A Heuristic Framework to Detect Concurrency Vulnerabilities

Changming Liu, Deqing Zou*
Peng Luo
Service Comp. Tech. and System Lab,
Cluster and Grid Computing Lab,
School of Computer Sci. and Tech.,
Huazhong Univ. of Sci. and Tech.
Wuhan, China

Bin B. Zhu
Microsoft Research Asia
Beijing, China
binzhu@microsoft.com

Hai Jin
Service Comp. Tech. and System Lab,
Cluster and Grid Computing Lab,
School of Computer Sci. and Tech.,
Huazhong Univ. of Sci. and Tech.
Wuhan, China

ABSTRACT

With a growing demand of concurrent software to exploit multi-core hardware capability, concurrency vulnerabilities have become an inevitable threat to the security of today's IT industry. Existing concurrent program detection schemes focus mainly on detecting concurrency errors such as data races, atomicity violation, etc., with little attention paid to detect concurrency vulnerabilities that may be exploited to infringe security. In this paper, we propose a heuristic framework that combines both static analysis and fuzz testing to detect targeted concurrency vulnerabilities such as concurrency buffer overflow, double free, and use-after-free. The static analysis locates sensitive concurrent operations in a concurrent program, categorizes each finding into a potential type of concurrency vulnerability, and determines the execution order of the sensitive operations in each finding that would trigger the suspected concurrency vulnerability. The results are then plugged into the fuzzer with the execution order fixed by the static analysis in order to trigger the suspected concurrency vulnerabilities.

In order to introduce more variance which increases possibility that the concurrency errors can be triggered, we also propose manipulation of thread scheduling priority to enable a fuzzer such as AFL to effectively explore thread interleavings in testing a concurrent program. To the best of our knowledge, this is the first fuzzer that is capable of effectively exploring concurrency errors.

In evaluating the proposed heuristic framework with a benchmark suit of six real-world concurrent C programs, the framework detected two concurrency vulnerabilities for the proposed concurrency vulnerability detection, both being confirmed to be true positives, and produced three new crashes for the proposed interleaving exploring fuzzer that existing fuzzers could not produce. These results demonstrate the power and effectiveness of the proposed heuristic framework in detecting concurrency errors and vulnerabilities.

*Corresponding author: Deqing Zou (deqingzou@hust.edu.cn). This work was supported by the National 973 Fundamental Basic Research Program under grant No. 2014CB340600.

ACSAC '18, December 3–7, 2018, San Juan, PR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274718

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

Concurrency Vulnerabilities, Fuzzing Test, Thread Schedule.

ACM Reference Format:
Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. 2018. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274694.3274718

1 INTRODUCTION

Concurrent programs can provide significantly more computing power than sequential programs and have been applied in many demanding applications, e.g. cloud services. However, concurrent programs are prone to concurrency vulnerabilities that may cause severe consequences, e.g. dirty copy on write[31], a well-known concurrency vulnerability found in the Linux kernel, and attacks specifically targeting at concurrent programs to disrupt confidentiality, integrity or availability of the system [33]. It is a great challenge to detect concurrency bugs and vulnerabilities since there are too many interleavings in a typical concurrent program.

```
          thread 1
1030      if (type == rr->type) /* SSL3_RT_APPLICATION_DATA or SSL3_RT_HANDSHAKE */
          {
          [...]
1050              if (!peek)
1051              {
1052                      rr->length-=n;
1053                      rr->off+=n;
1054                      if (rr->length == 0)
1055                      {
1056                              s->rstate=SSL_ST_READ_HEADER;
1057                              rr->off=0;
1058                              if (s->mode & SSL_MODE_RELEASE_BUFFERS)
1059                                      ssl3_release_read_buffer(s);
1060                      }
1061              }
1062              return(n);
1063      }


        thread 2
124 int ssl3_read_n(SSL *s, int n, int max, int extend)
125 {
          [...]
140       rb    = &(s->s3->rbuf);
141       if (rb->buf == NULL)
142               if (!ssl3_setup_read_buffer(s))
143                       return -1;
```

Figure 1: CVE-2010-5298 in s3_pkt.c of OpenSSL

Fig. 1 shows a real-world concurrency use-after-free vulnerability found in s3_pkt.c of OpenSSL. This vulnerability is triggered at line 143 where thread 2 sets up a buffer *s* for later usage, and before the buffer is used, another thread, thread 1, releases this same piece of memory *s* at line 1059. This would allow remote attackers to inject data across sessions or cause denial of service [17]. The patch to this vulnerability is simply to add a condition inside the if-condition clause at line 1058 to check if there is still unprocessed data left in *s* before releasing it at line 1059 [18].

Detecting concurrency errors has been extensively studied, mainly focusing on detecting data races, i.e. multiple simultaneous accesses to shared memory with at least one write. Both static and dynamic approaches have been used. However, methods aiming at detecting data races in concurrent programs are generally inadequate in detecting real-world concurrency vulnerabilities that can happen even when a concurrent program is race-free. For example, in the case shown in Fig. 1, making the two threads' accesses to the shared buffer *s*, i.e. lines 142 and 1059, race-free would not prevent the aforementioned vulnerability from happening.

The concurrency vulnerability shown in Fig. 1 is similar to the order violation described in [21, 36], wherein multiple concurrent accesses, protected by a lock respectively, to shared memory can cause crashes of the program. If the free operation at line 1059 is executed after finishing using the buffer, the vulnerability will never occur. On the other hand, if their execution order is reversed, the vulnerability will occur. In a concurrent program, the execution order of threads may be uncontrollable, and a wrong execution order may occur, leading to a vulnerability that may be exploited to inject data across sessions or cause denial of service. Existing methods [21, 36] of detecting order violation are all based on monitoring memory accesses, e.g. read/write, and the order violations they can detect are likely to cause concurrency errors instead of concurrency vulnerabilities that this paper focuses on. This limitation has been lifted in our approach.

In this paper, we propose a heuristic framework that combines both static analysis and fuzz testing to detect concurrency vulnerabilities, particularly concurrency buffer overflows, double-free, the two most common concurrency vulnerabilities as reported in the National Vulnerability Database [16], and the aforementioned concurrency use-after-free. A concurrency buffer overflow typically occurs when two threads access shared memory and one of them modifies the shared memory, possibly with maliciously crafted content, before the other passes the shared memory to a *memcpy*-like function. A real-world concurrency buffer overflow example will be presented in Section 3. Concurrency double-free is intuitive: two concurrent free operations on the same memory, and this can result in undefined behaviors. In addition, we also propose an interleaving exploring strategy in the heuristic framework to enable fuzz testing to explore thread interleavings effectively so that it can detect concurrency errors in concurrent programs more efficiently.

Our framework consists of the following three main techniques we have developed:

