ConFuzz—A Concurrency Fuzzer

Nischai Vinesh, Sanjay Rawat, Herbert Bos, Cristiano Giuffrida and M Sethumadhavan

Abstract Concurrency bugs are as equally vulnerable as the bugs found in the single-threaded programs and these bugs can be exploited using concurrency attacks. Unfortunately, there is not much literature available in detecting various kinds of concurrency issues in a multi-threaded program due to its complexity and uncertainty. In this paper, we aim at detecting concurrency bugs by using directed evolutionary fuzzing with the help of static analysis of the source code. Concurrency bug detection involves two main entities: an input and a particular thread execution order. The evolutionary part of fuzzing will prefer inputs that involve memory access patterns across threads (data flow interleaving) and thread ordering that disturb the data dependence more and direct them to trigger concurrency bugs. This paper suggests the idea of a concurrency fuzzer, which is first of its kind. We use a combination of LLVM, Thread Sanitizer and fuzzing techniques to detect various concurrency issues in an application. The source code of the application is statically analyzed for various paths, from the different thread related function calls to the main function. Every basic block in these paths are assigned a unique ID and a weight based on the distance of the basic block from the thread function calls. These basic blocks are instrumented to print their ID and weight upon execution. The knowledge about the basic blocks in the sliced paths are used to generate new sets of inputs from the old ones, thus covering even more basic blocks in the path and thereby increasing the chances of hitting a concurrency warning. We use Thread Sanitizer present in the LLVM compiler infrastructure to detect the concurrency bug warnings while executing each input. The inputs are directed to discover even new address locations with possible concurrency issues. The system was tested on three simple multi-threaded applications pigz, pbzip2, and pixz. The results show a quicker detection of unique addresses in the application with possible concurrency issues.

N. Vinesh (✉) · M. Sethumadhavan
TIFAC-CORE in Cyber Security, Amrita School of Engineering, Amrita Vishwa Vidyapeetham, Coimbatore, India
e-mail: nischai.vinesh@gmail.com

S. Rawat · H. Bos · C. Giuffrida
Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

A. K. Luhach et al. (eds.), *First International Conference on Sustainable Technologies for Computational Intelligence*, Advances in Intelligent Systems and Computing 1045,
https://doi.org/10.1007/978-981-15-0029-9_53

Keywords Concurrency fuzzing · Concurrency bugs · LLVM · Fuzzing · Static analysis · Source code analysis

1 Introduction

The testing phase of any software development life cycle typically deals with checking the different functionalities of the software or the application. It is done by either manually testing the software or automated unit testing. Also, most of the software takes input from outside the system. Even though basic sanity checks are made to make sure proper inputs are passed to the application, an attacker can still craft inputs in such a way that it can crash the system or can be used to exploit the system. The external inputs to a system have always been an important attack vector. This is where the significance of fuzzing comes into play. Fuzzing is a software testing technique, initially developed to find implementation bugs using malformed or semi-malformed inputs given to the software/application in an automated way. It was further developed to find bugs at an earlier stage before they turn into vulnerabilities.

There were many researches on finding and eliminating bugs in single-threaded applications and there were variety of fuzzing techniques suggested for the same. As the bugs in the single-threaded applications can be exploited, bugs in the concurrency applications can also lead to concurrency attacks. With the increase in complexity and size of programs, the increase in multi-threaded programs in the recent past has been enormous. At the same time, there are not many studies on detecting concurrency bugs in these multi-threaded applications.

With the increase in size and the usage of processing power by different applications and the implementation of multithreading for more efficiency, the concurrency issues have become more prominent nowadays. Unlike generic memory corruption bugs, concurrency bugs are hard to find mainly due to non-determinism of thread scheduling. To add to the difficulty, the execution of the (multi-threaded) application on a given input should also involve patterns that may lead to an unintended situation, like use-after-free bug. Hence, the input (which triggers the concurrency bug) and the particular thread execution order are very important in order to detect these kinds of bugs. Though not many researches have been done to detect concurrency bugs, most of those done have a focus on manually controlling the thread scheduling to trigger a specific set of thread interleavings so that they will result in a concurrency issue. But this approach has its own pros and cons. In the real-time application, this particular set of thread interleavings may not occur in the specific order that it was assumed and the interleavings happen in a non-deterministic way. So instead of relying on controlling the thread interleavings, finding inputs that can trigger the same concurrency issues are more significant in solving the concurrency bugs in a multi-threaded application.

Finding such inputs, which takes a particular execution order, is not possible manually and hence fuzzing is an obvious direction to look forward to. The fuzzing of single-threaded applications to find bugs was a huge success and much critical vulnerability were discovered with this technique. But fuzzing a multi-threaded application blindly with random inputs and expecting it to hit a specific address with possible concurrency issue is not a practical solution. In case of concurrency bugs, the fuzzing technique has to be refined even more for very specific conditions. Instead of fuzzing randomly, specific addresses in the application with possible concurrency issues have to be detected first and then modify the inputs in such a way that they hit these initially detected memory addresses. This way of fuzzing is known as directed fuzzing. Source code analysis is the best and easy way of detecting all these significant memory addresses. Analyzing the application at binary level for this will not be feasible for very large applications. Also, source code can give even more information like the different paths from main function to these memory addresses, distances of these paths, etc.

LLVM [1] is a popular compiler infrastructure that provides various front end and back end tools for analysis and optimizations. There are different sanitizer tools, like Address sanitizer and Memory sanitizer in LLVM that performs analysis to find vulnerabilities in the code. Out of all these sanitizers, the Thread Sanitizer [2] looks apt for our purpose. TSAN gives runtime warnings about possible concurrency bugs including data races, deadlocks, thread leaks, etc., for a specific input execution.

Combining these possibilities, we arrived at a research question, whether the Thread Sanitizer in LLVM can be used to detect inputs that can trigger concurrency bugs in an application and LLVM infrastructure to instrument source code of the application to analyze the significant addresses and the input execution. LLVM can also be used to help generate more inputs that execute the paths to the initially detected significant memory addresses and trigger a concurrency bug.

2 Overview

This paper is divided into two parts—the instrumentation phase and the fuzzing phase. The instrumentation phase consists of instrumenting the source code with a custom LLVM IR pass and Thread Sanitizer (TSAN) of LLVM. The source code of the program is compiled into LLVM bitcode format and passed onto the custom pass. The custom LLVM pass inserts instructions at IR level. This modified bitcode file with instrumentations is recompiled along with thread sanitizer instrumentations into an executable. Then the final instrumented executable and a set of seed input files are passed to the fuzzing phase (Fig. 1).

In the fuzzing phase, the instrumented executable is run with the provided seed inputs. The fuzzer gets information about the execution path during each input execution and generates a report based on that. The thread sanitizer also generates runtime warning messages if any concurrency issues are detected. Based on these reports, next generations of the inputs are created.

