations whose type is one of the above three and memory region that the operations may access during run-time. Fig. 5 shows the result of extracting vulnerable operations from Code 3. We get the code location ①, the operation 'free', and the accessed 'memory' in the *main* function. Suppose the operation accesses multiple memory regions in run-time due to multiple function calls. In that case, the operation has multiple memory region numbers, like the load operation in the *compute* function.

4.2. Generating thread-aware operation pairs

We pair two operations that may have a concurrency bug in this step. Because concurrency bugs are triggered by unexpected interference by other threads, we pair any two operations in different threads that access the same memory region concurrently during run-time. We generate pairs in two phases. In the first phase, we use the scheme used in the kernel analysis tool Razzer (Jeong et al., 2019) with modification. The second is the optimization phase that considers the reachability and thread relationship in user programs to remove false-positive pairs. In the following subsections, we describe both phases in detail.

4.2.1. Phase 1: Naively collecting pairs

We collect relevant operations for each thread to determine which thread executed the vulnerable operations. Then, we pair

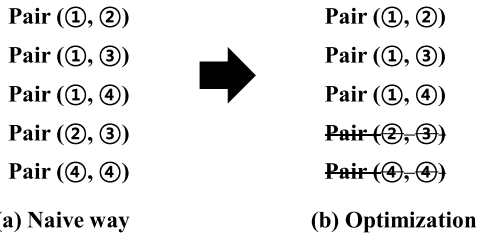


Fig. 8. The pair consists of two operations that may access the same memory region and run concurrently in different threads. We represent the two operations of a pair as the locations in the code. (a) the pairing result of the first phase and (b) the pairing result of the second phase.

two operations in different threads that may access the same memory region using the method used in Razzer (Jeong et al., 2019). Since the method used in Razzer is for the kernel, we modified it to be used for user programs. In a user program, a thread is started with an entry function. Also, we can list the functions executed by the threads. We construct a thread call graph and a function call graph to collect the operations executed by each thread. For Code 3, we can get the thread call graph and function call graphs, as shown in Fig. 6. Using the thread call graph, we know each entry function of the threads. We leverage the function call graph to derive all functions executed from the thread entry function. For example, in Fig. 6, we can identify that the entry function of the thread is *worker* using the thread call graph. We know which functions are executed in each thread with the function call graphs. The main thread executes *main*, *init*, and *compute* functions, and the worker thread executes *worker* and *compute* functions. Based on the functions executed by each thread, we can identify the vulnerable operations to be executed in each thread. Fig. 7 shows vulnerable operations collected per thread for Code 3. The main thread executes *main*, *init*, and *compute* functions, so the vulnerable operations in these functions are collected. Likewise, we can also collect the operations of the worker thread.

To find pairs, we combine two operations that access the same memory region in different threads by checking the operations belonging to each thread. We can know which memory regions might be accessed by operations through the points-to analysis. Fig. 8(a) shows the result of the pairing through this process. According to Fig. 7, operation ② of the worker thread accesses memory region "1" and operation ③ of the main thread also accesses memory region "1". Thus, these two operations become a pair. Combining two load operations is excluded from pair search because multiple reads rarely create a concurrency bug. Also, the memory region de-allocated by operation ① in the main thread can be accessed by operations accessing the same memory region in the worker thread (②, ③, and ④), so they are paired. The operation calling external functions with pointer parameters is considered as the store operation when finding pairs. This is because it is difficult to know the exact use of pointers in the external function body — whether they are used for reading or writing.

4.2.2. Phase 2: Optimization with thread-aware pairs

The pairs found in the previous phase may include many false-positive pairs that are never executed concurrently. Two operations cannot run concurrently because of the timing of executing the operations in different threads or the executed paths according to a given input. We need to remove these false-positives to have a better performance in fuzzing.

The optimization phase checks reachability for thread-aware pairs to eliminate these false-positive pairs. In a control flow

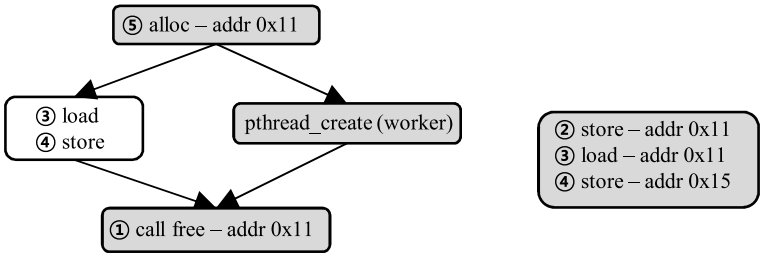


Fig. 9. Control flow of the threads in Code 3 when the input is 'A'. The grey boxes mean executed basic blocks according to the input. The number means the accessed memory address by operations in run-time.

graph, reachability means paths from one basic block to another. Operations before creating the child thread and operations executed in the child thread cannot be executed concurrently. In other words, the operations in the parent thread need to be executed between *thread_create* function and *thread_join* function to run concurrently with the operations in the child thread. Therefore, with a control flow graph, we check whether the operations can be reached after *thread_create* and before *thread_join*.

Fig. 9 illustrates the control flow of the threads in Code 3 when the input is 'A'. When checking the reachability in Fig. 9, the basic block having the load and store operations cannot be reached from *pthread_create* in the main thread. On the other hand, the free operation can be reached from *pthread_create*. Therefore, the pairs that include the store and load operations are removed because the pairs cannot run concurrently with operations in the worker thread. Thus the pairs (②, ③) and (④, ④) are removed (Fig. 8).

4.3. Instrumentation

We instrument several codes inside the target program to capture an execution trace and control the operations' execution order. We first insert the code that assigns thread and operation IDs to distinguish threads and operations executed in run-time. We insert the scheduling code to adjust the execution order of operations. To record an execution trace of the target program, the instrumentation stores the executed operations, threads, and accessed memory address.

4.3.1. Assigning context-aware thread and operation IDs

Multi-threaded programs have varying numbers of threads from two to hundreds and have many dynamically executed operations even if they are the same code operation. The fuzzer must discern which threads and operations are executed and controlled when controlling an intended thread interleaving. It is insufficient to mark threads and operations statically because the creation of threads and operations can be executed multiple times when executing loops and multiple function calls. For example, the operation ⑤ in the *log* function can be executed by different threads in the main and worker threads (Fig. 3). If an ID is assigned statically, the operations ⑤ executed in the main and worker threads have the same ID. Then, the fuzzer cannot discern the operations and attempt to control the execution order of wrong operations. Conservatively, we may apply the IDs assigned by the operating system (OS) to the threads. However, the IDs assigned by the OS change from one run to another. We need thread IDs independent of run instances to replay the vulnerable operation pairs of the target program with a given input.