- **Static Analysis for Concurrent Operations**. In this paper, we use static analysis to detect sensitive concurrent operations that are likely to lead to concurrency vulnerabilities. More specifically, we collect a set of sensitive concurrent operations and distill distinct operation patterns for each type of concurrency vulnerability by studying the characteristics of real-world concurrency vulnerabilities, and leverage static analysis to locate sensitive concurrent operations, whether protected by mutex or not, in a program. We compare each finding against the operation patterns of each type of concurrency vulnerability, and categorize it to a certain type of vulnerability, e.g. a concurrency buffer overflow, double-free, or use-after-free that we have chosen as an example to study the proposed heuristic framework in this paper. We should point out that our framework can be readily extended to detect other types of concurrency vulnerabilities.
- **Exploring Thread Interleavings in Fuzz Testing**. Fuzz testing is criticized for being inadequate to detect concurrency errors. One major reason is that, although very capable of exploring new branches at conditional jumps, current state-of-the-art fuzzers such as AFL [13] are unaware of thread scheduling and thus cannot explore enormous interleavings as capable as they are in exploring path changes. To enable a fuzzer to explore thread interleavings as effectively as it explores path changes, we develop a thread-aware fuzzer that randomizes priorities of forked threads to explore thread interleavings to cover as many interleavings as possible, i.e., in each iteration of fuzz testing, we select one or more threads to manipulate their priorities towards untested interleavings. This ensures that more interleavings are likely to be explored with increasing iterations of fuzz testing. We have found several new crashes using this approach. To the best of our knowledge, we are the first to design a fuzzer to effectively explore thread interleavings to detect concurrency errors/vulnerabilities.
- **Targeting Scheduling for Sensitive Concurrent Operations**. Like order violation mentioned in [21, 36], the execution order of concurrent operations is typically critical in triggering concurrency vulnerabilities. For example, the vulnerability shown in Fig. 1 can be triggered only if the free operation is called before the shared memory is used. Unlike order violation detection schemes in [21, 36] that detect order violation patterns in run time, we first apply static analysis to locate sensitive concurrent operations and identify the potential concurrency vulnerabilities they may lead to as well as the specific execution order to trigger each potential vulnerability. The information enables us to insert priority adjusting code to force the sensitive concurrent operations of a potential concurrency vulnerability to be executed in the specific order in fuzz testing so that the potential vulnerability has a high chance to be triggered.

This paper has the following major contributions by proposing:

- A novel approach to effectively detect concurrency vulnerabilities: locating sensitive concurrent operations that may lead to a potential concurrency vulnerability and forcing a specific execution order of threads to trigger the potential concurrency vulnerability in fuzz testing. By studying the characteristics of some common real-world concurrency vulnerabilities, we have found that each type of concurrency vulnerability has a few sensitive concurrent operations and distinct operation patterns. This allows us to apply static analysis to locate sensitive concurrent operations that potentially lead to a concurrency vulnerability and to use the operation patterns to identify the potential type of concurrency vulnerability along with the specific execution order to trigger it. This enhances the opportunity to trigger the concurrency vulnerability in fuzz testing by adjusting thread's priorities to force the program to be executed in the designated execution order.

- An effective method to explore thread interleavings of concurrent programs in fuzz testing: randomizing priorities of threads to explore as many interleavings as possible. This can be achieved by injecting code to adjust threads' priorities, forcing threads to sleep for a random or specific time, etc., towards untested interleavings. This empowers a fuzzer to explore effectively not only code paths but also concurrent interleavings and can significantly improve the effectiveness of fuzz testing on testing concurrent programs.

This paper is organized as follows. We present the related work in Section 2 and study real-world examples of concurrency vulnerabilities in Section 3. Our static analysis is described in Section 4, and the fuzzing strategies for concurrent programs are described in Section 5. Our implementation of the proposed heuristic framework is described in Section 6, and the evaluation results are presented in Section 7. Limitations of the current implementation of the heuristic framework and the future work are described in Section 8. The paper concludes with Section 9.

2 RELATED WORK

2.1 Static Analysis to Detect Concurrency Problems

Many static approaches have been proposed to handle concurrency problems, such as [23, 28, 29]. Context-sensitive correlation analysis is proposed in [23] to check if every memory location in a program is consistently correlated with a lock, and its detection is proved to be accurate. Aiming at the same, the method in [28] employs a concept of *relative lockset* to gain significant scalability. As we mentioned before, existing static analysis focuses mainly on data races, which are quite different from concurrency vulnerabilities we focus on. A static method specifically for double-fetch situations is proposed in [29] which designates certain static patterns for double-fetch situations and detects double-fetch situations by matching these patterns. This method is scalable and can find many double-fetch vulnerabilities, yet it is hard to extend to detect other types of concurrency vulnerabilities.

2.2 Concurrency Error Detection

Existing concurrency error detection techniques can be classified into two categories: heuristic techniques and test techniques. Heuristic techniques [19–21] detect concurrency errors based on error patterns or characteristics. These methods construct heuristic rules and statically scan the whole program to find violation of these rules. These heuristic rules may not catch all running situations, especially for concurrent programs. To tackle this problem, dynamic analysis has been developed. For example, CTrigger [19] uses a dynamic method to detect atomicity violations by analyzing interleaving characteristics of synchronization events in concurrent programs.

Test techniques detect concurrency errors by running target programs with system scheduling or designated tests to trigger concurrency errors. They typically aim at covering as many interleavings as possible by generating either tests [7, 26, 27] or schedules [3, 34] to detect concurrency errors. Compared with heuristic techniques, a test technique usually suffers from low efficiency and thus needs significant amount of time to test. These concurrency error detectors focus mainly on access interleavings of shared memory, with expensive analysis and complex test or scheduler generation, and are often used for unit tests instead of system tests due to their complexity. As a comparison, our interleaving exploring method for fuzz testing applies a lightweight method to adjust threads' priorities to explore thread interleavings rather than memory access interleavings, and is thus scalable to test much larger concurrent programs.

Since concurrency vulnerabilities are caused by concurrency errors, a natural thought would be to apply concurrency error detectors to detect concurrency vulnerabilities. This approach does not work well in general for detecting concurrency vulnerabilities since these concurrency error detectors focus mainly on detecting three types of concurrency errors: data races, atomicity violations, and order violations. As we mentioned in Section 1, concurrency vulnerabilities may occur even when all the types of concurrency errors these detectors focus on have been cleared off. Triggering a concurrency vulnerability normally needs to meet two requirements: a specific input and a specific scheduling. These concurrency error detectors aim at exploring bug-triggering interleavings and typically will not meet the required input and the required scheduling simultaneously to trigger a concurrency vulnerability.

Our method to detect concurrency vulnerabilities borrows some ideas from the order violation detection proposed in [21, 36] and the active testing proposed in [4, 11, 35]. The former focuses on detecting wrong execution orders that lead to concurrency errors in a concurrent program. The latter targets at specific bug types such as data races by applying a static detector to predict buggy thread interleavings and then executing a suspected buggy thread interleaving in a real execution to try to trigger the bug. These methods focus on detecting concurrency errors rather than concurrency vulnerabilities and, as just mentioned, unlikely effective in detecting concurrency vulnerabilities. We have extended these ideas to detect concurrency vulnerabilities.

An interesting yet loosely related work [37] has been proposed recently to detect concurrency attacks by relying on an attack inference model that models behaviors of concurrency attacks in the three stages of their life-cycle in launching an attack: a concurrency bug is first triggered to corrupt shared memory, then the corrupted memory propagates across functions and threads, which may go across memory boundaries (e.g., buffer overflows) during propagation, and finally the corrupted memory flows to vulnerable sites (e.g., eval() and setuid()) to complete an attack. The method has produced some sound results: it has detected 5 new concurrency attacks and eliminated 94.1% of the reports generated by existing concurrency bug detectors as false positive.

2.3 Logic-Based Methods

A logic-based approach applies model-checking to detect concurrency errors. It adopts a constraint solver to check if there is an error. Logic-based methods such as [9, 24] can produce sound reproducible results, but they have to apply methods such as approximation, pruning etc. to deal with path explosion and heavy workload in constraint solving, and thus are not scalable to a large amount of interleavings. As a comparison, our proposed scheme is light-weighted and thus is scalable to a large amount of interleavings.

2.4 Fuzz Testing

Fuzz testing has been widely used to detect software vulnerabilities over the past twenty-some years since Miller et al. [15] introduced it to test the robustness of UNIX utilities in 1990. Due to its effectiveness in detecting software bugs and vulnerabilities, fuzz testing has gained popularity since its introduction. The basic idea in fuzz testing is to feed test programs with many mutated or random inputs to produce irregular behaviors or to trigger vulnerabilities. Fuzz testing can be divided into three types in general: black-box fuzzing, white-box fuzzing, and gray-box fuzzing.