We propose a fuzzing technique that is both evolutionary and application-aware. The source code analysis and instrumentation part of the system helps fuzzer in

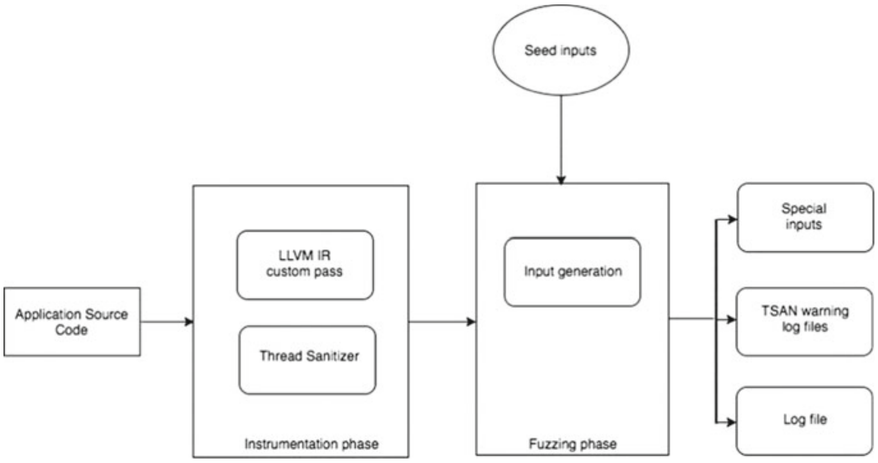

Fig. 1 Block diagram of overall system

understanding the information regarding the execution paths that lead to concurrency issues. It records all thread related function calls in the program and does a backward slicing to record the various paths leading to the call sites. The instrumentation pass takes care of loops and duplicate paths. The instrumentation also gives out important information to improve the next generation of inputs at runtime. The fuzzing component uses this information from the instrumentation phase to detect concurrency bugs at runtime for each input execution and saves the inputs that trigger it. The Fuzzer also generates specific number of new generation inputs using this information and passes it to the instrumented executable. This is repeated for a specific number of times which is configurable in the Fuzzer's config file. The Fuzzer makes sure it reports only unique bugs.

3 Concurrency and Multithreading

Concurrency is the ability of a program to run more than one task at a time. This allows for parallel execution of different tasks out of order, and the result would still be the same as that of those executed in order. The concept of concurrency and parallel processing in computing have become pervasive and critical because of the enormous usage of multicore processors and deployment of large scale distributed systems. Concurrent programs are written using threads and multiple threads are defined to carry out various tasks in parallel. Writing, testing and debugging the multi-threaded programs have continued to remain difficult since a long time and this impediment has led to many vulnerabilities and bugs in the concurrent programs. Addressing these challenges require advancements in many directions like concurrent program testing, concurrent program model design, concurrent bug detection, etc. Since concurrent programs exhibit more non-deterministic behavior, so do the bugs found in the concurrent programs and non-deterministic bugs are much more challenging when compared to normal bugs. Writing a bug-free concurrent program requires the developers to keep track of all the patterns in the interleavings between different threads that carry out concurrent tasks and the concurrently overlapping executions of different execution threads that use the shared memory. Creating test cases for all these scenarios and debugging a multi-threaded program execution makes it a notoriously difficult task.

3.1 Concurrency Bugs and Attacks

Testing is a common practice to discover bugs in a software but the existing testing techniques mainly focus on the sequential aspects of a program such as statement, branches, etc. and cannot address concurrency aspects like multi-thread interleavings within a concurrent program [3]. The exponential interleaving space of concurrent programs makes it even more challenging and exposing these concurrency bugs require a bug-exposing input along with the bug-triggering execution interleaving. Just as the errors in a sequential program can be exploited, concurrency errors can also lead to a security exploit and thereby the possibility of using it to carry out a concurrency attack, which can violate confidentiality, integrity, and availability of the system. Deadlocks in a concurrent program are a major issue, but most of the deadlocks can be avoided by the use of appropriate locks in the right places. Deadlock occurs when two or more operations circularly wait for each other to release the acquired resource and Datarace occurs when two conflicting accesses to one shared variable are executed without proper synchronization. Almost all the non-deadlock concurrency bugs can be classified into three main categories Atomicity violation, Order violation, and others. A study on Concurrency Bugs [4] describes that out of these three categories, atomicity and order violation constitutes the majority of the existing concurrency issues. Atomicity violations are caused by concurrent execution unexpectedly violating the atomicity of a certain code region. The order violations are caused when the program is executed in an order that does not follow the programmers intended order. The programmers atomicity and order assumption while writing the program could lead to most of these violations as the multi-threaded applications' thread interleavings are completely non-deterministic and it is very difficult to capture all these combinations. It is very common among the programmers to assume an order but enforcing that order is difficult, sometimes they may even forget to do it where it was possible. Most of these concurrency issues manifest due to the involvement of very small number of threads, in most cases there are just two threads.

The main reason behind this finding is that even though there are hundreds of threads used in a program, most threads do not closely interact with each other and most of the thread interactions happen between two or very small number of threads. Hence the vector space of concurrency issues can be reduced to a large extent by considering those threads that interact or collaborate. It is very much evident from the CVE database [5] that the concurrency bugs can be used to carry out a viable concurrency attack. A recent study [6] discusses and shows how the existing concurrency errors can be exploited to carry out an attack. To carry out such attacks, understanding and reproducing the concurrency errors produces a challenge due to its non-deterministic nature. The ability to exploit a concurrency bug depends on the vulnerability window. The Vulnerability Window is the size of the timing window within which the error may occur. The vulnerability window can be measured in terms of human time, disk access time and memory access time. An attacker using carefully crafted inputs can extend this vulnerability window to effectively carry out the concurrency attack. In some cases, these can be done using a sequence of UI events and in some other, the attacker can re-run a command few times, possible using a shell script. Exploiting the errors during the memory access time is more difficult compared to the other two as the offending events should occur within small timing windows along with the hardware cache leases or CPU time slices being larger than the timing windows masks the errors.

The current concurrency bug detectors are not mature enough as that of sequential tools and most of the bug detectors and the defense techniques are created to serve the sequential programs. These commonly used sequential defenses become unsafe when applied to concurrent programs. Through ConFuzz, we are trying for the first time to build a fuzzer that can detect concurrency bugs.

3.2 Fuzzing

Fuzzing is an automated way of software testing that involves providing invalid, unexpected or random inputs to the program and trying to discover various issues with the combinations of code and data along with the knowledge of protocols and heuristics. It is used to detect exploitable bugs in the code and check for corner cases by providing random inputs to check for crashes. The most significant part of a fuzzer is the ability to generate inputs that can trigger a bug or crash the application. The first form of fuzzing was carried out by Barton Miller from University of Wiscosin, Madison [7] by supplying random inputs to Unix utilities and then observing for crashes. This resulted in the discovery of various exploitable vulnerabilities. Since then the complexity of code has increased over the years and a set of random inputs may not be enough to cover the application well enough to reach the deeply buried bugs. Even though there have been lots of improvements in the field of testing, we need more heuristics about the application for a fuzzer to be efficient.

A Fuzzing system mainly has three components—a generator, a delivery mechanism, and a monitor. These components carry out the three major tasks in the process of fuzzing. The generator is responsible for generating inputs, random, malformed or crafted, which have to be passed on to the system under test. There are various methods to generate inputs depending on the bugs that the fuzzer is trying to detect. The passing of inputs to the system is handled by the delivery mechanism. The delivery mechanisms may vary depending on the target applications way of receiving the inputs. Once the application to be tested is run with the generated inputs, the monitor will identify errors, bugs, and crashes. It also keeps track of the inputs that triggered the bugs, type of inputs that can trigger the errors, code coverage and other general statistics required for analysis and improvements.

