Fuzz Testing of Multithreaded Applications Based on Waiting

Oleg Doronin, Karina Dergun, Andrey Dergachev, and Aglaya Ilina

Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, Russian Federation
dorooleg@niuitmo.ru, 245704@niuitmo.ru, amd@itmo.ru, dvanoska@mail.ru

Abstract. In our days, it is hard to imagine big software products written without the use of multithreading. However, they not only use multithreading, but are also complicated by being distributed. On one hand, it gives performance advantages, but on the other hand, it becomes much more difficult to find bugs and test such applications. When developing programs that use multithreading, we can find the following types of errors: priority inversions, deadlock, livelock, ABA problem, and others. Such errors can lead to large financial losses, for example, in the banking infrastructure, or losses of human lives in aircraft engineering, civil engineering, medical devices and other areas. Special tools such as Valgrind, Google TSAN and others are used to find such bugs. Until recently, such tools were not able to fuzzing testing multithreaded applications, but now Google TSAN has a special module. The main limitation of the testing fuzzing module is that it is not able to handle waiting on non-atomic variables. The results presented in this paper allow us to carry fuzzing testing of threads and at the same time correctly handle the situation with waiting on variables that are not atomic, as well as examples on which the improved algorithm successfully copes with handling such waiting.

Keywords: multithreading · data races · deadlock · bug-finding tools · fuzzing testing

1 Introduction

Fuzz testing or Fuzzing [6] allows to produce a new test coverage by trying out various variants of the program's execution in automated fashion. The simplest way is a brute force style of enumerating every execution variant. This approach is not effective, because computing power is not enough to go through every combination of inputs for most programs. Therefore, in fuzzing testing, statistics are collected and analyzed to reduce the required number of inputs. One option

is to analyze the code coverage and choose different error-finding strategies, with the subsequent build of assumptions related to the probability of error.

As for the fuzz testing of multithreaded applications, instead of trying out variants of the input arguments, the search goes through the combinations of thread execution sequences. And in this case there are different strategies for performing the search of execution sequences.

At the moment, fuzz testing is being implemented into the main development branch of the ThreadSanitizer (TSAN) [3] [4] tool. The following articles describe previous work done on TSAN:

1. [1] An architecture for fuzz testing of multithreaded application, with various thread execution planning strategies implemented to find errors in threaded code.
2. [1] Further work expanded the coverage of the code that can be tested using the developed module; added support for working with lock-free algorithms and atomic variables.
3. [1] However, there are still limitations to the current version, one of which is the lack of support for mutexes [2], so the present work removes this shortcoming.

1. Comparison of existing solutions

	RRD	**Lock-free fuzzing**	**Fuzzing with Lock**
Doesn't require changes in user code	-	+	+
Supports the blocking synchronization algorithms	+/-	-	+/-
Supports working with thread local storage	-	+	+
Supports working with non-atomic variables	-	-	-
It does not require the individual implementation of each blocking algorithm	-	X	-

Here's a comparison of the various solutions that exist at the moment. The comparison is made on 5 criteria:

1. **Doesn't require changes in user code** - this means that the user can use the tool to phase out thread testing for the source code in the c++ language without making any changes to it. For example, Relacy Race Detector (RRD) [10] in some cases requires changes in the code, which is a big drawback. As a result, the code-changing approach can be more difficult. It's very expensive for large enterprise applications to make changes to the source code, as well as to maintain and accompany two versions. The second problem is that such changes can lead to new errors, making it difficult to test and develop software. As a result, the following decisions win in this case: Lock-free fuzzing [1] and Fuzzing with Lock [2]

2. **Supports the blocking synchronization algorithms** - the above tools are mainly focused on working with atomic variables. When it comes to the primitive synchronization (mutex, shared_mutex, conditional_variable), then not all tools are able to work with them. For example, RRD and Fuzzing with Lock can only work with some synchronization primitives, and Lock-free fuzzing can't work with them.
3. **Supports working with thread local storage** - many advanced algorithms use thread local storage. It's hard to imagine a memory allocator that doesn't use TLS [7]. Also, many lock-free algorithms use such memory for optimization or for storage of states. These algorithms include Hazard Pointers [8]. RRD doesn't know how to work properly with TLS. The reason for this behavior is the substitution of real threads for fibers
4. **Supports working with non-atomic variables** - this wording is very subtle and means that the algorithm is able to work with expectations on non-atomic variables. An example of this case is described below. It turns out that not one of the existing solutions does not know how to work correctly with such variables
5. **It does not require the individual implementation of each blocking algorithm** - the downside of RRD and Fuzzing with Lock is that they only support a limited number of synchronization primitives. When you add new synchronization primitives, you have to wait for a new implementation.

This work improves the fuzz testing module described in [1] and [2] by supporting the correct handling of locks/waiting on non-atomic variables and eliminates the requirement to implement all synchronization algorithms.

2 Waiting-based algorithm

Let's start by describing the fuzz testing algorithm for threads based on waiting, with an example that causes previous approaches to fuzz testing to hang:

Example barrier

```
volatile std::byte barrier = 0;
void thread1() {
    while (!barrier);
}

void thread2() {
   barrier = true;
   //...
}
```

This example uses a simple barrier that suspends thread 1.

The code is fully valid up to the CPU memory model, as the minimum addressed memory unit is a byte.

The main assumption on which we base the developed algorithm is that errors in multithreaded applications appear only at synchronization points. Therefore, thread switching points are selected among operations: reading/writing into an

atomic variable, capturing/releasing a mutex, waiting/notification on a conditional variable, and others.