Black-box fuzzing requires neither knowing internal logics of tested programs nor source code. As a result, many generated test inputs may be uninteresting or cannot explore any deep path in program semantics. Many methods [25, 30] have been proposed to generate effective test inputs and explore deeper paths with the aid of domain knowledge. To compare effectiveness of different black-box fuzzing methods, Maverick et al. [32] proposed an analytic framework to evaluate existing black-box fuzzing algorithms by using a mathematic mutation model.

White-box fuzzing requires complete knowledge of the source code and behaviors of targeted programs. Generally, it applies heavy analysis techniques, such as dynamic symbolic execution, to generate test inputs and explore as many paths as possible. It is very efficient at exploring new program paths in order to trigger more bugs and vulnerabilities. A great challenge white-box fuzzing faces is scalability: it is hard to scale to large programs due to path explosion [5]. An example of white-box fuzzing methods is presented in [10].

AFL [13] is a popular gray-box fuzzer to detect software bugs. It instruments a targeted program at every conditional jump instruction in compiling time, and then it keeps mutating an input and running the program in order to explore new branches to find more bugs. AFL is well-known to be explore sophisticated programs in a shallow manner. Recently proposed gray-box fuzzers [6, 14] have focused on addressing this low code coverage problem.

All existing fuzz testing methods have focused on exploring more paths. They are unaware of thread scheduling and thus cannot explore enormous concurrent interleavings as capable as they explore path changes. As a result, they are ineffective in detecting concurrency errors and vulnerabilities.

3 CASE STUDY OF CONCURRENCY VULNERABILITIES

In this section, we study examples of real-world concurrency vulnerabilities selected from the National Vulnerability Database [16]. The study leads to finding sensitive concurrent operations and distinct operation patterns for each type of concurrency vulnerability we study in this paper. We will use these in our static analysis which is to be described in detail in Section 4.

3.1 Real-World Concurrency Vulnerabilities

We have shown a real-world concurrency use-after-free vulnerability in Section 1. Fig. 2 shows another real-world concurrency vulnerability, a concurrency buffer overflow, which is triggered after computing how many escape characters contained in a NULL-ended string *s* with the for-loop in lines 1921-1925 and its *length* (including the ending NULL) at line 1928. The string and its length are then passed to function *apr_pmemdup*. Meanwhile, if another thread is allowed to modify the same piece of memory to make *s* longer than *length* bytes, execution of line 119 in *apr_pmemdup* will make NULL-ended string *res* not contain proper ending NULL. String *res* is returned at line 120 and again at line 1932 for more processing. When the content of the string is subsequently used, such as in a *memcpy*-like function, the content beyond the allocated memory will be included since the proper NULL ending of the NULL-ended string has been overwritten by another thread, resulting in a buffer overflow. This may lead to information leakage or even getting total control over the CPU that happened in the real world [12].

The above concurrency buffer overflow can be a data race problem wherein two threads access string *str* simultaneously and can be prevented by applying a mutex to lock operations from line 1921 to 1933 to prevent other threads from accessing *str* during execution of these lines. However, if a finer lock is applied, such as the calling function and the called function in Fig. 2 being separately locked, i.e., a mutex is used to lock accessing *str* in the calling function, i.e., from line 1921 to line 1929, and the mutex is used to lock accessing *str* (i.e., *m*) in function *apr_pmemdup* to prevent other threads from accessing *str* simultaneously, then the program is race-free, yet the concurrency buffer overflow can still happen when another thread modifies the content of *str* after line 1928 has been executed but before function *apr_pmemdup* starts to execute. There are more real-world concurrency vulnerabilities, such as CVE-2011-0990, CVE-2010-3864, etc. in the National Vulnerability Database [16], that can still occur even when a program is race-free.

3.2 Characteristics of Concurrency Vulnerabilities

Let us study the characteristics of concurrency buffer overflows. A buffer overflow is triggered when the input data exceeds the buffer's boundary and overwrites adjacent memory locations. It usually occurs in memory replication. Fig. 3 shows an example of *for-loop* memory replication. In a concurrent program, *source*, *dest*, or *length* might be modified in another thread after the correct values of these three variables have been determined and before the memory replication process has completed. This may trigger a concurrency buffer overflow. Thus concurrency buffer overflows have the following characteristics:

- Memory replication is required. Memory replication may manifest in several ways: calling memory replication functions such as *memcpy* and *strcpy*, using memory replication

```
APR_DECLARE(void *) apr_pmemdup(apr_pool_t *a, const void *m, apr_size_t n)
{
    void *res;

    if (m == NULL)
        return NULL;
    res = apr_palloc(a, n);
    memcpy(res, m, n);
    return res;
}
...

    /* Compute how many characters need to be escaped */
    s = (const unsigned char *)str;
    for (; *s; ++s) {
        if (TEST_CHAR(*s, T_ESCAPE_LOGITEM)) {
            escapes++;
        }
    }

    /* Compute the length of the input string, including NULL*/
    length = s - (const unsigned char *)str + 1;

    /* Fast path: nothing to escape */
    if (escapes == 0) {
        return apr_pmemdup(p, str, length);
    }
```

Figure 2: An example of concurrency buffer overflow in server/util.c in Apache

```
1: for(i=0; i<length; i++)
2:     dest[i]=source[i];
```

Figure 3: An example of memory replication using for-loop

statements such as the *for-loop* shown in Fig. 3 or a *while-loop*.

- At least one of *source*, *dest*, or *length* is a shared variable and can be modified by other threads.
- The execution order is important to trigger a concurrency buffer overflow: modification by another thread must be executed before the memory replication completes.

Concurrency double-free and concurrency use-after-free can also be characterized in a similar manner, for example, a shared variable that can be accessed concurrently, and there are at least two concurrent free operations on this shared variable for the former or one free operation on the shared variable in one thread and accessing the shared variable in another thread that may occur after the free operation for the latter.

From the above concurrency vulnerabilities we can observe the following common essential requirements to trigger one of these concurrency vulnerabilities:

- **Concurrent Access to Shared Memory**. There must be at least one shared variable that can be concurrently accessed from multiple threads.
- **Sensitive Concurrent Operations on Shared Memory**. Among concurrent accesses to the shared variable, there is at least one sensitive operation that is vital to trigger a concurrency vulnerability. Different concurrency vulnerability has different sensitive operations. For example, sensitive concurrent operations for a concurrency buffer overflow are memory replication and content modification on shared memory; two free operations on shared memory for concurrency double-free; and one free operation and another memory access on shared memory for concurrency use-after-free.
- **Execution in Right Order**. A certain execution order of the sensitive concurrent operations is typically critical in triggering a concurrency vulnerability. For example, the memory modification must occur before (or during) the memory replication for a concurrency buffer overflow, and the free operation must happen before accessing the shared memory for concurrency use-after-free. There is no ordering for concurrency double-free since the two free operations play an identical operation.

The above sensitive concurrent operations, operation patterns, and execution orders to trigger concurrency vulnerabilities will be used in our heuristic framework to detect concurrency vulnerabilities in concurrent programs, as described in detail in the subsequent two sections.

4 STATIC ANALYSIS

Our heuristic framework consists of static analysis and thread-aware fuzzing. The static analysis is described in this section, while the thread-aware fuzzing is described in the next section.

In our framework, static analysis aims at locating sensitive concurrent operations and categorizing each finding into a potential type of concurrency vulnerability so that the thread-aware fuzzing would adjust threads' running priorities to enhance the chance to trigger the potential concurrency vulnerability in fuzz testing.

Our static analysis consists of four steps: discovering shared memory, marking sensitive operations, merging data flows, and categorizing potential concurrency vulnerability type. Fig. 4 shows the whole procedure of state analysis for concurrency double-free at line 4 and line 9 in the code shown on the left-most side of the figure.