Code coverage is one of the most important factors among the metrics that can be used to evaluate a fuzzer. It specifies the amount of code within the application that is tested. The higher the code coverage, the better the chances of finding vulnerabilities. Input space is another factor to measure the capability of a fuzzer. It consists of various combinations of inputs that the application accepts or that can be passed to the testing application. The possibilities of such an input space is large and hence different heuristics are used to constraint the space.

Many attack heuristics like buffer overflows, format string attacks, parse errors, etc. can be used to find bugs easily. Since the various inputs within the generated input space can be similar to each other, there is always a chance of the application taking the same execution path that leads to a crash. Hence only unique crashes need to be logged and the inputs that trigger duplicate crashes can be discarded. The number of unique crashes found by a fuzzer can also be used as a factor to evaluate fuzzer.

4 Design

The design of the concurrency fuzzer is divided into two parts, instrumenting the application for coverage calculation and concurrency detection, and generating inputs and fuzzing the application. The first part of the system is instrumenting the application. There are two types of instrumentations needed for the concurrency fuzzer. First, the source code of application has to be converted into bitcode format so that LLVM can apply the custom pass instrumentations at the IR level. Hence, all the files are to be converted and combined into one single bitcode file using LLVM tools. The custom LLVM pass, at the IR level, will find all paths from significant thread related function call sites to the main function using backward slicing and instrument each basic block within these paths. The thread related functions, which need to be considered, could be provided through a config file. A rank is calculated for each input based on the execution path that it takes. The application code also needs to be instrumented with LLVM Thread Sanitizer for detecting concurrency issues at runtime. We use TSAN warnings to detect and analyze concurrency issues at runtime for each input. After both the instrumentations are complete, the final bitcode file with instrumentations is to be converted into an executable. The fuzzing part of the system does the major part in detection of possible concurrency issues. It runs the instrumented application and reads the information output from the custom pass instrumented codes. This information is used in ranking each input. The parents for the next generation of inputs are considered based on this rank. The fuzzer also reads and analyzes the warning messages thrown by the TSAN during each run.

The input generation, a part of the fuzzing component, is mutation-based that requires minimum number of seed inputs to start with. The seed inputs are initially run and ranked to consider for creating the next generation of inputs. We use a similar strategy as that of VUzzer [8] for randomly selecting bytes. Then the seeds are passed to crossover function to generate new inputs and mutate them. The fuzzer monitors each input execution for detection of concurrency issues and discovery of new basic blocks within the analyzed paths to thread function call sites. These information are used to optimize the generation of next set of inputs.

4.1 LLVM Custom Pass

The custom LLVM pass is a transformation module pass, which inserts a function call at the specified target within a basic block. A separate C file with a custom print function is compiled, converted to bitcode file and then combined with the application bitcode file before any instrumentation is done. The print function accepts two parameters and prints them into a file, which is later used to calculate the rank of the input. The first parameter in the print function is an integer value, which will be the unique ID assigned to the particular basic block, and the second is a float value, which is the calculated weight of the basic block. The custom pass loops through each instructions and looks for critical sections on the code, which are the specific thread function calls (mutex lock/unlock, pthread lock/unlock, etc.) mentioned in the ConFuzz's config file, and creates a multi-linked-list of call traces from the function call sites to the main function using backward slicing. Once all the different paths are found, the basic blocks from the overlapping or repeated paths are avoided and only the unique basic blocks are kept. Each unique basic block is assigned with an ID and a Weight is calculated for it. The Weight of a basic block is based on how far it is from the thread function call site. The farther the basic block, the lesser it's weight. Whenever an instrumented basic block is executed, the custom print function is called and the respective ID and Weight are printed in file in the disk. After every input execution, this file is read by the fuzzer to calculate the rank by summing up the weights of basic blocks executed. The fuzzer also detects whether any new basic block is executed from the previous executions and keeps track of it. A high-level logic is explained in Fig. 2.

The custom pass takes care of loops of basic blocks within a function and loops of functions within the module. Also, the pass is designed to print only unique basic blocks to avoid overhead. The pass also skips TSAN related functions and function calls as they may trigger false datarace warnings.

```
Initialize TARGETS (Set of critical section/thread function call sites);
Initialize PATH, A multi linked list where all the paths from critical sections to main
  function are stored;
Initialize FUNCTION_NAMES, a set of thread function names specified in the config
  file;
while within the Module - Loop though Functions in the module do
    while within a function - Loop through Basic blocks of the function do
        while within a Basic Block(BB)  Loop through instructions of the BB do
            if instruction EQUALS function call of fn, fn ∈ FUNCTION_NAMES AND
              current BB not in TARGETS then
                Add the BB to TARGETS;
                Add BB as the head node of a new list in PATHS;
                GetPredecessors(BB) and add them to PATHS under the list;
                while While current function name NOT EQUALS main do
                    Find the parent function(s) and the call site of the current function;
                    Repeat the steps (10-13) for the parent function(s);
                end
            end
        end
    end
end
while Loop through the list of recorded BBs in the PATH do
    Calculate and Assign weight to BB;
    Instrument BB  insert instruction to call the custom print function;
end
```

Fig. 2 Algorithm of static analysis LLVM pass

4.2 LLVM Thread Sanitizer

As mentioned earlier, Thread Sanitizer is a LLVM tool to detect possible dataraces during runtime. When application code is instrumented with TSAN, it adds many TSAN related function calls into the source code to check for possible concurrency issues upon the execution of the application. During the execution of the instrumented code, TSAN raises warning messages when it detects a possible datarace or any other type of concurrency bugs. TSAN has the capability to detect many concurrency related issues including dataraces, thread leaks, deadlocks and many more by detecting shared memory accesses by two or more threads. It records information about each memory accesses and checks whether the memory access participates in a race. Each thread is instrumented to store its own timestamp and the timestamps for other threads for synchronization purpose. Timestamps are incremented every time a shared memory is accessed. By checking for the consistency of memory accesses across threads, data races can be detected independent of the actual timing of the access. Therefore, the Thread Sanitizer can detect races even if they didn't manifest during a particular run [9].

The concurrency fuzzer makes use of the TSAN warning messages to detect if the input generated can trigger a datarace or not. It also gives more priority for such inputs in the process of creating next-generation of inputs. There is a known slowdown of 5×–15× and a memory overhead of 5×–10× when running any application with TSAN instrumentation.

4.3 Input Generation

Fuzzing a complex application and finding bugs, which can be found in the deeper execution paths, is extremely difficult. It is even more difficult when it comes to detecting concurrency bugs. To find such bugs, prior information of the program structure is required and the inputs have to be crafted in such a way that their execution will reach the specific locations or follow specific input execution paths. There are different ways of achieving this. Fuzzers like AFL keep track of basic blocks executed and looks for inputs that explore new basic blocks with the help of a feedback loop. It then mutates those bytes in the input that do not help in detecting new basic block for better coverage. Symbolic/concolic execution is more efficient in finding those offsets in the input that are to be mutated and with what value, but the overhead is too high when it comes to large application. In case of fuzzing concurrency applications, the shared memory address along with the execution path need to be known and the inputs have to be directed towards it to have a chance of triggering the bug. Practically, this is extremely difficult without a proper static analysis of the application's source code.