The instrumentation assigns a context-aware ID to each thread and operation. We utilize the executed thread and operation sequence context to discern threads and operations executed dynamically. We first generate IDs of threads in the following way to reflect the context. The main thread has the ID of 1, and other threads have an ID that is a hash value of the parent thread ID (T_{id}), the number of times the child thread is created at the code location by the parent thread (C_{num}), and the location of the code creating the thread (loc).

$$Child_{tid} = \begin{cases} 1 & \text{if main thread} \\ Hash(T_{id}, C_{num}, loc) & \text{otherwise} \end{cases}$$

This ID is calculated dynamically at run time and always has the same ID over multiple runs for a given input. The parameters used to calculate the hash value are only dependent on the input. By including the parent thread ID in the hash value, we reflect the path context to the thread ID. Nested thread creations assign a fixed thread ID to each thread because each parent thread has its unique ID. Hence, we can handle complex interleavings.

We need to distinguish operations even with loops and multiple function calls to adjust the execution order. Our purpose is to identify which operation is executed in which thread. We know the thread's ID for an executed operation and the operations executed sequentially in a thread. Therefore, we discern the operations using *(the thread ID, operation sequence in a thread)*. For example, in Code 3, the ID of the load operation in the *compute* function is (1, 2) in the main thread and (2, 2) in the worker thread (we assume that the ID of the worker thread is two for explanation).

The light-grey box in Code 5 shows the instrumentation to hook thread creation and assign the thread IDs and operations IDs. Before creating a new thread, the thread ID is calculated using the hash function in line 21 of Code 5 and is passed on to the new thread. We replace the original function with our hooking function and pass the original entry function of the thread, its argument, and the new thread ID as arguments (line 22 of Code 5). The hooking function sets the new thread ID and initializes the number of operations executed and the number of threads created. In a thread, operations are executed sequentially; thus, the first executed operation starts with ID one, and subsequent operations have IDs continuously increased by one.

Code 4: Scheduling code injected before and after each suspicious operation to control interleaving when a pair is (first operation ID, second operation ID)

```
1  void before(){
2      if (operation ID == second
           operation ID &&
3          the first operation has not
               been executed)
4              wait;
5  }
6
7  void after(){
```

```
8       //record accessed memory,
            operation ID
9       record(accessed memory address)
            ;
10      if(operation ID == first
            operation ID
11          && the second operation is
               waiting)
12          wake_up(second operation ID
               );
13  }
14
15  void worker (void * p) {
16      before();
17      p[0] = 1
18      after();
19      compute (p,0);
20  }
```

4.3.2. Scheduling code

The fuzzer controls a thread interleaving, i.e., the execution order of two operations in different threads, by sending a scheduling directive to the target program to expose a concurrency bug. The scheduling directive consists of the two operation IDs and their execution order (first operation ID, second operation ID). The first value in the directive indicates the operation that should be executed first. We insert the scheduling code around operations in the target program to control the execution order based on the scheduling directive.

The functions in Code 4 are instrumented to control the execution order of the two operations. When there is an operation pair (A, B) to be fuzzed, we instrument the target program by inserting *before* and *after* function calls around A and B like lines 16 and 18 in Code 4. This scheduling code's *before* and *after* functions work as follows to control the execution order. Let us assume that there is a pair (A, B), operation A first and operation B later. The *before* function checks whether the thread and operation IDs match with operation B. If B has reached first and A has not been executed yet, the program waits without executing operation B in the *before* function. The *after* function also checks the thread and operation IDs. After executing operation A, the target program resumes execution of operation B if operation B is waiting.

4.3.3. Recording an execution trace

The target program needs to record an execution trace to be used as feedback by the fuzzer. Since the fuzzer only needs to interleave the pairs in which operations access the same memory address in different threads, it requires an execution trace that shows what threads and operations are executed in run-time to decide whether to interleave threads or not. We insert the code that records the operation located in the target program, the context-aware IDs of executed operation and its thread, operation types such as *store*, *free*, and the accessed memory address. The code is inserted in the *after* function that we put next to the vulnerable operation for controlled scheduling.

In the case of allocation operations (e.g., *malloc* and *calloc*), we additionally record the start address and allocation size. The de-allocated memory block and the memory accessed by other operations may overlap. Since the start address of the memory region is used for de-allocation, we need to know the memory size to find operations that access this overlapping memory. For example, in Code 3, the worker thread stores a value to 'memory' with index one, but the main thread de-allocates 'memory' with the address of index zero. Although the accessed address and address used for de-allocation are different, whether the same memory region is accessed should be determined. Thus, we record the allocated size to determine whether the operation accesses the memory region that may be de-allocated. For example, after executing the allocation operation in the main thread, the instrumentation records the execution trace as (5, 0 × 11, alloc, operation #1, main thread, 20). The first value is the code location of allocation operation found with the static analysis. We also store the operation type 'alloc' and 0 × 11 is the memory address that the operation allocates, and we store the allocated size '20'.

Code 5: Instrumentation of hooking thread creation and assigning context-aware IDs. The light-grey box indicates the instrumentation

```
1   struct pass{
2       void *(*fun)(void*);
3       void *arg;
4       u32 tid;
5   }
6
7   void hooking_fun(void* pass){
8       __thread u32 T_id = pass->tid;
9       __thread u32 operation_id = 0;
10      __thread u32  C_num = 0;
11      pass->fun(pass->args);
12  }
13
14  void main(char * input){
15                  ...
16      struct pass p;
17      //original called function
18      p.fun = worker;
19      //original argument
20      p.args = a;
21      p.tid = H(T_id, C_num, loc);
22      pthread_create(hooking_fun, p);
23      C_num++;
24                  ...
25  }
```

5. Dynamic fuzzing with automatic interleaving

In the dynamic fuzzing step, we aim to detect concurrency bugs by combining automatically controlled interleavings with bug-inducing inputs. The main parts of the fuzzing are interleaving-aware power schedule of seed for bug-inducing inputs, selecting mode using feedback from the target program, controlling thread interleaving, and recording crash information to validate a crash.

5.1. Power schedule of seeds

Coverage-guided fuzzers have a power schedule method to decide how many test cases to be created from a given seed. The number of created test cases from a seed is called "energy" in the power schedule. The seed covered more paths has a high probability of generating test cases that explore deeper paths and sometimes crash. Hence, it is desirable to give more energy to the seed that covered more paths than the seed that covered fewer paths (Böhme et al., 2017). If a seed has high energy, the fuzzer mutates the seed more times to generate more test cases.