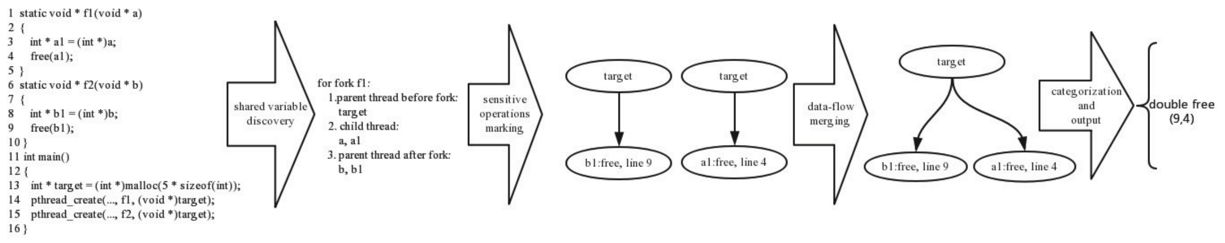
4.1 Shared Memory Discovery

As described in Section 3.2, shared variables that can be concurrently accessed are essential in triggering a concurrency vulnerability. The first step focuses on finding shared memory that is passed as a pointer when forking a new thread: whenever a new thread is forked, we record the pointers that are passed through *pthread_create* and potentially point to shared memory that can be concurrently accessed.

Additionally, global variable access is another major source of concurrent access. We handle this by recording all pointers that point to a global variable in following three different places:

(1) A parent thread before a fork;
(2) A child thread;
(3) A parent thread after a fork.

Note that pointers that are passed through assignments such as *p2 = p1*; *p3 = p2*; ..., are merely for data propagation rather than genuine modification. These pointers point to the same memory and thus should be treated as if an identical pointer. We apply a filter on pointers to identify redundant pointers that essentially point to the same piece of memory.



```
1  static void * f1(void * a)
2  {
3     int * a1 = (int *)a;
4     free(a1);
5  }
6  static void * f2(void * b)
7  {
8     int * b1 = (int *)b;
9     free(b1);
10 }
11 int main()
12 {
13    int * target = (int *)malloc(5 * sizeof(int));
14    pthread_create(..., f1, (void *)target);
15    pthread_create(..., f2, (void *)target);
16 }
```

for fork f1:
1.parent thread before fork: target
2. child thread: a, a1
3. parent thread after fork: b, b1

Figure 4: The whole procedure of static analysis for concurrency double-free vulnerability

Table 1: Sensitive operations and their meaning for a shared variable

Operation	The shared variable is
memcpy	passed to a *memcpy()*-like function.
read	normally read.
memset	passed to a *memset()*-like function.
free	passed to a *free()*-like function.
null	assigned to NULL.
set	set as left operand of an assign operation.

4.2 Sensitive Operation Marking

After locating shared variables in a concurrent program, we examine operations on these shared variables to collect all sensitive concurrent operations on shared memory in a concurrent program. More specifically, we first construct a data-flow graph with the following connections among a parent thread and its child thread in a fork operation:

- A connection from the parent thread before the fork to its child thread;
- A connection from the parent thread before the fork to the parent thread after the fork.

Fig. 5 shows the above connections in constructing a data-flow graph. We then mark sensitive operations on the data-flow graph. Table 1 lists common sensitive operations on a shared variable. In this table, the left column lists the name of a sensitive operation we refer to in this paper, and the right column explains the meaning of corresponding sensitive operation. For example, sensitive operation *memcpy* denotes that the share variable is passed as an argument to system function *memcpy()* or *memcpy*-like functions or code blocks defined by users.

4.3 Data-flow Merging

Since a data-flow graph represents only sequential relations among marked sensitive operations, we need to further construct a data structure to reflect concurrent relations among these sensitive operations. This is done by

- Merging all data-flows that share a common ancestor since a shared common ancestor for different data-flows means different concurrent modifications to the same piece of shared memory,

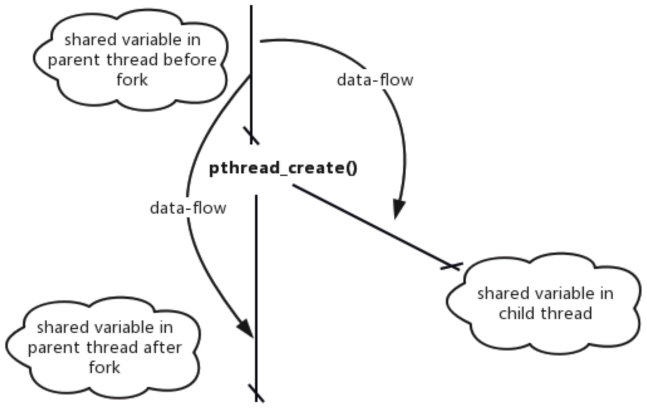

shared variable in parent thread before fork
data-flow
pthread_create()
data-flow
shared variable in child thread
shared variable in parent thread after fork

Figure 5: Data-flow graph construction

- Fine-tuning marked sensitive operations via a control-flow graph to make sure that each operation pair we come up with is indeed concurrent.

For the three types of concurrency vulnerabilities we use as an example to study the detection performance of our heuristic framework, sensitive concurrent operations of each type of concurrency vulnerability form a pair. Fig. 4 shows a pair of sensitive concurrent operations (at line 4 and line 9 of the code shown on the left-most side of the figure) our static analysis finds out for concurrency double-free. This pair is a candidate to trigger a concurrency double-free vulnerability.

4.4 Vulnerability Categorization

After obtaining pairs of sensitive concurrent operation in the last step, we need to categorize each pair into a potential type of concurrency vulnerability based on each type's operation patterns that we have distilled in Section 3.2. This categorization is necessary since a different type of concurrency vulnerability requires a different pair of sensitive concurrent operations and a different execution order of the sensitive concurrent operations in order to trigger the concurrency vulnerability.

Table 2 provides exemplary pairs of sensitive concurrent operations for each type of concurrency vulnerability studied in this paper. For a pair (A, B) of sensitive concurrent operations A and B in Table 2, the sensitive operation on the left side, i.e., A, must be executed before the sensitive operation on the right side, i.e., B, to trigger the corresponding concurrency vulnerability unless both

Table 2: Exemplary pairs of sensitive concurrent operations for each type of concurrency vulnerability

Concurrency Vulnerability	Operation Pair
Double-Free	(free, free)
BOF	(memset, memcpy) (set, memcpy) ...
Use-After-Free	(free, read) (null, read) ...

sensitive concurrent operations play an identical operation, i.e., $A = B$. When the two sensitive operations in a pair are identical, e.g., (free, free), any execution order between the two sensitive operations is equivalent. We note that the two sensitive operations in a pair must be concurrent, i.e., executed in different threads, to trigger the corresponding concurrency vulnerability.

In Table 2, the pair of sensitive concurrent operations for concurrency double-free is self-explained. The exemplary pairs of sensitive concurrent operations for concurrency use-after-free are also intuitive: the shared memory is freed or set to null in one thread and then accessed such as read, (free, read) or (null, read), in another thread. For concurrency buffer-overflows, when shared memory is passed to a *memcpy*-like function as either the source buffer or the length to be copied, and is modified concurrently in another thread, e.g. the length is changed from 10 to 20, or, as shown in Fig. 2 and discussed in Section 3.1, a NULL-ended string is overwritten with the proper NULL ending being removed, a concurrency buffer-overflow would likely occur. When the shared memory is passed to a *memcpy*-like function as the destination buffer, a concurrency buffer-overflow would likely occur if its memory address is concurrently modified in another thread, e.g. the pointer is assigned with another pointer. Each of the above buffer-overflow cases can be described with a pair of sensitive concurrent operations, with the first sensitive operation modifying shared memory followed by the second sensitive operation to pass the shared memory to a *memcpy*-like function, such as (memset, memcpy) and (set, memcpy) shown in Table 2. Listing 1 shows an exemplary output of this stage: a pair of sensitive concurrent operations (null, read) for a potential concurrency use-after-free vulnerability.