We propose a fuzzer that is designed to be application-aware. It uses an evolutionary fuzzing strategy based on static analysis of the source code. The strategy is mutation-based which makes use of valid seed inputs provided to mutate and generate a new set of inputs. It makes use of information about the memory locations in the application that is shared by more than one thread and the different execution paths towards them. This is done by looking for *pthread* related function calls in the application and marking the basic blocks that contain the *pthread* related function calls as important targets to fuzz. The focus is also given to inputs that explore new paths within the previously gained set of execution paths.

The evolutionary fuzzing strategy starts with valid seed inputs that are mutated or crossover multiple times. The magic bytes of the seed inputs are maintained as such using the ConFuzzs config file so that the execution of input with the application does not crash due to invalid format. The crossover technique selects two-parent inputs and produces two children, while the mutation technique transforms one parent into a child. A combination of these two techniques, along with single or double crossover, is used to improvise the input generation. Each input that is generated is passed to the application to execute and are ranked based on its coverage of the recorded execution paths towards the shared memory locations. These ranks are used when considering parent inputs for the next generation of inputs. This process of execution, ranking, input generation, and monitoring continues until a concurrency issue is found or till a specified number of generations is over. This strategy is very similar to that of

VUzzer [8], which also weighs basic blocks and use that information to better the input generation.

5 Implementation

The implementation of the concurrency fuzzer and its different components are explained in this section. We will also discuss the approaches that we considered in implementing the fuzzer.

5.1 Instrumentation—Source Code Analysis

The implementation of concurrency fuzzer is divided into 2 parts. The first part is to analyze the source code of the application to be fuzzed. Depending on each application's build methods and the structure, all files of the application are converted into bitcode format using clang [10] emit-llvm and combine all bitcode files into one single file using llvm-link. This final bitcode file will also contain the defined print function that needs to be instrumented into the source code of the application. This way of conversion is applicable only to those applications that have a final single executable generated once it is built. In case of applications that generate multiple executables or use other ways like shared objects, we have to tweak the process to suit it or make use of the flto and llvm-gold plugin in the build scripts of the applications to generate bitcode files automatically. This method will generate single/multiple (according to the structure of the program) pre-codegen bitcode files before it creates the final executable(s). Hence, we just have to use these pre-codegen bitcode files and combine them together to make it a single final bitcode file to be instrumented.

5.2 LLVM IR Pass

Once the final bitcode file is ready, this file needs to be analyzed. Since the bitcode file is in LLVM IR level, it is easy to analyze with the help of llvm defined functions and properties. We wrote a custom llvm transformation module pass to analyze the generated final bitcode file.

The pass will loop through all the functions and basic blocks of the module and will look for specific function calls in the instructions list. This function calls can be configured using a PassConfig.txt text file. The names of the functions can be mentioned in this text file with a newline. If the file is empty or there is no file found with this name, the pass will consider the default values mentioned in the code. In our case, the function names will be thread related calls like pthread create(), pthread join(), pthread mutext lock(), pthread mutex unlock(), etc. Once a particular function call site is found, the pass will do a backward slice of the program from the function call site to the main function of the program. It records different paths separately in a connected linked-list. Since the paths are dynamically added during the analysis time, linked-list is preferred as each node of it can store the details of basic block like the address, the parent function and unique id assigned to every unique basic block. The pass looks out for loops at the instruction level, the basic block level, and the function level and bypasses it when recording the sliced path. Once all the sliced paths are found for every function mentioned in the PassConfig.txt, the unique basic blocks in these paths are instrumented with a function call instruction to the PrintID function. These basic blocks are also assigned with a weight depending on its distance from the thread function call. The nearest basic block will have a weight of one, the next nearest will have 1/2, the next will have 1/3 and so on. These weights are calculated in decimal values. The PrintID function takes two arguments, an integer value, and a float value. The first integer value is the basic block ID, which is unique for each basic block in the recorded sliced paths. The second argument is the weight of the particular basic block. The PrintID will print these two values with a space in between into a file, WeightID.txt, in the current path at runtime, whenever the instrumented basic block is executed.

5.3 Fuzzing Technique

The second part of the implementation is the fuzzer. We used Python 2.7 to implement the fuzzer. The fuzzer depends on two files, a config file, and the weightID.txt, to fuzz the application. The values of the configurable variables are taken from the config files by the fuzzer. The fuzzing technique can be divided into different steps as follows:

Configuration: The config file needs to be updated with proper values for the variables defined in it, before the fuzzing process starts. The config file contains the variables for the seed inputs folder path, generated inputs folder path, fuzzer log path, folder path for TSAN warnings, number of generations to be created, total population size to be maintained, percentage of best parents to be considered for creation of each subsequent generation. The magic bytes that need to be skipped while randomizing the input bytes can also be configured in the config file depending on the application and the input file format the application supports.
Initialization: The fuzzer starts with verifying whether the configuration variables are set properly. It also creates separate folders for keeping the inputs that explore new paths and for those that triggers TSAN warning(s). The command-line arguments, if any, for running the application needs to be defined in the fuzzer script itself.
Fuzzing: The fuzzer runs the instrumented application executable passed to it as runtime argument with the input files in the seed input folder. This dry run is to check if the application binary is working fine or not. Once the application is run with all the seed inputs, the fuzzer ranks each input using the weightID.txt file. The

rank of an input file is calculated as the total sum of the weights of the unique basic blocks executed. This information is gathered from the weightID.txt, as the weights of all basic blocks executed will be printed into this file during runtime. It also records how many unique basic blocks from within the sliced path are executed by reading the IDs of the executed basic blocks from the weightID.txt file. Once all the seed inputs are successfully executed with the application and the rankings of the inputs done, the fuzzer starts creating the next generation of inputs from the seeds based on their ranks. After the next generation is over, the fuzzer repeats the execution process until the number of generations mentioned in the config file.

Check for TSAN warnings: Upon every execution of the instrumented application binary, the fuzzer checks for any newly generated TSAN warning log files in the TSAN log folder mentioned in the config file. Whenever there is a concurrency issue, say a datarace, the thread sanitizer instrumented binary will generate a log file at the specified path with the details of the concurrency bug. Hence the fuzzer checks for such log files after the execution of each input file. The fuzzer reports the concurrency bug to the console only if it is a unique one. There is a chance of thread sanitizer warning about the same concurrency issue upon execution of application binary with different inputs. Hence the fuzzer has to check for unique TSAN warnings by reading the TSAN log files for the addresses of significance and record the same for comparisons in future.