We propose an interleaving-aware power schedule method to enlarge the explored interleaving space. In our method, the number of new test cases (energy) is determined according to the number of threads, operations, and pairs executed based on the

following reason. The input creating more threads is considered a meaningful seed since concurrency bugs exist in multi-threaded processes. If many operations are executed only in one thread, and few are executed in other threads, the available thread interleaving space decreases. To increase the thread interleaving space, it has to execute operations evenly across threads. We calculate the balance β of a seed representing how evenly distributed the executed operations are across threads and consider the seed to have a good balance as a meaningful seed. In addition, to detect vulnerabilities, it is necessary to test a large number of thread interleavings. Thus we use the number of operation pairs that can be interleaved to determine a meaningful seed. Then, we show how to decide the number of new test cases from the seed using the number of threads, operations, and pairs executed.

When the target program is executed, the program records the execution trace that includes the accessed memory address and the IDs of the executed threads and operations using the instrumentation code. We can obtain how many operations were executed per thread, how many threads were executed, and how many operations pairs were covered. For example in Code 3, the target program records that (5, 0 $\times$ 11, alloc operation #1, main thread), (1, 0 $\times$ 11, free operation #2, main thread), (2, 0 $\times$ 11, store operation #1, worker thread), (3, 0 $\times$ 11, load operation #2, worker thread), and (4, 0 $\times$ 15, store operation #3, worker thread) when the input is 'A'. We know that two operations are executed in the main thread, three are executed in the worker thread, the total number of executed operations is five, and the total number of executed threads is two. It also finds three pairs of which two operations access the same memory region in different threads.

We calculate the energy per seed as follows. We count the number of covered pairs P_{num} whose two operations are in different threads and access the same memory. We define balance β to reflect that operations are executed evenly across threads. To obtain balance β, we leverage the chi-squared test to get the discrepancy between the expected and actual results.

$$\beta = \frac{\sum(X_i - E)^2}{E}$$

where X_i denotes the number of executed vulnerable operations, which are obtained through instrumentation, in the ith thread, and E is the expected result. The expected result E is an equal number of operations executed per thread (total number of executed operations/total number of executed threads). Since the chi-squared test means the discrepancy between the expected and actual results, large β implies that executed operations are not evenly distributed. Therefore, to consider the seeds that cover more interleaving space favorably, we design the power schedule as $\alpha \frac{P_{num}}{\beta}$, where α is an empirical constant parameter. We assign energy to seeds in proportional to P_{num} and inversely proportional to β. For the above trace example, P_{num} is three, and β is 0.2. Hence we assign the energy 15α to the seed.

5.2. Selection of an execution mode using feedback

The fuzzer obtains the feedback in run-time from the target program by the instrumentation. The feedback is a trace of the executed operations by recording the location at the code, the accessed memory address, and IDs of operations and threads. For example, when the input is 'A' (Fig. 9), the target program records (the code location, accessed memory, operation IDs, thread IDs) as described in the previous subsection.

The fuzzer selects either the normal or interleaving mode depending on the feedback. To determine which mode the fuzzer has to execute, the fuzzer checks whether new operation pairs require thread interleaving. We have a list of operation pairs found through the static analysis. Suppose two operations in different threads access the same memory address, and the location of two operations is a pair in the list. In that case, the fuzzer obtains the executed pair and enters the interleaving mode.

For example, in the case of Code 3, we have the list of pairs such as (1,2), (1,3), and (1,4), shown in Fig. 8. In the case that a pair includes *free* operation, the fuzzer inspects whether the memory address accessed by the other operation overlaps with the memory region de-allocated by the *free* operation. If the memory region being accessed overlaps with the memory region de-allocated, the fuzzer judges that it also accesses the same memory region and enters into the interleaving mode. For the above example, the memory region de-allocated by the main thread and the memory region used by operations in the worker thread overlap. Hence, the fuzzer extracts three pairs that need to be interleaved — ((2),(3)), ((2),(4)), and ((2),(5)) that are real pairs whose interleaving schedule to be controlled at run-time. If the interleaving of these pairs has not been tested, we enter the interleaving mode and control the execution order of these pairs.

If there are no available pairs or have already been tested, the fuzzer enters the normal mode without controlling the thread interleaving. For example, when the input is 'B' in Code 3, no pair can be controlled because the target program does not create 'worker' thread. Then the fuzzer continues the normal mode by feeding input to the target program and obtains feedback of an execution trace of the target program.

5.3. Interleaving mode - Controlling thread interleaving

In the interleaving mode, the fuzzer sends a scheduling directive to the target program to have the wanted thread interleaving. The target program controls the execution order in run-time as in the directive. When (A, B) is in the directive, the target program runs twice, one by setting first = A and second = B and again by setting first = B and second = A to test both orders. The instrumented code checks whether the operation and thread IDs match with the information in the directive and controls based on settings. Finally, the fuzzer marks that the pair (A, B) is executed and continues with the next pair.

To ensure that previously tested thread interleaving of the two operations does not repeat, we mark tested thread interleaving, and only newly discovered interleaving candidates (newly covered pairs) are tested. However, if the memory address to be accessed differs from a covered pair, it is considered a new interleaving candidate, and the interleaving is scheduled again. It would require a repeated re-execution of the test program with different interleavings until testing all obtained pairs.

We do not remove the instrumentation code at run-time because the operations tested may be paired with other operations. Then we need to control the execution order of the operations even tested before. For example, suppose that operation A is also paired with operation C. When testing a pair (A, C), the fuzzer requires operation A's before and after function calls. Hence we do not remove the instrumentation after testing the pair (A, B).

5.4. Validation of crashes

The conventional fuzzer saves a test case (an input) used in the target program to validate the crash when a crash occurs. Since the same input usually explores the same path in a single-threaded program, it is possible to validate and reproduce the crash. However, concurrency bugs in multi-threaded programs require running the same path and the thread interleaving triggering the bugs. Therefore, we save a log that contains the input that caused the crash, the two operations that were manipulated if in interleaving mode, and their order.

For an example of Fig. 9, we can trigger the use-after-free vulnerability by controlling the execution order of a pair [(1, 0 × 11, free operation #2, main thread) and (3, 0 × 11, load operation #2, worker thread)]. When a crash occurs, the fuzzer saves the test case 'A' and the operation IDs [(free operation #2, main thread), (load operation #2, worker thread)] with the execution order as a log.

We do not need to record a log if it does not create a problem while testing a target program. Since the fuzzer knows the test case and the execution order of two operations before running the program, the fuzzer can save a log after detecting a crash or identifying a problem in the test program.

6. Implementation

We implemented AutoInter-fuzzing to detect concurrency bugs for programs written in C with POSIX multi-thread functions (e.g., *pthread_create* and *pthread_join*). Next, we describe the detailed implementation of static analysis and dynamic fuzzing.