Listing 1: Static analysis output

```
Type: Concurrency use-after-free
Operation Statement        Location
read:     printf("%s",str); example2.c:78
null:     str = NULL;       example1.c:101
```

4.5 Semantic Checking

In our static analysis, we have used both a data-flow graph and a control-flow graph to find pairs of sensitive concurrent operations. A data-flow graph focuses mainly on dependency relations among different data, while a control-flow graph is about execution paths. They do not explore semantics of the statements around the two sensitive operations in a found pair to determine if the pair could possibly lead to the suspected concurrency vulnerability or not.

For example, for the reported pair shown in Listing 1, if there is a condition to check if string *str* is NULL or not before calling function *printf* for the first sensitive operation of the pair, then the suspected concurrency use-after-free vulnerability will never occur. Sending this pair to fuzz testing is simply a waste of time. For a found pair of sensitive operations of concurrency use-after-free, if we can determine that the associated variable is properly set when shared memory is freed for the left (i.e., first) sensitive operation in the pair, and there is a proper check to see if the shared memory is freed before being used for the right (i.e., second) sensitive operation in the pair, then the pair of sensitive concurrent operations cannot lead to the suspected concurrency use-after-free and should be deleted. Similar semantic checking should also be applied to reported pairs of other concurrency vulnerabilities. This would significantly reduce the set of candidate pairs to be tested by fuzz testing.

Semantic checking can be realized in several ways. We have adopted a simple approach by checking preceding conditions related to shared memory for a sensitive operation to determine if the condition that would trigger the suspected concurrency vulnerability would never be met. For example, if we determine that *printf* in Listing 1 is called only when *str* is not NULL, then we can conclude that the condition to trigger the suspected concurrency use-after-free reported by the pair shown in Listing 1 would never be met. This approach is similar to the path exploration of symbolic execution but much simpler since we focus on determining if a certain condition, i.e., the condition to trigger the suspected concurrency vulnerability, will be met or not. If we cannot determine easily, we can always resort to fuzz testing to further test it, with a possible adverse impact on the workload of fuzz testing.

5 THREAD-AWARE FUZZING

A key issue in applying fuzz testing to effectively detect concurrency vulnerabilities is how to make a fuzzer explore as many thread interleavings as possible [34]. The more thread interleavings a fuzzer explores, the more likely a concurrency error or vulnerability is triggered. However, existing fuzzers are designed to explore as many code paths as possible and thus perform poorly in exploring thread interleavings. To the best of our knowledge, there is no existing fuzzer that can explore deep thread interleavings well.

In realizing the above limitation of existing fuzzers, we advocate using a thread-scheduling fuzzing strategy to effectively explore thread interleavings of concurrent programs. The core idea in this strategy is to adjust execution orders of threads, either randomly or in a targeted manner, to generate as many thread interleavings as possible or specific thread interleavings, depending on the fuzzer's targeted applications. There are a few ways to adjust or influence execution orders of threads, such as adjusting a thread's priority, forcing a thread sleep for a certain or random time, etc.

In this section, we describe a simple thread scheduling scheme by adjusting threads' priorities. For simplicity, we assume that fuzz testing of a concurrent program is bound to one CPU core as a fuzzer would normally do. This enables us to set the thread scheduling of a concurrent program to strict *First-In-First-Out* (FIFO), which makes manipulating thread scheduling much easier. This scheme manifests in two forms, aiming at performing different tasks. They are described in detail in the following two subsections.

5.1 Interleaving Exploring Priority

Our thread scheduling in this form, called *interleaving exploring priority*, aims at exploring as many thread interleavings as possible in fuzz-testing a concurrent program. This is achieved by inserting assembly code after a new thread is forked, i.e. pthread_create is called, to manipulate the priority of the thread that executes this inserted code. The assembly code, if ever reached, will adjust the priority of the thread the assembly code resides in to a certain level such as the highest or the lowest level of priority. For each thread interleaving, the concurrent program will be tested for a fixed number of times in different iterations of fuzz testing. When a thread interleaving has completed testing, a new, untested interleaving is generated and tested. This process is repeated until all thread interleavings have been tested. If the fuzzer still runs by then, the whole process is repeated to test different interleavings again until the fuzz testing is stopped. In doing so, we hope to cover as many thread interleavings as possible, and each thread interleaving is sufficiently tested. Our experimental results indicate that this approach is very effective in finding concurrency crashes.

5.2 Targeted Priority

As we have mentioned, execution orders are critical in general in triggering concurrency vulnerabilities. The interleaving exploring priority described above, although effective in exploring thread interleavings, is ineffective in triggering concurrency vulnerabilities since, as we mentioned in Section 2.2, triggering a concurrency vulnerability normally requires meeting two requirements simultaneously: a specific input and a specific scheduling. By aiming at exploring as many interleavings as possible, it is difficult for the interleaving exploring priority to meet both requirements at the same time to trigger a concurrency vulnerability. To improve the chance to trigger concurrency vulnerabilities, we have developed another thread scheduling scheme, called *targeted priority*, to aim at exploring concurrency-vulnerability-dependent interleavings that would likely trigger targeted concurrency vulnerabilities.

Since each concurrency vulnerability candidate consists of a pair of sensitive concurrent operations, and a specific execution order of the two concurrent operations is required to trigger the potential concurrency vulnerability, we can instrument the priority-adjusting assembly code at the two sensitive operations to adjust the priorities of the two threads that run the two sensitive concurrent operations respectively so that the two threads would likely be executed in the specific order that would trigger the potential concurrency vulnerability.

More specifically, suppose there is a pair, (A, B), of sensitive concurrent operations A and B, where operation A must be executed before operation B to trigger the suspected concurrency vulnerability[1]. The inserted priority-adjusting assembly code would do the following:

- If the inserted priority-adjusting assembly code that sensitive operation B resides in is executed first, the priority-adjusting assembly code will set the thread that runs it and B to the lowest priority. This thread's original priority will be restored only after sensitive operation A has been executed.
- If execution hits the inserted priority-adjusting assembly code that sensitive operation A resides in first, nothing will be scheduled.

The above process is illustrated in Algorithm 1 (see Section 5.3 for definition of some terms used in the algorithm). In this way, fuzz testing likely executes sensitive operations A and B in the desirable order: A is executed before B, and thus likely trigger the potential concurrency vulnerability.

ALGORITHM 1: Algorithm to schedule a pair of sensitive concurrent operations in a scheduling unit

```
Input: A pair ($A$, $B$) of sensitive concurrent operations to schedule, where
       $A \neq B$, the counter of this unit, $Counter$, which is initialized to 0,
       and a threshold $\Theta$ for all counters.
if $Counter \geq \theta$ then
    return
end
if hit A then
    execute $A$;
    if B's priority has been modified then
        restore $B$'s original priority;
    end
    $Counter$++;
end
if hit B then
    if A has not been executed then
        set B's priority to the lowest;
    end
end
```

5.3 Load Balance

In a concurrent program, there are usually a set of pairs of sensitive concurrent operations that need to be tested in fuzz testing. Each pair is associated with the instruction code described in Section 5.2 to adjust the two relevant threads' priorities to make the two threads executed in a desirable order in order to trigger the suspected concurrency vulnerability. The instrumentation code for a pair is referred to as a *scheduling unit.*

In fuzz testing, a program will be executed many times. It would be beneficial if each pair of sensitive concurrent operations is tested with equal probability, i.e., each scheduling unit is executed with the same number of times. To achieve this goal, we use a counter in each scheduling unit to count the number of times the scheduling unit has been executed, as shown in Algorithm 1. Whenever a scheduling unit is executed in fuzz testing, the counter is increased by 1. If a counter exceeds a preset threshold, this corresponding scheduling unit will not be scheduled, i.e. the two threads would execute as if there were no scheduling unit. When counters of all scheduling units have exceeded the threshold, we will boost the threshold by a certain amount so that all scheduling units will be scheduled again.

[1]If sensitive operations A and B are identical, such as in a pair (free, free) for a concurrency double-free vulnerability, the reverse execution order can also trigger the suspected vulnerability. In this case, there is no need to force any specific execution order.