Input Generation: The fuzzer uses a mutation-based evolutionary way of input generation. The approach is similar to the VUzzer's [8] logic of input generation. A set of parents is chosen based on elitism approach from the pool of inputs to create the next generation of inputs. The parents are chosen based on their ranks, calculated after the execution. There are different mutation and crossover operations defined and the parents undergo mutation and crossover operations randomly chosen by the mutation and crossover functions for creating new child input(s). The number of magic bytes defined in the config file is maintained as such for every parent inputs to avoid crash of the application due to invalid format. If an input triggers a TSAN warning or explores a new basic block from the sliced paths, the input will be copied to special folders and will be considered for each of the subsequent generations of inputs.

Statistics collection: The fuzzer maintains a log file for each fuzzing run. It logs information about the duration of fuzzing, number of files created for each generation and time taken for creating them, time of execution of inputs, number of TSAN warnings generated and the possible addresses which are vulnerable to concurrency bug and time taken to trigger these warnings. The fuzzer also records the number of basic blocks executed from the sliced path and logs if any input explores a new basic block out of this recorded list. The special folders contain those inputs that triggered the TSAN warnings and explored new basic blocks, which can be used to reproduce the concurrency issue and produce a TSAN warning.

6 Evaluation and Results

The evaluation of the concurrency fuzzing system done on three different applications is discussed below. The applications are as follows:

1. pbzip2 [11] parallel bzip
2. pigz [12] parallel gzip
3. pixz [13] parallel xz.

The evaluation was done by fuzzing these applications for a long time and for large number of generations and then observing the logs. The selection of applications had to be limited to open source (so that we could do the source code analysis), comparatively small (so that it is easy to instrument) and multi-threaded (which uses thread related functions) which takes a file as input. The application with known concurrency issues like datarace would be even better for us to verify and discover more bugs at the same time. The input file size also had to be kept as small as possible to keep the fuzzing process effective, at the same time maintaining the minimum input size to trigger the parallel processing.

6.1 pbzip2-1.1.13

bzip/bzip2 [14] is an extensively used open-source compressing utility, which has many advantages over other compressing applications. pbzip2 is the parallel version of bzip2 that uses pthreads to parallelize the compressing process. It takes file(s) as input and outputs compressed files compatible with bzip2. It uses thread related function calls like pthread create, pthread mutex lock, pthread mutex unlock, pthread cond wait and many more. This makes it a right candidate for evaluating the ConFuzz as it satisfies the conditions.

Experimentation

The application source code consists of mainly three files and it uses the bzlib library to do the compression. Since bzip does not use threads, we need not have to instrument bzlib library. So the three main source code files are converted to bitcode files, combined along with the bitcode file of printID into a single final bitcode file. All these files were instrumented with thread sanitizer also. The final bitcode file is passed to the custom IR pass for instrumentation. The instrumented final bitcode file is converted into an executable.

We considered seven files of varied sizes, from 2 to 500 MB, as seed inputs, as it was noticed that the dataraces were not triggered with smaller sized inputs. Hence we wanted to use a combination of files with different sizes and different file types like pdf, mp4, and text les. We have set the number of generations flag in ConFuzz to 1000 and the population size flag per generation to 100. The other variables in the ConFuzz's config file for different folder paths were also set like log path for Thread sanitizer warnings, input folder path, folder path for inputs that trigger concurrency issues and for inputs that explore new basic blocks during fuzzing. We have also passed the following parameters to the binary so that it is optimized for multithreading:

– b5: block size in 500k steps
– r: read complete file into RAM and split between the processors
– m500: Max memory usage of 500 mb

The output of each execution, which is a. bz2 compressed file, is removed before the next execution, to save the disk space, as we are dealing with large-sized files for fuzzing.

Results

The ConFuzz took more than 12 h to run 1000 generations. The tsan log files in the log folder gives the details of all the concurrency issues found per input execution and the ConFuzz log file gives information about the number of unique concurrency issues and the time taken to find it. The total summary of the run is also provided at the end of execution in the log file. There were 55 basic blocks instrumented in the various sliced paths from different thread function call sites to the main function and a total of 2197 basic blocks instrumented outside these sliced paths. Upon fuzzing, 56% of the basic blocks were executed from the sliced paths and around 23% of basic blocks outside the sliced paths were executed on an average. ConFuzz detected 15 unique dataraces during the run.

We compared the results of ConFuzz with our own implementation of a dumb fuzzer, which does not have any intelligence and just randomly changes the inputs and fuzzes the given application. Even though the source code of this dumb fuzzer is derived from ConFuzz, the logic of directed fuzzing by considering the executed basic blocks from the sliced path and the rank calculation is removed and all the basic blocks are instrumented evenly. The same inputs used in ConFuzz are also used for dumb fuzzer with similar configurations. It was noticed that the dumb fuzzer also found 13 unique dataraces in 6 min from the start of fuzzing, ConFuzz found 15 dataraces in 2 min and 10 s, which proves the efficiency of ConFuzz over the dumb fuzzer. We have also noticed that the maximum number of basic blocks from the sliced path is covered during the first few generations and they tend to remain constant for the rest of the run.

6.2 *pigz-2.3.4*

gzip is another popular data compression program with a superior compression ratio. pigz is the parallel implementation of gzip that makes use of multiple processors and multiple cores when compressing data. pigz also takes file(s) as input and outputs a .gz compressed file and uses pthreads for multithreading, thus satisfying the conditions of a right candidate to evaluate ConFuzz.

Experimentation

Pigz has twelve files that have to be converted into bitcode files and combine them all together with bitcode file of printID to form a single bitcode file. This final bitcode file is passed to our custom IR pass for instrumentation and the output bitcode file is converted to an executable. There were 466 basic blocks instrumented within the various sliced paths from the pthread function call sites to the main function and 2731 basic blocks outside the sliced paths.

In this case also, we have considered seed inputs of comparatively large size, but didn't require as large as the inputs we used for pbzip2. We started with 5 inputs of sizes from 1 mb and varied up to 5 mb consisting of image file, pdf, text file, and an avi file. We have kept the ConFuzz configuration constant at 1000 generation with 100 population size. We did not pass any parameters while executing the binary, as pigz will make use of the number of CPUs available by default.

Results

ConFuzz ran for more than 12 h. The ConFuzz log file reports a coverage of 21% of the instrumented basic blocks within the sliced path and around 9% of basic blocks outside the sliced path. ConFuzz could detect 44 unique dataraces and generated more than 5000 TSAN warning files.

When we compared the results of ConFuzz with our own dumb fuzzer using the same inputs, it was found that the dumb fuzzer also could detect 42 unique dataraces. The number of basic blocks covered by the dumb fuzzer is also similar to the ConFuzz. But the difference between the ConFuzz and the dumb fuzzer can be found in the time taken to find these dataraces. ConFuzz's efficiency was in finding the first datarace on the 19th second from the start of the fuzzing and the second datarace on the 20th second, while the dumb fuzzer could detect the first datarace only on the 43rd second. The ConFuzz log shows more information about the time taken to detect the concurrency issues by ConFuzz as well as the dumb fuzzer. Since the logic of the dumb fuzzer is similar to ConFuzz, except for considering the ranking of the inputs used for execution, we assume that this can be the reason for the similarities in the number of concurrency issues detected and the basic blocks coverage over the time.

6.3 pixz-1.0.6

pixz is the parallel, indexing version of xz, which is another lossless compression program. pixz automatically indexes tarballs during compression and uses all the available CPU cores by default. Like the above candidates, the basic operation of pixz also takes an input file and outputs a compressed file in .xz format. It also uses pthread for multithreading, hence making it a right candidate for evaluating ConFuzz.