6.1. Implementation of static analysis

To implement a static analysis, we utilized LLVM 9.0.0 and SVF (Sui et al., 2016), which provides a points-to analysis service. Although SVF is a valuable pointer analysis tool, it does not support several operations such as NULL store, non-pointer type global variable access, external functions call with pointer arguments, and memory de-allocation. Hence we modified SVF to handle these operations. To generate thread-aware operations pairs from a points-to analysis, we wrote a program in Python. We constructed a thread graph and a function call graph to remove impossible pairs, which reduces the interleaving space to be explored. To check the reachability of the operations, we utilized an LLVM pass that returns whether the two operations are potentially reachable. Extracted pairs are saved in a database (MySQL), and the fuzzer uses the result of static analysis to check if there are pairs whose execution order can be controlled in the dynamic fuzzing step.

6.2. Instrumentation for interleaving control

The significant parts of the instrumentation are assigning context-aware IDs, controlling the execution order, and recording an execution trace. To inject the instrumentation, we modified the AFL-llvm-pass, which compiles the target program to coverage-guided programs in AFL. We perform four instrumentations in the target program to test the program through AutoInter-fuzzing.

Hooking thread creation. We replace the original function with our hooking function and pass the following information as arguments to the hooking function: the original function, its arguments, and ID of the child thread. The hooking function sets the thread ID of the child thread. Also, it initializes the number of operations executed and the number of threads created that are needed to generate context-aware IDs of the child thread, as shown in Code 5.

Assigning context-aware IDs. For each thread, we use thread-local storage (TLS) to store a thread ID, the number of executed operations (operation_id), and the number of threads created (C_{num}). Because operation IDs are determined for each thread, the "operation_id" represents the current operation ID in the thread. The count (C_{num}) increases automatically with each thread creation. The location of operations (loc) is determined statically from the code. We use the hash function to calculate the thread ID before a thread creation code at run-time, as like line 21 of Code 5.

Scheduling code to adjust the execution order. We inject the scheduling code, Code 4, to control the execution order of two operations. The *before* and *after* functions are inserted around the operation. To wait or wake up the execution of operations, we call *sigwait* and *pthread_kill* functions to send a signal.

Recording an execution trace for feedback. While running the target program, it records the location of operation in the code, context-aware IDs, operation types, the accessed memory address, and the allocated memory size by the executed operations. We inject the recording code in the *after* function to know the accessed memory like line 9 of Code 4. To deliver the execution trace as feedback to the fuzzer, we store the execution trace in the DB using MySQL.

6.3. Implementation of dynamic fuzzing

We implemented our fuzzer based on AFL by adding the interleaving mode and modifying the power schedule. The fuzzer obtains feedback (executed operations and accessed addresses) from the target program through shared memory. After comparing the feedback with the results of static analysis (i.e., extracted pairs) in the DB, the fuzzer decides the execution mode (normal mode or interleaving mode). To detect some memory-related bugs (e.g., use-after-free) that do not result in a crash, we leveraged ASAN (Serebryany et al., 2012). When crashes occur, we leave a log (a test case and the order of execution of two operations) to reproduce the same path and interleaving, which helps identify bugs or vulnerabilities.

7. Experimental evaluation

We evaluate AutoInter-fuzzing by targeting the following questions:

RQ1 What is the capability of AutoInter-fuzzing in detecting concurrency vulnerabilities?

RQ2 What is the effect of the power schedule on the bug-finding capability?

RQ3 What is the effect of pair generation in AutoInter-fuzzing?

RQ4 Can AutoInter-fuzzing reproduce the crash due to the concurrency vulnerabilities?

7.1. Evaluation setup

Fuzzing Benchmark. We selected 13 multi-threaded C programs used in other researches (Liu et al., 2018; Chen et al., 2020) and Github. The selection criterion is whether the benchmarks are written in C with POSIX multi-thread functions since our method was implemented to detect concurrency vulnerabilities for programs written in C with POSIX API in static analysis and control of thread interleavings. We included all benchmarks from ConAFL (Liu et al., 2018) and selected some benchmarks from MUZZ (Chen et al., 2020) that use the POSIX API. The tested target programs are listed in Table 1.

Seed Corpus. We used the same seeds from small text files to evaluate compression, decompression, and downloader categories programs. We used a simple array as a seed for boundedbuf, swarm and qsort. We also used files provided as seeds by MUZZ when testing libwepb and x264. For the parallel downloader program, we set up a localhost server using Apache, and the fuzzer mutated a local file while the program downloaded the file from the localhost. We used a protein or nucleic acid sequence file to test pfscan and implemented a multi-threaded program to test ctrace. For SSDB, we used commands in the

Table 1
The benchmarks: Multi-thread C programs written in C using POSIX multi-threaded functions.

Category	Program	LoC
Parallel compression/decompression	pbzip2-0.9.4	1.5k
	pbzip2-1.1.13	4.4k
	lbzip2	3.3M
	bzip2smp	5.3k
Parallel downloader	aget-0.4.1	1.1k
Producer–consumer module	boundedbuf	0.4k
Parallel programming framework	swarm	2.2k
Parallel quick sort	qsort	0.7k
WebM for VP8/VP9	libwebp	7.3M
Video encoder	x264	11.5M
File scanner	pfscan	1.1k
Multi-threaded program tracer	ctrace	1.5k
Database	SSDB	6.7M

documentation as seeds. The lines of code (LoC) of tested real-work C programs varies from 0.4K to 11.5M, which indicates the scalability of AutoInter-fuzzing.

Environment Setup. All of our evaluations were performed on an Intel(R) Core(TM) i7-4770 CPU @ 3.40 GHz (8 MB cache) machine with 8 GB of RAM. The O.S. is Ubuntu 16.04 with Linux 4.4.0-169-generic 64-bit. We have applied AutoInter-fuzzing to a benchmark suite consisting of 13 real-world C programs. The evaluation of a specific benchmark was conducted on one machine.

7.2. Detected vulnerability (RQ1)

We tested AutoInter-fuzzing on real-world applications to check the efficiency of vulnerability detection while comparing it with other grey-box fuzzers: AFL, ConAFL, and MUZZ. When we tested AutoInter-fuzzing, ConAFL, and AFL for 24 h, AutoInter-fuzzing and ConAFL found vulnerabilities. We extended the test time of AFL for 36 h to check whether AFL could find the concurrency vulnerabilities if given enough time (AFL is not designed to test multi-threaded programs). We were able to test all benchmarks with AutoInter-fuzzing and AFL. However, since the static analyzer LOCKSMITH (Pratikakis et al., 2006) used in ConAFL is not scalable, we could not test large programs whose lines exceed tens of thousands by ConAFL. Furthermore, since MUZZ is not publicly available, we instead referenced the detection result of MUZZ in the paper.