6 IMPLEMENTATION

We have implemented the proposed heuristic framework to explore thread interleavings in fuzz testing and to detect concurrency vulnerabilities for concurrent programs written in C with POSIX multi-thread functions. The implementation details are described in this section.

6.1 Implementation of Static Analysis

To implement the static analysis described in Section 4, we leveraged an existing concurrent static analysis tool in order to reduce our implementation workload. Such a tool should be open source so that we could modify its code to implement the desired functionalities. It should also be able to work on concurrent C programs so that we could apply it in our evaluation (see Section 7.1 for details). Among available concurrent static analysis tools meeting our requirements, LOCKSMITH [23] was selected since it was easy to use and modify. It is a static analysis tool that uses a constraint-based technique to automatically detect data races in concurrent C programs. We used it to discover shared variables for the functionality described in Section 4.1, construct data-flow graphs and control-flow graphs, and obtain information of locked areas. We modified LOCKSMITH's code to mark sensitive concurrent operations on the data-flow graph to fulfill the functionality described in Section 4.2, and mark preceding operations on the data-flow graph for each one in a pair of sensitive operations and examine these operations on both the data-flow graph and the control-flow graph to fulfill the functionality described in Section 4.5.

To implement the functionaries described in Sections 4.3 and 4.4, we wrote a program in Python using *NetworkX* module to process results from LOCKSMITH for merging data-flows and categorizing each pair of concurrent sensitive operations into a specific type of vulnerability.

Listing 2: Instrumentation assembly flags in source code

```
// in thread 1
T1:1: asm ("#con_afl_48\n\t");
T1:2: str = NULL;
T1:3: asm ("#con_priority_afl_48\n\t");
...
// in thread 2
T2:1: asm ("#con_afl_49\n\t");
T2:2: printf("%s",str);
```

6.2 Implementation of Thread-Aware Fuzzing

Our two thread fuzzing priorities were implemented based on AFL [13]. We inserted instrumentation code to adjust thread priorities to designated interleavings. This was done before and during AFL-compiling the source code of a program, as described in detail next. As a result, the source code is needed for our heuristic framework to detect concurrency errors and vulnerabilities in a concurrent program.

The instrumentation code was inserted in two steps: the first step inserted assembly flags in the source code before AFL-compiling to mark locations where our instrumentation code should be inserted, while in the second step each assembly flag was replaced with scheduling assembly code at AFL-compiling time. Same as the original instrumentation of AFL, replacing assembly flags with scheduling assembly code was done on assembly code files (i.e., .s files) generated during AFL compiling. Since we had access to source code, for simplicity, we used -o0 optimization level to AFL-compile all the tested programs.

For the interleaving exploring priority, the thread-priority adjusting code was inserted right after a call of *pthread_create* function. For the targeted priority, the thread-priority adjusting code was inserted around each sensitive operation. Inserting thread-priority adjusting code for the interleaving exploring priority is straightforward as compared with inserting thread-priority adjusting code for the targeted priority. We shall focus on describing the latter in the remaining part of this subsection.

Listing 2 shows an example of inserted assembly flags in a scheduling unit for the pair shown in Listing 1. In this listing, each assembly flag is associated with a number, such as 48 and 49 in Listing 2. These numbers indicate the execution order of the two sensitive operations in a pair to trigger the suspected concurrency vulnerability: the sensitive operation associated with a flag of a smaller number in a pair should be executed before the sensitive operation associated with a flag of a larger number in the same pair. For example, in Listing 2, sensitive operation *str = NULL* should be executed before sensitive operation *printf("%s", str)* to trigger the suspected concurrency use-after-free since the former is associated with 48 while the latter is associated with 49.

When the program was compiled by AFL, the assembly flags in a pair were recognized and replaced with a scheduling unit of scheduling assembly code. More specifically, assembly flags in the generated .s files during AFL compiling were first located, and each assembly flag right before a sensitive operation, e.g. the assembly flag at line T1:1 and that at line T2:1 in Listing 2, was replaced with scheduling assembly code to adjust the two threads' priorities according to Algorithm 1, with the sensitive operation associated with a smaller number being executed first as we mentioned above. Each sensitive operation that should be executed first in a pair is followed with an assembly flag, such as line T1:3 in Listing 2. This assembly flag was replaced with assembly code to restore the original priority of the other thread in the pair if the thread was adjusted to the lowest priority level, as described in Algorithm 1.

During fuzz testing, we allocated a scheduling trace table to record the execution information of instrumentation code, which tells what interleaving was actually executed in a test run, and how many times an interleaving was executed. We also recorded some global information such as a global threshold. If any crash was triggered in fuzz testing, the recorded information could identify the input and the interleaving associated with the crash, which would help us validate detected concurrent vulnerabilities.

7 EVALUATION

We have applied the implemented heuristic framework to a benchmark suite of six real-world C programs. Experiments were performed on Intel Xeon CPU E5-2630 v3 @ 2.40GHz with 32 logic cores and 64 GB of memory, running on Red Hat 4.4.7-17. The version of AFL [13] we used was AFL v2.51b.

Table 3: Experimental results (LOC = lines of code)

Application	LOC	Exploring Priority	Vulnerability Detected				Performance Overhead
		# of new crashes	Time	Vuln. type	# found by static analysis	# detected by targeted priority	
boundedbuf	0.4k	0	2.3s	Buffer Overflow	1	0	272%
swarm	2.2k	0	3.5s	Double-Free	1	0	109%
bzip2smp	6.3k	2	1500s	Double-Free	3	1	51%
pfscan	1.1k	1	1.2s	None	0	0	98%
ctrace	1.5k	0	2.9s	Double-Free	3	1	59%
qsort	0.7k	0	0.5s	Buffer Overflow	2	0	104%

7.1 Benchmark Suite

Since there is no available benchmark suite for detecting concurrency vulnerabilities, to the best of our knowledge, we selected several typical multi-thread C programs from previous works [8, 23, 33] using the following selection criteria:

- Lines of code could not exceed tens of thousands. This is because the static analysis tool we based on to implement our own static analysis has adopted a very precise thus costly method, which limits the tool to detect no more than tens of thousands of lines of code [28]. This limitation excludes many sophisticated but interesting software.
- Multi-thread programs written in C using POSIX multi-thread functions.
- Do not interface with the network since AFL mutated a local file that was fed into the program.
- Do not fork any new thread via a thread pool since a thread pool would affect the accuracy of data-flow in the static analysis, which would lead to too many false positives.

We collected six programs in the benchmark suite to evaluate our heuristic framework. They were *boundedbuff*, a program that implements a multi-thread producer-consumer module; *swarm*, a parallel programming framework for multi-core processors [1]; *bzip2smp*, a parallel version of bzip2 compressing tool; *pfscan*, a multi-thread file scanner; *ctrace*, a library for tracing the execution of multi-threaded programs; and *qsort*, a multi-thread implementation of quick sort.

7.2 Experimental Results

Table 3 shows the detection results of our heuristic framework in testing the benchmark suite described in Section 7.1. Our heuristic framework contains actually two separated parts to perform two different detection tasks. One is a modified AFL with the interleaving exploring priority to enable AFL to explore thread interleavings as effectively as possible to detect concurrency errors, while the other consists of our static analysis and a modified AFL with the targeted priority to detect targeted concurrency vulnerabilities such as the three types of concurrency vulnerabilities studied in this paper. The former will be referred to as the *interleaving exploring fuzzer* while the latter as the *vulnerability detection fuzzer*. The detection results for both fuzzers are included in Table 3. The detail is described next.