Experimentation

Since the source code of pixz is a little more complicated than the previous two utilities, we have to pass the parameters to the configure script and use the—flto

Table 1 Result summary

Application	Fuzzed time	ConFuzz	DumbFuzz
pbzip2	13 h	15 in 2.10 min	13 in 6 min
pigz	12+ hours	44 (1st in 20 s)	42 (1st in 43 s)
pixz	12+ hours	0	0

option along with llvm-gold plugin. We use 'plugin-opt save-temps' to create bitcode of various module partitions. The generated bitcode files are used, after the 'make' command is called, to combine it with the bitcode file of printID to form the final single bitcode file. This final bitcode file is then passed to the custom IR pass for instrumentation and the output is converted to an executable. In the source code, 466 basic blocks were instrumented within the sliced path between different *pthread* function call sites to the main function and 2731 basic blocks outside the sliced paths. We have considered seed inputs of various sizes from 200 kb to 500 mb and same ConFuzz configurations as the previous experiments of 1000 generations with 100 population size. pixz uses the available CPU cores by default and hence it does not have to pass as parameter. We can set the number of blocks to be allocated for the compression queue by setting the −q size parameter and change the size of each compression block by setting the parameter '−f fraction' [15]. We have set the default values (−q 1.3*cores + 2, rounded up and −f 2.0)[1].

Results

We fuzzed pixz with ConFuzz for more than 12 h and the ConFuzz log reported coverage of 41% of basic blocks from the sliced paths and 21.7% of basic blocks from the non-sliced paths. But, unfortunately, ConFuzz could not trigger any TSAN warnings for concurrency issues. We compared the results with our own-implemented dumb fuzzer and as expected, dumb fuzzer also did not trigger any TSAN warnings.

Table 1 summarizes the evaluation results of the ConFuzz fuzzing on the three applications. It showcases the efficiency of ConFuzz on DumbFuzz with respect to the time taken to detect concurrency issues in each application. The overall results also point to the fact that after some time of fuzzing, it stops detecting unique concurrency issues.

7 Discussion

We considered many multi-threaded applications including libav [16], vlc player [17], ffmpeg [18], and ImageMagick [19] and all these applications including the ones we considered for evaluation, threw TSAN warnings for concurrency issues even upon execution with thread sanitizer instrumented. Hence fuzzing was not needed for these to find concurrency issues. That is when we decided to fuzz these applications to find even deeper concurrency issues, which would not be detected in a simple run. But due to the design constraints of ConFuzz, many of these applications could not be instrumented with our custom IR pass, only the above mentioned three applications could be done. It is easier to generate bitcode files of the applications for instrumentation, which do not have any dependencies and have a straightforward way of building the source code. These methods vary from application to application.

One of the challenges we faced while searching for an appropriate application for the evaluation of ConFuzz was that almost all the applications already had concurrency issues and the thread sanitizer threw TSAN warnings due to concurrency issues. So there was no requirement to fuzz the application to find out a new concurrency issue.

The major challenge we encountered in finding a suitable application was to convert all the source code files into respective bitcode formats and then combine all the bitcode files into a single bitcode file, which can then be passed to the custom IR pass for instrumentation. Every application has its own method of building the source code and creating the executable. Also the applications will have dependencies on other libraries, statically and dynamically, which have to be considered while generating bitcode files of the source code. Again, it should also be verified that all the bitcode files generated from the source code files have to be combined into one single bitcode file and then be able to generate the executable from that single bitcode file by linking the respective binaries. We have to consider only those applications which have one single executable generated at the end of building the source code. For example, ImageMagick version 6 generated multiple utility executables and each executable has some common and some specific object files to be combined while generating the executable. This was really difficult to figure out from the source code Makefile and such applications are not suitable for the current design of ConFuzz. Similarly, ImageMagick links with some object files dynamically, which also is a scenario that cannot be accommodated with ConFuzz's current design.

Another roadblock we faced during evaluating the ConFuzz was finding a fuzzer with which we could compare our results. Initially, we considered AFL-fuzz to compare the crashes it could find against the unique concurrency issues ConFuzz could find. But there were many factors, which made that infeasible. First of all AFL-fuzz is more of a general fuzzer and ConFuzz is a specific purpose fuzzer. The crashes that AFL reports may not be related to concurrency issues and requires more analysis of the detailed report to figure out. Also, AFL can throw multiple TSAN warnings from the same address and report as unique crashes, which complicates the comparison. As of now, as per the AFL team, AFL does not support thread sanitizer, even though it supports address sanitizer (ASAN) and memory sanitizer (MSAN). But the AFL source code can be tweaked a little bit to add support for thread sanitizer (TSAN) as well. Even though we made these changes, it was not reliable, as AFL does not report TSAN warnings all the time. Also, AFL is at its best when the seed input size is less than 1 KB and it does not run if we add seed inputs of bigger size. This can also be tweaked in the AFL's config file to support bigger sized inputs, but the AFL's efficiency could be compromised. Sometimes AFL does not run with bigger seed inputs and simply crashes. Hence we decided to simulate the basic fuzzing technique of AFL in our own implementation of the dumb fuzzer and compare the results with those of the ConFuzz.

8 Related Works

We reviewed some of the existing works done and its advantages and disadvantages. We will discuss these and how our design and implementation differ from them.

8.1 Concurrency Bug Detection

With the increase in demand for speed and efficiency of the computing systems in this fast-growing world, parallel programming and multithreading have become prevalent. With these arise the problem of concurrency (shared memory accesses without proper synchronization among threads) and such programs can lead to very severe consequences like memory corruption, wrong outputs and program crashes [4]. Worse, a prior study [6] showed that many real-world concurrency bugs can lead to concurrency attacks: once a concurrency bug is triggered, attackers can leverage the memory corrupted by this bug to construct various violations, including privilege escalations [20, 21], malicious code injections [22], and bypassing security authentications [23–25].

The detection of these bugs in concurrency programs is very difficult as they are non-deterministic in nature. There were many approaches in the recent past towards this direction of detecting concurrency bugs [26–35], their diagnosis [36–40] and correction [41–44]. Some of these approaches are targeted towards a specific type of application like .Net based [27] or JAVA based [45] and approaches like that of Eraser [26], RacerX [28], RaceMob [34] and even Helgrind [46], the race detector of Valgrind, uses lockset algorithm to record locks and analyze the programs based on these locksets. ConMem [32] targets only those bugs which crashes the program. Unlike these, there are other approaches that make use of hardware to detect concurrency bugs [47, 48]. Most of the other approaches tackle the problem of non-determinism of concurrency bugs by changing the thread schedulers (Eg. NeedlePoint [35] and Concuerror [49]). Since the concurrency bugs are caused due to inappropriate thread interleavings, manually controlling the schedulers was obviously a solution to reproduce them. The backward approach of ConSeq [31] in the bugs life cycle is similar to that of ConFuzzs except that ConSeq deals with binaries and not source code. ConSeq finds the error sites, like an assert call, within the binary and tries to connect critical memory read instructions to these error sites. But in the end, ConSeq detects alternative interleavings from a correct run of the program and then tries out these suspicious interleavings by perturbing the programs re-execution. But these approaches ignore the possibility of an external input triggering a concurrency bug.