Table 2 shows the detection results of vulnerabilities with AutoInter-fuzzing, AFL, ConAFl, and MUZZ. We denote the total number of unique crashes as N_c. Since AutoInter-fuzzing aims to detect vulnerabilities that are sensitive to the buggy thread interleaving, we divide detected vulnerabilities into two groups: Interleaving-relevant vulnerability and interleaving-irrelevant vulnerability. Interleaving-relevant vulnerability is triggered when controlling thread interleavings, whereas interleaving-irrelevant vulnerability is triggered regardless of thread interleaving. We denote these numbers of vulnerabilities as N_{cv} and N_{sv}.

AutoInter-fuzzing detected six interleaving-relevant vulnerabilities and four interleaving-irrelevant vulnerabilities, whereas AFL did not find interleaving-relevant vulnerabilities. In the benchmarks that ConAFL can test, ConAFL found two interleaving-relevant vulnerabilities, but it did not find interleaving-relevant vulnerabilities in pbzip2-0.9.4 and aget-0.4.1 found by AutoInter-fuzzing. Also, the vulnerability in ctrace was not discovered during fuzzing and was revealed when scheduled manually after inspecting the result of static analysis in ConAFL. Compared with MUZZ, AutoInter-fuzzing did not find the vulnerability in libwebp found by MUZZ. The reason is that the seed selection strategy is different, and the seed causing the vulnerability is not selected in AutoInter-fuzzing.

We describe interleaving-relevant vulnerabilities against AutoInter-fuzzing and ConAFL in Table 3. We mark the newly detected vulnerabilities with a star. In pbzip2-0.9.4, AutoInter-fuzzing found two concurrency vulnerabilities: the NULL-pointer dereference bug already known and a new NULL-pointer dereference bug. The known bug is the one in which the main() function frees the *fifo->mut* in the queueDelelete() function, but the consumer threads may still need *fifo->mut*. For the new bug we found, the main() function frees mutex *MemMutex*, but the consumer threads might use *MemMutext* to update the compressed data. However, ConAFL did not detect these two vulnerabilities. Because the NULL-pointer dereference vulnerabilities existing in pbzip2-0.9.4 are triggered from a mutex variable, the LOCKSMITH that ConAFL is using could not notice them. This result shows that AutoInter-fuzzing can detect vulnerabilities triggered in the wrong execution order even if they are data-race free.

We found the double-free vulnerability in bzip2smp like ConAFL. The double-free is triggered when sorting the block in the *threadFunction* and ending compression in the *writherFunction*. This vulnerability requires sophisticated interleaving between operations. Therefore, AutoInter-fuzzing that controls the execution order of two operations effectively finds such a hidden concurrency vulnerability.

AutoInter-fuzzing detected the unique vulnerability reported by ASAN in aget-0.4.1, but ConAFL did not detect this vulnerability. The heap overflow detected is not a concurrency vulnerability, but triggered when we control the execution order. The controlled execution order causes the download to fail and causes the program to resume. When resuming the program, the vulnerability is revealed. AutoInter-fuzzing was able to find the heap overflow since it has created the environment to trigger the vulnerability by controlling the execution order.

We detected a double-free vulnerability in ctrace library. The double-free occurs in the *trc_end* function and the *trc_remove_thread* function. Because the global variable *_thread* is not protected by a mutex, the two threads call the *free* function, respectively.

AutoInter-fuzzing detected a NULL-pointer dereference in SSDB. SSDB uses *ad hoc* synchronization for thread synchronization, but the destructor of the BinlogQueue class can be called before the del_range() function. The destructor of BinlogQueue sets a DB variable to NULL, and the del_range() function dereferences the DB variable. The controlled interleaving of AutoInter-fuzzing executes the destructor before the del_range() function, which causes a NULL-pointer dereference and crashes the program. We were able to find this new bug because we included operations accessing non-pointer type global variables in the thread-aware operation pairs and controlled the execution order of these operations. In addition, finding the vulnerability in the large-size program SSDB proves that AutoInter-fuzzing is robust against large-size programs.

7.3. Efficiency of power schedule (RQ2)

We validate the efficiency of the power schedule with time-to-expose (TTE), the number of interleavings, and the number of unique crashes. Because the power schedule aims to enlarge the interleaving space, we measure the efficiency against benchmarks having interleaving-relevant vulnerabilities. We ran each benchmark for four hours with and without the power schedule because we found the vulnerabilities in the benchmarks within 4 h. Table 4 shows the efficiency results of the power schedule.

Overall, the experimental results with the power schedule show faster TTE, more thread interleaving, and a higher number

Table 2
Fuzzing results of crashes and vulnerabilities on AutoInter-fuzzing, AFL, ConAFL, and MUZZ. Some programs are excluded since any fuzzers have not detected crashes/vulnerabilities. We reference the fuzzing results of MUZZ in their papers and mark '—' if the fuzzer is not applicable to the target. N_c: number of unique crashes; N_{cv}: number of interleaving-relevant vulnerabilities; N_{sv}: number of interleaving-irrelevant vulnerabilities.

Benchmark	AutoInter-fuzzing			AFL			ConAFL			MUZZ		
	N_c	N_{cv}	N_{sv}	N_c	N_{cv}	N_{sv}	N_c	N_{cv}	N_{sv}	N_c	N_{cv}	N_{sv}
bzip2smp	2	1	0	0	0	0	2	1	0	–		
pfscan	12	0	1	18	0	1	15	0	1	–		
ctrace	1	1	0	0	0	0	0	1	0	–		
pbzip2-0.9.4	35	2	1	12	0	1	0	0	0	–		
aget-0.4.1	4	1	0	0	0	0	0	0	0	–		
pbzip2-1.1.13	2	0	1	2	0	1	1	0	1	6	0	1
libwebp	0	0	0	0	0	0	–			19	0	1
x264	25	0	1	54	0	1	–			103	0	1
SSDB	5	1	0	0	0	0	–			–		

Table 3
Comparison of the detection results by AutoInter-fuzzing and ConAFL for interleaving-relevant vulnerabilities. The newly detected vulnerabilities are marked with a star.