In Table 3, the third column shows the number of new crashes found with the interleaving exploring fuzzer, i.e., crashes found with our modified AFL with the interleaving exploring priority but not found by running the original AFL sufficiently long. The remaining columns in the table except the last one show the detection results of the vulnerability detection fuzzer: the execution time in seconds of the static analysis in the fourth column; the type and the number of suspected concurrency vulnerabilities reported by the static analysis in the fifth and sixth columns, respectively; and eventually in the seventh column the number of concurrency vulnerabilities detected by the modified AFL with the targeted priority after sending each case reported by the static analysis to the modified AFL for further testing. Thus the seventh column shows the detection results of the vulnerability detection fuzzer. The last column of able 3 shows the performance overhead of our modified AFL against the original AFL for each tested program, which will be described in detail later in this subsection.

Table 3 does not show any execution time taken by AFL fuzz testing since the time spent in AFL fuzz testing was non-deterministic. In most cases, it took about ten minutes or less for the interleaving exploring fuzzer to produce the first crash. As a comparison, the original AFL might not report any crash after running for several days. For example, in testing *bzip2smp*, our interleaving exploring fuzzer produced a crash after running in less than 10 minutes, while the original AFL did not report any crash after running for 2 days.

As we described above, the crashes reported in the third column of Table 3 were all new crashes found by the interleaving exploring fuzzer. Since there was no report on crashes of the programs in the benchmark suite by any existing fuzzer, we compared the detection results of our interleaving exploring fuzzer with the results of the original AFL. If a crash was reported by the interleaving exploring fuzzer but not reported by the original AFL after running it sufficiently long, the crash was considered new and reported in the third column in Table 3.

We have also studied the impact of our thread scheduling on the performance of AFL by comparing the total number of executions of a program to be tested in a fixed duration of time with our modified AFL against that with the original AFL. The last column in Table 3 shows the performance overhead of our modified AFL against the original AFL for each tested program, which is defined as the difference of the average execution time in running a tested program with our modified AFL, including both using the interleaving exploring priority and using the targeted priority, and with the original AFL, normalized by the original AFL's average execution time. The performance overhead ranged from 51% to 272% for the benchmark suite.

7.3 Validation of Detected Concurrency Vulnerabilities

For each concurrency vulnerability detected by the vulnerability detection fuzzer, we need to verify if it is a true positive or a false positive. We used the following manual validation process to verify the two concurrency vulnerabilities reported in Table 3: a concurrency double-free for each of *bzip2smp* and *ctrace*.

When a crash was reported, our modified AFL recorded its input and the thread interleaving setting in the crash file. With the crash report, we manually inserted the scheduling code in the source to set the thread interleaving the same as the crash interleaving and inserted assertive code before the sensitive operations to assert the condition that would trigger the detected concurrency vulnerability. Then we repeatedly ran the tested program fed with the crash input in order to hit the assertive code. If the assertive code was hit, we concluded that the detected concurrency vulnerability was a true positive. If the assertive code was not hit after running the tested program many times, we concluded that the detected concurrency vulnerability was highly likely a false positive.

Using this manual validation process, the two concurrency double-free vulnerabilities detected by the vulnerability detection fuzzer and reported in Table 3 were confirmed to be true positives.

7.4 Analysis of Static Analysis Results

The goal of the static analysis is to locate potential concurrency vulnerabilities and obtain their information to provide the modified AFL with the targeted priority to test. False positives in the static analysis would increase the workload of fuzz testing. The semantic checking in the static analysis aims at avoiding wasting time on testing obvious false positives in fuzz testing instead of at accurately detecting concurrency vulnerabilities. As a result, we had used a simple method in the semantic checking to eliminate cases that could be easily determined to be false positives, i.e., the condition that would trigger a suspected concurrency vulnerability could be easily determined to never be met.

Nevertheless, a more accurate semantic checking would help reduce the workload of fuzz testing. To analyze the performance of the static analysis, we investigated the cases reported by the static analysis but not detected by the modified AFL with the targeted priority. There were 8 such cases in total, as we can see from Table 3. By examining and debug-testing the code, we could determine that 4 cases out of the total 8, the one in *boundedbuf*, the two in *qsort*, and one in *ctrace*, were false positives, thanks partially to the small code base of these programs. The other 4 cases in larger programs could not be determined in our investigation: they looked like true positives as reported by the static analysis in our manual examination but we could not trigger them in our fuzz testing. As a result, we were unable to determine if they were true positives or not. As we shall describe in Section 8, AFL might have failed to execute the sensitive operations in a pair or insufficiently tested such a pair that had failed to trigger the concurrency vulnerability. Both would lead to false negatives.

7.5 Abnormal Time Cost of Static Analysis

From Table 3, we can see an extreme time cost, 1500 seconds, of the static analysis on *bzip2smp*, while the time cost for other programs in the benchmark suite are all 3.5 seconds or less. This observation led us to investigate the root cause of the outlier.

By examining the code of *bzip2smp*, we found a macro that was repeatedly called many times in *bzip2smp*. Listing 3 shows the piece of code. It contains a macro *BZ_ITAH*, which is called literally 50 times. This would cause the static analysis to generate at least 50 branches in both the data-flow and the control-flow graph, resulting in a long execution time for the static analysis. When we replaced the 50 calls of the macro with a for-loop, *for (i=0; i<=49; i++)*, the semantics and functionality of the piece of code remain intact, but the complexity of the data-flow and the control-flow graph in the static analysis is significantly reduced: the time cost reduced to 13 seconds from the original 1500 seconds.

Listing 3: Macro used in bzip2smp

```
#define BZ_ITAH(nn)
mtfv_i = mtfv[gs+(nn)];
bsW(s,s_len_sel_selCtr[mtfv_i],
s_code_sel_selCtr[mtfv_i])
BZ_ITAH(0);   BZ_ITAH(1);   BZ_ITAH(2);
BZ_ITAH(3);   BZ_ITAH(4);
...
BZ_ITAH(45);  BZ_ITAH(46);  BZ_ITAH(47);
BZ_ITAH(48);  BZ_ITAH(49);
```

8 LIMITATIONS AND FUTURE WORK

As reported in Section 7.2, our interleaving exploring fuzzer found three new crashes that the original AFL did not find, and typically produced the first crash within 10 minutes of running while the original AFL might not report any crash after running for several days. This indicates that the original AFL is ineffective in exploring thread interleavings in testing a concurrent program, and the same fuzzer, when combined with our interleaving exploring priority, can explore thread interleavings very effectively. This is because our interleaving exploring priority aims at exploring as many thread interleavings as possible. Our interleaving exploring priority empowers a fuzzer to effectively detect concurrency errors, a great enhancement to existing fuzzers.

In addition, our vulnerability detection fuzzer could detect two concurrency vulnerabilities, and both vulnerabilities were confirmed to be true positives, as reported in Section 7.2. This demonstrates the power and effectiveness of our vulnerability detection fuzzer in detecting targeted concurrency vulnerabilities.

Nevertheless, there are several limitations for the current implementation of the heuristic framework, mainly due to the tools we based on to implement the framework. These limitations are discussed in the following subsections. We are actively working on improving the heuristic framework to address some of these issues.

8.1 Scalability of Static Analysis

LOCKSMITH [23], the static analysis tool we based on to implement our static analysis, is precise but complex, which prevents it from working on programs exceeding tens of thousands of lines of code [28]. This was the main reason to choose small utility programs instead of more interesting ones in our evaluation experiments. It is desirable to choose a more scalable open-source static analysis tool to implement our static analysis so that larger and commonly used concurrent programs can also be tested with the heuristic framework.

In addition, the semantic checking can be improved to reduce false positives to avoid wasting time on testing false positive in our fuzz testing.

8.2 Capacity of AFL in Exploring Paths

We have adopted AFL to implement our interleaving exploring fuzzer and our vulnerability detection fuzzer. It is well-known that AFL explores sophisticated programs in a shallow manner, and this problem has been addressed recently in [6, 14]. It is desirable to use a more sophisticated fuzzer that can explore code paths deeply or guide testing towards executing the sensitive operations reported by the static analysis such as that presented in [2].