8.2 Fuzzing

Fuzzing has been used to test softwares from 1990 [50] and since then there were many methods discovered to fuzz applications. Even though it started as a vulnerability-testing tool, it has become a common method for testing all kinds of bugs in various applications. It started with randomly generating inputs and mutating valid inputs to create new ones. This is known as Traditional fuzzing. Traditional fuzzing is very simple and easy to implement but may generate more number of invalid inputs than the number of inputs of interest and the huge set of inputs generated may trigger the same bug. Blackbox fuzzers like Spike [51] and FileFuzz [52] uses such methods and is good only to find shallow bugs and have less probability of penetrating deep into the application. Whitebox fuzzers like BuzzFuzz [53] tries to overcome the drawbacks of blackbox fuzzers by accessing the internal working and design of the applications. Methods like dynamic taint analysis at the source code level are used for this purpose. Greybox fuzzing is another classification of fuzzers that uses different dynamic and static analysis with no access to source code to understand the working and design of the application to be tested. Fuzzers like Mayhem [54] employs techniques like symbolic/concolic execution to analyze the application. These techniques help in generating valid inputs that can penetrate deeper into the applications' control flow to trigger bugs. Apart from these general classifications of fuzzers, there is another way of fuzzing application known as Directed Fuzzing. Dowser [55] uses this approach with the help of static analysis to detect the regions with probable buffer overflows and the inputs are tainted and traced to find regions that process the inputs, which are considered as probable regions that can lead to a crash. AFL-Fuzz [56] is also instrumentation guided fuzzer that uses the compiler to instrument the code being fuzzed and mutate the inputs to explore all the code paths. It makes use of a feedback loop to retain inputs that contribute to a new execution path and mutate other inputs for better coverage. Symbolic execution is used to get information on what offsets to mutate and with what value [57]. But it does not scale well with large projects. VUzzer [8] overcomes these pitfalls by using the control flow features to prioritize interesting execution paths and the dataflow features to decide which part of inputs to be mutated to explore new execution paths. This makes VUzzer fast, scalable and can penetrate deep into the program code to detect bugs.

8.3 Other Techniques

There were enormous researches on defense techniques for sequential programs like taint tracking, anomaly detection and static analysis. But most of these techniques or tools can be weakened or even bypassed [6] with the implementation of multithreading. There are already many static vulnerability detection techniques [58–63].

Some of them target general programs and some others focus on specific behaviors in specific programs for scalability and accuracy. The vulnerability detection approaches for general programs are not sufficient to cope with concurrency bugs and the concurrency attacks that follow [64]. Similarly, the approaches that focus on specific behaviors in specific programs check web application logics, android applications, cross-checking security APIs, verifying the Linux security module, etc. These approaches are complimentary to ConFuzz and can be further used in improving the efficiency of detecting concurrency bugs.

9 Conclusion

In this paper, we try to explore the possibilities of detecting concurrency issues by fuzzing the application. We use the static analysis of the source code of the application along with the Thread sanitizer present in the LLVM compiler framework. The paper looks at a design to use fuzzing techniques in detecting the vulnerable memory locations with the possibility of concurrency issues. The design aims to leverage the information from the source code analysis about the executions paths and the thread related function call sites to rank the inputs and use the high ranked inputs to trigger thread sanitizer warnings.

We gained insights into the design of thread sanitizer and its working during this research. We designed a custom LLVM pass to find all the execution paths from various thread function call sites to the main function and rank the basic blocks between them. This pass also helped in calculating the total rank of input after its execution. This information is used in fuzzing to prioritize inputs for the next generations. The unique TSAN warnings are calculated by reading the memory locations in the TSAN warning messages.

There were numerous implementation difficulties during the research into the approach presented, with regards to source-level instrumentation. It was not easy to convert all the source code files into bitcodes especially when the application was really huge and there were dependencies with static and dynamic binaries. It was also difficult to find suitable applications to fuzz using this approach, as the application should be multi-threaded and small enough as to easily instrument. The input sizes were also a constraint as most of the parallel processing in these multi-threaded applications are triggered only when the input size is above a threshold and it is not suitable to fuzz with large sized inputs.

We evaluated ConFuzz on three different open source applications, namely pbzip2, pigz, and pixz. All three applications were fuzzed with around 1000 generation for more than 12 h. The output showed that the fuzzer reached its maximum coverage after numerous generations and then the coverage of the basic blocks within the sliced paths remained almost constant. It recorded many TSAN warnings during these runs and unique TSAN warnings were identified. The inputs that triggered unique warnings were separately stored in a different folder, which can be used for further analysis. The results from the evaluation show a potential in the approach if we can fine-tune it a little more and the discussed implementation constraints worked out.

10 Future Works

ConFuzz fuzzes the multi-threaded applications to detect possible memory locations where concurrency bugs can happen. Even though it operates under a number of constraints, there holds tremendous scope for improvement at the design as well as implementation levels. We can use DFSan [65] available in the LLVM framework to compute the taint dependency of the memory accesses made by the target calls, detected during the static analysis of the source code, and mutate those bytes to check if they trigger TSAN warnings. As per our current design, we are monitoring only a few target calls (thread function call sites). Instead, we can design the current system to monitor the memory accesses made between a pair of calls, say lock/unlock function calls. A static analysis can also be performed to find other basic blocks that access these memory locations. From the newfound set of basic blocks, we can do similar slicing and weigh the sliced paths, so that we can use them in the fuzzing strategy to rank the inputs. We can also use DFSan to compute the taint dependency of the calculated slices and mutate the inputs at those offsets to drive the input generation. We can also make use of the OWL [64] application to verify if we can actually carry out a concurrency attack on the detected vulnerable memory locations.

The access to source code provides us with a number of possibilities to improve on and the above-mentioned ones are just a few of them. The many existing as well as developing tools within the LLVM framework can also be put to use to further explore the possibilities of improving ConFuzz.

References

1. Llvm. https://llvm.org/
2. Thread sanitizer. url. http://clang.llvm.org/docs/threadsanitizer.html
3. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: pldi (2007)
4. Seo, E., Zhou, Y., Lu, S., Park, S.: Learning from mistakes a comprehensive study on real world concurrency bug characteristics. ACM Trans. Comput. Syst. **2**(4), 277–288 (2008). ISSN 0734-2071
5. Common vulnerabilities and exposures database. http://cvedetails.com
6. Stolfo, S., Sethumadhavan, S., Yang, J., Cui, A.: Concurrency attacks. In: Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR 12) (2012)
7. Fredriksen, L., Miller, B.P., So, B.: An empirical study of the reliability of unix utilities. Commun. ACM **33**(12), 3244 (1990)
8. Kumar, A., Cojocar, L., Giuffrida, C., Rawat, S., Jain, V., Bos, H.: Vuzzer: application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2017)