Vul.	Benchmark	Vul. Type	AutoInter-Fuzzing	ConAFL
V_1	pbzip2-0.9.4	NULL-pointer dereference	✓	✗
$^*V_2$	pbzip2-0.9.4	NULL-pointer dereference	✓	✗
V_3	bzip2smp	Double free	✓	✓
$^*V_4$	aget-0.4.1	Heap overflow	✓	✗
V_5	ctrace	Double free	✓	✓
V_6	SSDB	NULL-pointer dereference	✓	–

Table 4
The efficiency of power schedule by measuring time to expose (TTE), the number of tested interleavings (N-Inter), and the number of unique crashes (N-Crash) with/without power schedule.

Benchmark	With power schedule			Without power schedule		
	TTE (s)	N-Inter	N-Crash	TTE (s)	N-Inter	N-Crash
pbzip2-0.9.4	182 s	1.35k	35	659 s	1.09k	19
bzip2smp	324 s	3.05k	2	320 s	3.02k	2
aget-0.4.1	2700 s	5.39k	4	2820 s	2.83k	1
ctrace	266 s	0.4k	1	325	0.3k	1
SSDB	125 s	25.2k	5	120 s	19.7k	2

of unique crashes. Since the power schedule helps the program create more threads, more interleavings can be tested. The table shows that the power schedule has better effectiveness in pbzip2-0.9.4, aget-0.4.1, and SSDB. They were tested with more interleavings and produced more unique crashes.

However, bzip2smp and SSDB have slightly less TEE when fuzzing without a power schedule. In the case of bzip2smp, each seed has similar energy with/without a power schedule. When the power schedule was applied, the non-buggy seed had slightly high energy, so the execution of the buggy seed was delayed. Therefore, the fuzzing with a power schedule took more TEE than fuzzing without a power schedule. In the case of SSDB, the seed initially selected in SSDB covers several pairs, so it has high energy when fuzzing with the power schedule. It spent time adjusting the thread interleaving of these pairs that were not buggy pairs. Therefore, the fuzzing with the power schedule took longer to reach a crash by exposing the concurrency vulnerability.

Some benchmarks do not show that the power schedule is effective when the seeds explore similar paths. However, the power schedule allows the program to explore more interleavings when running new threads or operations. If various seeds execute different paths, the efficiency of the power schedule can be further increased. Therefore, if there are abundant seeds, the power schedule can enlarge the interleaving space explored and help find concurrency vulnerabilities.

7.4. Efficiency of generating pairs (RQ3)

We find pairs that need controlled thread interleaving through two phases. First is the naive phase pairing relevant operations. The second is the optimization phase that removes false-positive pairs that cannot run concurrently. Generating only necessary pairs is essential because it increases the performance of fuzzing. Finding a bug within a limited time is vital in fuzzing since we cannot run the fuzzing indefinitely. To evaluate the efficiency of generating pairs, we measure the number of vulnerable operations, the number of pairs resulting, and the average execution speed of benchmarks from each phase.

Table 5 shows the efficiency of generating pairs. With the optimization phase, the number of vulnerable operations and pairs are reduced in all benchmarks except ctrace. In the case of ctrace, all pairs from Phase 1 (the naive phase) can run concurrently due to reachability; no pairs are removed in the optimization phase. Since the overhead of instrumentation is reduced in the optimization phase by removing false-positive vulnerable operations, the average execution speed of the benchmarks is increased.

We measure the bug detection capability depending on the method of pair generation. When testing benchmarks with pairs from each phase, AutoInter-fuzzing found six identical vulnerabilities on both phases. Since we use conservative optimization, we do not remove the operation pairs that may cause vulnerabilities in the optimization phase.

We also evaluate the bug detection time of the naive and optimization phases, respectively, for the six vulnerabilities, as shown in Table 6. The table shows that interleaving space optimization can reduce the time to detect vulnerabilities because the optimization reduces the execution time and the time for adjusting the execution order of impossible pairs. Since ctrace does not change the number of operations and pairs after applying the optimization, both take the same amount of time to detect vulnerabilities. Therefore, the optimization is efficient because it can improve the performance of the fuzzing by finding vulnerabilities faster in limited time resources.

7.5. Reproducibility of crashes due to concurrency bugs (RQ4)

We evaluate the reproducibility of crashes due to concurrency bugs because reproducing them is vital for bug analysis. We compare the reproducibility with and without the log. The log consists of the test case used as input, the two operations, and the execution order. We ran the experiment on a CPU in a stable state where no other program was running concurrently.

The crashes due to two NULL-pointer dereference vulnerabilities are not reproduced when we run the benchmark pbzip2-0.9.4 1000 times with the same input but without an interleaving log. However, we could reproduce NULL-pointer dereference in

Table 5
The number of vulnerable operations and pairs after the naive and optimization phases.

Benchmark	O_{naive}	O_{opt}	P_{naive}	P_{opt}	E_{naive} (execs/s)	E_{opt} (execs/s)
pbzip2-0.9.4	548	227(41%)	4193	841(20%)	0.81	1.65
pbzip2-1.1.13	1112	365(33%)	7861	1456(19%)	0.97	1.98
lbzip2	125	83(67%)	336	154(46%)	12.16	15.47
bzip2smp	677	425(62%)	2925	1234(42%)	1.06	1.63
aget-0.4.1	532	173(33%)	2664	1537(58%)	0.49	0.92
boundedbuf	29	18(62%)	33	18(55%)	0.5	0.55
swarm	94	26(28%)	123	36(29%)	7.46	12.41
qsort	122	52(43%)	134	58(43%)	9.06	16.66
libwebp	106	26(25%)	405	201(50%)	0.80	0.93
x264	1955	1340(68%)	369 259	305 324(82%)	0.12	0.34
pfscan	991	78(7%)	3096	935(30%)	0.93	26.89
ctrace	19	19(100%)	21	21(100%)	0.67	0.67
SSDB	2129	1245(58%)	236 689	167 379(71%)	0.05	0.07

Table 6
The time to detect vulnerabilities (average in 10 runs) with the naive and the optimization.

Vul.	Benchmark	Naive (s)	opt (s)
V_1	pbzip2-0.9.4	210 s	175 s
V_2	pbzip2-0.9.4	215 s	182 s
V_3	bzip2smp	580 s	324 s
V_4	aget-0.4.1	3340 s	2700 s
V_5	ctrace	266 s	266 s
V_6	SSDB	480 s	125 s

pbzip2-0.9.4 with the log by controlling the execution order of two operations. Our method can automatically validate the vulnerability because the target program already has a scheduling code. We only need to deliver the execution order and the thread and operation IDs of two operations we want to control through the shared memory. As a result, we can reproduce two NULL-pointer dereferences in one attempt. Likewise, crashes due to concurrency bugs in bzip2smp, ctrace, and SSDB are difficult to reproduce with only input, so they require an interleaving log.