8.3 Restrictions of Manual Validation

The manual validation described in Section 7.3 to validate detected concurrency vulnerabilities is a labor-intensive work. Based on a crash report by our vulnerability detection fuzzer, we need to manually insert scheduling code into the source to ensure that the same interleaving that caused the crash in the fuzz testing would be used in validation, then examine the program code to determine the root cause of a reported concurrency vulnerability in order to decide the condition to confirm the concurrency vulnerability. Next we need to insert assertive code around the sensitive operations to determine if the condition is really hit in validation, and then run the program fed with the crash input repeatedly in order to hit the assertive code.

We need to run the tested program repeatedly in the validation process since the crash report is insufficient to replay the crash. According to [22], it requires to record the information of eight factors to deterministically replay a concurrency error, which is far more than the information recorded by AFL.

Among all these limitations, the insertion of scheduling code into the source during validation can be automated in a way similar to *ConMem-v* in [36]. Writing such automatic tool is of lower priority since the number of cases to be validated is small by now. Although taking some time, running a program to be tested repeatedly in the validation phase has a high chance to repeat the crash.

The most challenging task in our manual validation is actually the comprehension of the code in order to identify the root cause of a reported concurrency vulnerability so that we can determine a condition to place into inserted assertive code such that triggering the assertive condition confirms the reported concurrency vulnerability. This assertive condition differs from the condition for a pair of sensitive operations that the static analysis finds and the fuzz testing uses to trigger a suspected concurrency vulnerability. The latter is the execution order of the two sensitive operations in a pair that would trigger the its potential concurrency vulnerability. It is coarse, at the thread level. The former, on the other hand, is fine-grained and requires understanding the root cause of the concurrency vulnerability. It needs to guarantee confirmation of the suspected concurrency vulnerability once triggered. Obtaining this assertive condition in our validation typically requires thorough understanding of the relevant code written by others, which is labor-intensive and time-consuming, especially for concurrency buffer overflows. Due to its complexity, there is a chance that the derived assertive condition is incomplete, which may lead to failure to confirm a true positive. As a result, a false positive determined by our manual validation is probabilistic instead of deterministic. On the other hand, a true positive determined by our manual validation is always deterministic.

8.4 Additional Limitations

In addition to the above limitations, there are some additional limitations in our implementation of the heuristic framework. The heuristic framework currently works only with concurrent programs written in C using POSIX multi-thread functions requires the source code to detect concurrency errors and vulnerabilities in a concurrent program. It is desirable to extend the heuristic framework to cover programs written in other languages and using other multi-thread functions, and to cover binary programs without using source code. The ideas presented in this paper work for the these extensions, but it requires a great effort to realize them.

9 CONCLUSION

In this paper, we proposed a heuristic framework to detect concurrency errors and vulnerabilities in concurrent programs. It includes two separate fuzzers. One fuzzer, the interleaving exploring fuzzer, explores interleavings effectively to test as many interleavings as possible. It can detect concurrency errors effectively and efficiently. The other fuzzer, the vulnerability detection fuzzer, first applies static analysis to locate sensitive concurrent operations, categorize each finding to a potential concurrency vulnerability, and determine the execution order of the sensitive operations in each finding that would trigger the potential concurrency vulnerability; and then directs fuzz testing to explore the specific execution order of each finding in order to trigger the potential concurrency vulnerability.

We used three types of common concurrency vulnerabilities, i.e., concurrency buffer overflow, double-free, and use-after-free to evaluate the proposed heuristic framework with a benchmark suite of six real-world programs. In our experimental evaluation, the interleaving exploring fuzzer reported three new crashes that were not reported by the existing fuzzer, AFL, that our fuzzer was based on. The interleaving exploring fuzzer typically produced the first crash within 10 minutes of running while the original AFL might not report any crash after running for several days. These experimental results indicate that our interleaving exploring fuzzer can effectively explore interleavings in detecting concurrency errors while the original AFL cannot. Additionally, the vulnerability detection fuzzer detected two concurrency vulnerabilities, and both vulnerabilities were confirmed to be true positives. This demonstrates the power and effectiveness of the vulnerability detection fuzzer in detecting targeted concurrency vulnerabilities.

REFERENCES

[1] D. A. Bader, V. Kanade, and K. Madduri. 2007. *SWARM: A Parallel Programming Framework for Multicore Processors.* 1–8 pages. https://doi.org/10.1109/IPDPS.2007.370681

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17).* ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[3] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

[4] Jacob Burnim, Koushik Sen, and Christos Stergiou. 2011. Testing Concurrent Programs on Relaxed Memory Models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 122–132. https://doi.org/10.1145/2001420.2001436

[5] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[6] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. *CoRR* abs/1803.01307 (2018). arXiv:1803.01307 http://arxiv.org/abs/1803.01307

[7] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-guided Generation of Concurrent Tests. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 266–277. https://doi.org/10.1109/ICSE.2017.32

[8] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. 2013. CONCURRIT: a domain specific language for reproducing concurrency bugs. *Acm Sigplan Notices* 48, 6 (2013), 153–164.

[9] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting null-pointer dereferences in concurrent programs. In *Proceedings of ACM Sigsoft International Symposium on the Foundations of Software Engineering*. 1–11.

[10] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 16th Network and Distributed System Security Symposium*, Vol. 8. 151–166.

[11] Pallavi Joshi, Mayur Naik, Chang Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proceedings of Computer Aided Verification*. Berlin, Heidelberg, 675–681.

[12] Marek Kroemeke. 2014. Apache 2.4.7 mod_status - Scoreboard Handling Race Condition. https://www.exploit-db.com/exploits/34133/.

[13] Lcamtuf. 2018. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[14] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[15] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 32, 12 (1990), 32–44.

[16] NIST. 2018. National Vulnerability Database. https://nvd.nist.gov/.

[17] NVD. 2018. CVE-2010-5298 Detail. https://nvd.nist.gov/vuln/detail/CVE-2010-5298.

[18] OpenBSD. 2014. OpenBSD 5.4 errata 8. https://ftp.openbsd.org/pub/OpenBSD/patches/5.4/common/008_openssl.patch.

[19] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[20] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. 2015. UNICORN: a unified approach for localizing non-deadlock concurrency bugs. *Software Testing, Verification and Reliability* 25, 3 (2015), 167–190. https://doi.org/10.1002/stvr.1523 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1523

[21] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/1806799.1806838

[22] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/1772954.1772958

[23] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-sensitive correlation analysis for race detection. *Acm Sigplan Notices* 41, 6 (2006), 320–331.

[24] Nishant Sinha and Chao Wang. 2010. Staged Concurrent Program Analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/1882291.1882301

[25] Sherri Sparks, Shawn Embleton, Ryan K Cunningham, and Cliff Changchun Zou. 2007. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 477–486.

[26] Sebastian Steenbuck and Gordon Fraser. 2013. Generating unit tests for concurrent classes. In *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 144–153.

[27] Valerio Terragni and Shing-Chi Cheung. 2016. Coverage-driven Test Code Generation for Concurrent Classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1121–1132. https://doi.org/10.1145/2884781.2884876

[28] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 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 (ESEC-FSE '07)*. ACM, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

[29] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. 2017. How Double-fetch Situations Turn into Double-fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=3241189.3241191

[30] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Washington, DC, USA, 497–512. https://doi.org/10.1109/SP.2010.37

[31] Wikipedia. 2018. Dirty COW. https://en.wikipedia.org/wiki/Dirty_COW.

[32] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2508859.2516736

[33] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=2342788.2342803

[34] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 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 (OOPSLA '12)*. ACM, New York, NY, USA, 485–502. https://doi.org/10.1145/2384616.2384651

[35] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 251–264. https://doi.org/10.1145/1950365.1950395

[36] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 179–192. https://doi.org/10.1145/1736020.1736041

[37] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. 2018. OWL: Understanding and Detecting Concurrency Attacks. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*. 219–230.