9. Apple developer page for llvm thread sanitizer
10. Clang. http://clang.llvm.org/index.html
11. Pbzip2. http://compression.ca/pbzip2/
12. Pigz. https://zlib.net/pigz/
13. Pixz. https://github.com/vasi/pixz
14. Bzip. http://www.bzip.org/
15. Pixz man page. https://www.mankier.com/1/pixz
16. Libavi. https://libav.org/
17. Vlc. https://www.videolan.org/vlc/download-sources.html
18. Ffmpeg. https://www.ffmpeg.org/download.html
19. Imagemagick. https://www.imagemagick.org/script/download.php
20. Linux kernel bug on uselib(). http://osvdb.org/show/osvdb/12791
21. Mysql bug 24988. https://bugs.mysql.com/bug.php?id=24988
22. Msie javaprxy.dll com object exploit. http:// www.exploit-db.com/exploits/1079/
23. Cve-2010-0923. http://www.cvedetails.com/cve/cve-2010-0923
24. Cve-2008-0034. http://www.cvedetails.com/cve/cve-2008-0034/
25. Cve-2010-1754. http://www.cvedetails.com/cve/cve-2010-1754/
26. Nelson, G., Sobalvarro, P., Anderson, T., Savage, S., Burrows, M.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997)
27. Chen, W., Yu, Y., Rodeheffer, T.: Racetrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 05), pp. 221–234 (2005)
28. Ashcraft, K., Engler, D.: Racerx: effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 03), pp. 237–252 (2003)
29. Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Lu, S., Park, S., Zhou, Y.: Muvi: automatically inferring multivariable access correlations and detecting related semantic and concurrency bugs. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 07), pp. 103–116 (2007)
30. Qin, F., Lu, S., Tucek, J., Zhou, Y.: Avio: detecting atomicity violations via access interleaving invariants. In: Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 06), pp. 37–48 (2006)
31. Olichandran, R., Scherpelz, J., Jin, G., Lu, S., Zhang, W., Lim, J., Reps, T.: Conseq: detecting concurrency bugs through sequential errors. In: Sixteenth International Conference on Architecture Support for Program- ming Languages and Operating Systems (ASPLOS 11), pp. 251–264 (2011)
32. Sun, C., Zhang, W., Lu, S.: Conmem: detecting severe concurrency bugs through an effect-oriented approach. In: Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 10), pp. 179–192 (2010)
33. Chen, P.M., Flinn, J., Wester, B., Devecsery, D., Narayanasamy, S.: Parallelizing data race detection. In: Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 13), pp. 27–38 (2013)
34. Zamfir, C., Kasikci. B., Candea, G.: Racemob: crowdsourced data race detection. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP 13) (2013)
35. Martin, M.M.K., Nagarakatte, S., Burckhardt, S., Musuvathi, M.: Multicore acceleration of priority-based schedulers for concurrency bug detection. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI ’12) (2012)
36. Lu, S., Park, S., Zhou, Y.: Ctrigger: exposing atomicity violation bugs from their hiding places. In: Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 09), pp. 25–36 (2009)
37. Park, C.-S., Sen K.: Randomized active atomicity violation detection in concurrent programs. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIG- SOFT 08/FSE-16), pp. 135–145 (2008)

38. Sen, K.: Race directed random testing of concurrent programs. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), pp. 11–21 (2008)
39. Pereira, C., Pokam, G., Kasikci, B., Schubert, B., Candea, G.: Failure sketching: a technique for automated root cause diagnosis of inproduction failures. In: Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP 15) (2015)
40. Chow, M., Attariyan, M., Flinn, J.: X-ray: automat- ing root-cause diagnosis of performance anomalies in production software. In: OSDI (2012)
41. Deng, D., Liblit, B., Jin, G., Zhang, W., Lu, S.: Automated concurrency bug fixing. In: Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 221–236 (2012)
42. Cristian, Z., Jula, H., Tralamazza, D., George, C.: Deadlock immunity: enabling systems to defend against deadlocks. In: Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI 08), pp. 295–308 (2008)
43. Kudlur, M., Lafortune, S., Wang, Y., Kelly, T., Mahlke, S.: Gadara: dynamic deadlock avoidance for multithreaded programs. In: Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI 08), pp. 281–294 (2008)
44. Cui, H., Wu, J., Yang, J.: Bypassing races in live applications with execution filters. In: Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI 10) (2010)
45. Whaley, J., Naik, M., Aiken, A.: Effective static race detection for java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 06), pp. 308–319 (2006)
46. Valgrind. http://valgrind.org/docs/manual/hg-manual.html
47. Zhang, Weihua, Yu, Shiqiang, Wang, Haojun, Dai, Zhuofang, Chen, Haibo: Hardware support for concurrent detection of multiple concurrency bugs on fused cpu-gpu architectures. IEEE Trans. Comput. **65**, 3083–3095 (2016)
48. Alam, M.U., Begam, R., Rahman, S., Muzahid, A.: Concurrency bug detection and avoidance through continuous learning of invariants using neural networks in hardware (2013)
49. Gotovos, A., Christakis, M., Sagonas, K.: Systematic testing for detecting concurrency errors in erlang programs. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST '13), pp. 154–163 (2013)
50. Fredriksen, L., Miller, B.P., So, B.: An empirical study of the reliability of unix utilities. Commun. ACM **33**(12), 32–44 (1990)
51. Aitel, D.: An introduction to spike, the fuzzer creation kit. (presentation slides) (2002)
52. Sutton, M., Greene, A.: The art of file format fuzzing. In: Blackhat USA Conference (2005)
53. Leek, T., Ganesh, V., Rinard, M.: Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering, pp. 474–484. IEEE Computer Society (2009)
54. Rebert, A., Cha, S.K., Avgerinos, T., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 380–394 (2012)
55. Neugschwandtner, M., Haller, I., Slowinska, A., Bos, H.: Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In: USENIX Security Symposium, pp. 49–64 (2013)
56. American fuzzy loop (afl-fuzz). https://github.com/rc0r/afl-fuzz
57. Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Stephens, N., Grosen, J., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: NDSS, vol. 16, pp. 1–16 (2016)
58. Livshits, V.B., Lam, M.S.: Finding security errors in java programs with static analysis. In: Proceedings of the 14th Usenix Security Symposium, pp. 271–286 (2005)
59. Arp, D., Yamaguchi, F., Golde, N., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP 14), pp. 590–604 (2014)
60. Kruegel, C., Felmetsger, V., Cavedon, L., Vigna, G.: Toward automated detection of logic vulnerabilities in web applications. In: Proceedings of the 19th USENIX Conference on Security (USENIX Security 10), pp. 1010 (2010)

61. Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., Arzt, S., Rasthofer, S., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI14), pp. 259–269 (2014)
62. McKinley, K.S., Srivastava, V., Bond, M.D., Shmatikov, V.: A security policy oracle: detecting security holes using multiple api implementations. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 11), pp. 343–354 (2011)
63. Edwards, A., Zhang, X., Jaeger, T.: Using cqual for static analysis of authorization hook placement. In: Proceedings of the 11th USENIX Security Symposium, page p. 33–48 (2002)
64. Zhao, J., Ning, Y., Cui, H., Yang, J., Gu, R., Gan, B.: Understanding and Detecting Concurrency Attacks
65. Data flow sanitizer. http://clang.llvm.org/docs/dataflowsanitizer.html