These experimental results show that it is not easy to reproduce crashes due to concurrency bugs using only the same input. Reproducing them needs the same input and a specific thread interleaving. Therefore, AutoInter-fuzzing provides a necessary log and effectively reproduces them.

8. Discussion

We present several limitations of the current implementation of AutoInter-fuzzing and discuss their possible solutions.

Low path coverage compared with other fuzzers. AutoInter-fuzzing is aimed to reveal interleaving-relevant vulnerabilities but is not designed to explore more paths. As a result, our approach found several interleaving-relevant vulnerabilities that AFL could not find. However, it spends time adjusting thread interleavings without generating new test cases. Comparing paths explored in 24 h to AFL, AutoInter-fuzzing explores fewer paths.[1] We need to run our approach for 36 h to have the same path coverage as AFL. To complement this, we can combine AutoInter-fuzzing with other researches (Lyu et al., 2019; Rebert et al., 2014; Stephens et al., 2016) that have proposed an optimized mutation scheduling scheme, hybrid fuzzing scheme using symbolic execution, and seed selection scheme to increase the path coverage. We expect higher path coverage and testing of vulnerable thread interleaving when combined.

[1] We compared AutoInter-fuzzing with only AFL in terms of path coverage. ConAFL is implemented based on AFL and uses the same strategy as AFL to increase the path coverage. In addition, the path coverage result of AFL was better than ConAFL.

Run-time overhead. Although the evaluation shows that our optimization reduces the number of instrumentation places, our approach nonetheless incurs a run-time overhead. Moreover, adjusting the execution order for thread interleavings introduces an overhead by slowing program execution speed compared to normal execution. To overcome this problem, we consider adjusting the execution order of several pairs rather than adjusting the execution order of one pair at each run.

Supporting the limited interface. Our method currently only supports programs that use POSIX API for multi-threading. We need to extend AutoInter-fuzzing to cover programs using other multi-threading interfaces.

9. Related work

9.1. Detection tools for concurrency vulnerabilities

Tools can be classified into two perspectives: static or dynamic. Static detection tools (Pratikakis et al., 2006; Sui et al., 2016; Engler and Ashcraft, 2003; Deligiannis et al., 2015) analyze program paths and generate potential concurrency vulnerabilities and data races. For example, LOCKSMITH (Pratikakis et al., 2006) uses a constraint-based technique to detect the data races, and FSAM (Sui et al., 2016) suggests a sparse flow-sensitive pointer analysis using a thread-interleaving analysis. However, static tools result in a high false-positive rate owing to a lack of information at run-time and are not scalable.

Dynamic detection tools (Zhang et al., 2011; Park et al., 2009) monitor the accessed memory and synchronization by running the target program. In addition, dynamic tools control an environment to trigger bugs; for example, MAPLE (Yu et al., 2012) identifies concurrency bugs based on interleaving idioms. These tools test the target programs by adjusting the interleaving of threads to find possible thread interleavings for the given input. However, they are unsuitable for detecting concurrency vulnerabilities since they do not provide a method to generate the input to exploit the vulnerable path. They concentrated on reproducing interleaving bugs with a given input.

9.2. Grey-box fuzzing for multi-threaded programs

We introduce several fuzzing tools and compare them in three aspects: scalability, controllability, and detection of concurrency vulnerabilities. Because the real-world multi-threaded programs are large-sized, we check the scalability of fuzzing whether the fuzzing can handle large programs. We classify fuzzing tools as scalable if the fuzzer can test the program with more than 1M LoC. We measure the controllability to see whether the tool can control a specific thread interleaving. Also, we check whether the tools can find interleaving-relevant vulnerabilities, including

Table 7
Analysis of fuzzing for multi-threaded programs. Scalability: Tools can run a large-size program; Controllability: Tools can adjust a specific thread interleaving and handle dynamically executed operations; Interleaving-relevant vulnerability: Tools can detect interleaving-relevant vulnerabilities.

Fuzzing tools	Scalability	Controllability	Vulnerability
AutoInter-fuzzing	✓	✓	✓
ConAFL (Liu et al., 2018)	✗	△	✓
MUZZ (Chen et al., 2020)	✓	✗	✓
Krace (Xu et al., 2020)	✓	✗	△
Razzer (Jeong et al., 2019)	✓	△	△

concurrency vulnerabilities. Table 7 shows the comparison among the existing fuzzing along with our proposed method.

Several fuzzing techniques (Liu et al., 2018; Chen et al., 2020) have been suggested to test user programs. ConAFL (Liu et al., 2018) is a thread-aware fuzzing that explores as many thread interleavings as possible and detects several interleaving-relevant vulnerabilities. However, ConAFL cannot test the benchmarks exceeding tens of thousands of lines. Thus, ConAFL has low scalability and is less practical for testing large-sized programs. Although ConAFL controls the targeted thread interleaving, it cannot distinguish multiple executed operations (e.g., loop) and requires manual instrumentation to control the thread interleaving. MUZZ (Chen et al., 2020) is a thread-aware fuzzing tool that selects multi-threading-relevant seeds by distinguishing execution states in multi-threading contexts. MUZZ focuses more on selecting seeds for better fuzzing and does not control the interleaving of specific operations. Hence, it is considered that MUZZ has a different scope from AutoInter-fuzzing. Since MUZZ can test large-sized programs and detect multi-threading relevant vulnerabilities with seed selection, we consider that MUZZ can complement AutoInter-fuzzing in selecting multi-threading-relevant seeds.

Fuzzing targeting multi-threaded relevant bugs in a kernel has been reported (Jeong et al., 2019; Xu et al., 2020). RAZZER (Jeong et al., 2019) utilizes a customized hypervisor to control thread-interleaving to detect data races in the Linux kernel. RAZZER controls a thread interleaving by generating a deterministic sequence of syscalls. Meanwhile, in contrast to the kernel, user programs create threads dynamically depending on input at run-time. Thus RAZZER has difficulty in controlling a thread interleaving in user programs. Krace (Xu et al., 2020) is a kernel file system fuzzing used to detect a data race by applying alias coverage, lockset & happened-before modeling, and an evolution algorithm used to generate test cases. Krace explores interleaving using a delay injection. Thus it is hard to control a specific thread interleaving. Krace does not report vulnerabilities restricted by the same lock because the lock prevents data races. However, concurrency vulnerabilities can occur even at atomic accesses according to the order of execution. Razzer and Krace focusing on data races occasionally miss concurrency vulnerabilities.

10. Conclusion

In this paper, we presented AutoInter-fuzzing that detects interleaving-relevant vulnerabilities by controlling a specific thread interleaving and an interleaving-aware power schedule. The fuzzer extracted the pairs to be explored as the vulnerable interleaving space with the static analysis. We optimized the interleaving space by removing false positive pairs by checking thread-aware reachability. In the dynamic fuzzing step, the fuzzer used the interleaving-aware power schedule to generate test cases that could expose new thread interleavings. Also, it controlled the execution order of two operations using context-aware IDs to execute the untested thread interleaving. When a crash occurs, our method records a log including the input and the execution order to reproduce and validate the crash.

We applied AutoInter-fuzzing to 13 real-world programs of various sizes for evaluation. It reduced the interleaving space by removing the false-positive pairs by 49.6% on average and exposed execution orders hard to test without the power schedule. AutoInter-fuzzing detected six interleaving-relevant vulnerabilities, including concurrency vulnerabilities that AFL and ConAFL did not find. Moreover, AutoInter-fuzzing found four interleaving-irrelevant vulnerabilities. The proposed AutoInter-fuzzing tests more thread interleavings to expose concurrency bugs than the existing fuzzing methods.

CRediT authorship contribution statement

Youngjoo Ko: Conceptualization of this study, Methodology, Software, Writing – original draft. **Bin Zhu:** Supervision, Methodology, Investigation, Writing – review & editing. **Jong Kim:** Supervision, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2018H1A2A1063488/Global Ph.D. Fellowship Program). MSIT : Ministry of Science and ICT.

References

Andersen, L.O., 1994. Program Analysis and Specialization for the C Programming Language (Ph.D. thesis). University of Cophenhagen.
Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I., 2018. RacerD: compositional static race detection. Proc. ACM Program. Lang. 2 (OOPSLA), 1–28.
Böhme, M., Pham, V.-T., Roychoudhury, A., 2017. Coverage-based greybox fuzzing as markov chain. IEEE Trans. Softw. Eng. 45 (5), 489–506.
Burckhardt, S., Dern, C., Musuvathi, M., Tan, R., 2010. Line-up: a complete and automatic linearizability checker. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 330–340.
Caballero, J., Grieco, G., Marron, M., Nappa, A., 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 133–143.
Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., Liu, Y., 2020. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2325–2342.
2018a. CVE-2018-9514. Available from MITRE. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9514.
2018b. CVE-2018-9539. Available from MITRE. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9539.
2018c. CVE-2019–2000. Available from MITRE. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2000.
Deligiannis, P., Donaldson, A.F., Rakamaric, Z., 2015. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 166–177.
Engler, D., Ashcraft, K., 2003. RacerX: effective, static detection of race conditions and deadlocks. Oper. Syst. Rev. 37 (5), 237–252.
Godefroid, P., 1997. Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 174–186.
Hong, S., Kim, M., 2015. A survey of race bug detection techniques for multithreaded programmes. Softw. Test. Verif. Reliab. 25 (3), 191–217.
Jeong, D.R., Kim, K., Shivakumar, B., Lee, B., Shin, I., 2019. Razzer: Finding kernel race bugs through fuzzing. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 754–768.

Liu, C., Zou, D., Luo, P., Zhu, B.B., Jin, H., 2018. A heuristic framework to detect concurrency vulnerabilities. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 529–541.

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., Beyah, R., 2019. MOPT: Optimized mutation scheduling for fuzzers. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1949–1966.

Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2018. Fuzzing: Art, science, and engineering. arXiv preprint arXiv:1812.00140.

Miller, B.P., Fredriksen, L., So, B., 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM 33 (12), 32–44.

Musuvathi, M., Engler, D.R., et al., 2004. Model checking large network protocol implementations. In: NSDI, Vol. 4. p. 12.

Park, S., Lu, S., Zhou, Y., 2009. CTrigger: exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 25–36.

Park, S., Vuduc, R.W., Harrold, M.J., 2010. Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. pp. 245–254.

Peng, H., Shoshitaishvili, Y., Payer, M., 2018. T-Fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 697–710.

Pratikakis, P., Foster, J.S., Hicks, M., 2006. LOCKSMITH: context-sensitive correlation analysis for race detection. Acm Sigplan Not. 41 (6), 320–331.

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. Vuzzer: Application-aware evolutionary fuzzing. In: NDSS, Vol. 17. pp. 1–14.

Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D., 2014. Optimizing seed selection for fuzzing. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 861–875.

Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D., 2012. AddressSanitizer: A fast address sanity checker. In: Presented as Part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12). pp. 309–318.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS, Vol. 16. pp. 1–16.

Sui, Y., Di, P., Xue, J., 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. pp. 160–170.

Voung, J.W., Jhala, R., Lerner, S., 2007. RELAY: static race detection on millions of lines of code. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 205–214.

Xu, M., Kashyap, S., Zhao, H., Kim, T., 2020. Krace: Data race fuzzing for kernel file systems. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1643–1660.

Yu, J., Narayanasamy, S., Pereira, C., Pokam, G., 2012. Maple: a coverage-driven testing tool for multithreaded programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 485–502.

Zalewski, M., 2020. American fuzzy lop. URL https://lcamtuf.coredump.cx/afl/.

Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., Reps, T., 2011. ConSeq: detecting concurrency bugs through sequential errors. ACM SIGARCH Comput. Archit. News 39 (1), 251–264.

Zhang, W., Sun, C., Lu, S., 2010. ConMem: detecting severe concurrency bugs through an effect-oriented approach. ACM Sigplan Not. 45 (3), 179–192.

Youngjoo Ko received the B.S. degree in computer science and engineering from the Pohang University of Science and Technology (POSTECH), Korea, in 2017. She is working toward the Ph.D. degree in computer science and engineering at POSTECH. Her research interests include system security, hardware security, reliable programs, and vulnerability detection.

Bin Zhu received the B.S. degree in physics from the University of Science and Technology of China, Hefei, China, and the M.S. and Ph.D. degrees in electrical engineering from the University of Minnesota, Minneapolis, MN, in 1986, 1993, and 1998, respectively. From 1997 to 2001, he was a Lead Scientist with Cognicity, Inc., a startup company he cofounded with his dissertation Advisor and a Colleague in 1997. Since 2001, he has been a Researcher with Microsoft Research Asia, Beijing, China. His research interests include Internet and data security, privacy protection, and data and multimedia processing.

Jong Kim received the B.S. degree in electronic engineering from Hanyang University, Korea, in 1981, the M.S. degree in computer science from the Korean Advanced Institute of Science and Technology, Korea, in 1983, and the Ph.D. degree in computer engineering from Pennsylvania State University in 1991. He is currently a professor in Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Korea. From 1991 to 1992, he was a research fellow in the Real-Time Computing Laboratory of the Department of Electrical Engineering and Computer Science, University of Michigan. His major areas of interest are fault-tolerant, parallel and distributed computing, and computer security.