Based on the above, we design the interface for thread scheduling fuzzing in the simplest form of a single SynchronizationPoint method.

Interface for schedulers

```
class IScheduler {
    virtual void SynchronizationPoint() = 0;
};
```

To improve the thread scheduler fuzzing module, we now need to solve two problems:

1. Implement planning algorithms for the presented interface
2. Introduce the developed interface into the TSAN architecture

Let's start with the second problem and describe how TSAN works:

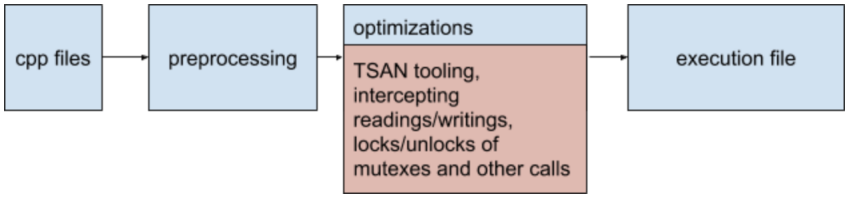

Fig. 1. Compilation pipeline

The image above shows a simplified layout of the program's compilation in the C language. It embeds TSAN in the source code during the compilation phase. As a result, we get the source code with replaced functions to work with mutexes, conditional variables, instrumented reading and writing operations into variables, and other intercepted functions.

This approach allows to replace certain functionality transparently and to create algorithms based on it to find errors in the code. From the user's point of view, it suffices to compile the source code with special compiler options, and algorithms for finding errors will work. A downside of this approach is that additional compilation time is required, and even with separate compilation the linking stage can take considerable amount of time. However, for well-structured programs this shortcoming is negligible.

Let's look at what the described architecture looks like from a code perspective. Suppose the user code gains ownership of the mutex:

Mutex Example

```
pthread_mutex_t mutex;
//...
pthread_mutex_lock(&mutex);
// application logic
pthread_mutex_unlock(&mutex);
```

In fact, this code is converted to the following:

Converted code

```
pthread_mutex_t mutex;
// ...
tsan_mutex_lock(&mutex);
// ...
tsan_mutex_unlock(&mutex);
```

The implementation of tsan_mutex_lock/unlock is taken care of by the developers of algorithms for finding errors. For SynchronizationPoint, for example, the use of tsan_mutex_lock will look like this:

Converted code

```
int tsan_mutex_lock(void* mutex) {
   IScheduler::SyncronizationPoint();
   //...
   pthread_mutex_lock(mutex);
   //...
   IScheduler::SynchronizationPoint();
}
```

In example above we made two points of synchronisation: before taking the mutex and after it. This is how IScheduler is embedded into TSAN. We now progress to description of our algorithm for fuzz testing of multithreaded applications. This algorithm should allow to deal with cases of thread hangs such as described above. The states each thread in IScheduler can be in are:

1. UNKNOWN - the thread is in this state until it reaches the first synchronization point.
2. RUNNING - marks the main execution thread. Only one thread in the program can have such this state at every moment in time.
3. WAIT - a thread in this state is waiting for its turn for execution.
4. OUT_TIME - this state happens if a thread has exhausted its execution quant but has not reached the next SynchronizationPoint. One reason for this state can be the hanging of the thread on the waiting event, just as described in the example with the barrier. Here, the thread remains on execution, and there may be several threads in OUT_TIME state. When these threads reach SynchronizationPoint, they go into WAIT.

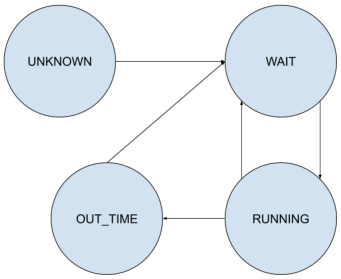

Fig. 2. Graph of states

Figure 2 shows the graph [9] states in which the process and transitions between states may be located. Let's describe an example on two threads in which threads will move between states on the graph:

1. Let two T1 (UNKNOWN) and T2 (UNKNOWN) threads be created in the system. Since these threads have not met the synchronization point, they are at the UNKNOWN point and the scheduling algorithm believes that such threads do not exist in the system.

2. Suppose the T1 thread read the atomic variable or captured the synchronization primitive, then it immediately goes into t1 (WAIT). The transition from WAIT to RUNNING can happen instantly if the scheduling algorithm decides so.

3. Let's say T1 (WAIT) stayed that state, but now the T2 thread has recorded into an atomic variable, and it's gone into T2 (WAIT) and instantly switched to T2 (RUNNING).

4. While the T2 thread was running, its time quant could run out and the WatchDog thread decided to mark it OUT_TIME and run the T1(RUNNING) thread. When the T2 thread reaches the next synchronization point, it will go into T2 (WAIT) and wait for the T1 thread to reach the next synchronization.

The example above describes a typical example of a thread wandering through such a graph. It is worth noting that WatchDog constantly monitors all the threads running in the system at some interval and can change these states for arbitrary threads.

The problem of thread hang-ups on a normal variable is solved by a state of OUT_TIME, when physically several threads can be executed. This algorithm imposes restrictions on the data structures used in IScheduler: they must be thread-safe. But what we get is the benefits of no hang-ups in these algorithms for different thread planning strategies; it works for any production-ready applications.

Let's look now at how to manage thread states. WatchDog is used to manage the OUT_TIME states. The schema shows the application architecture:

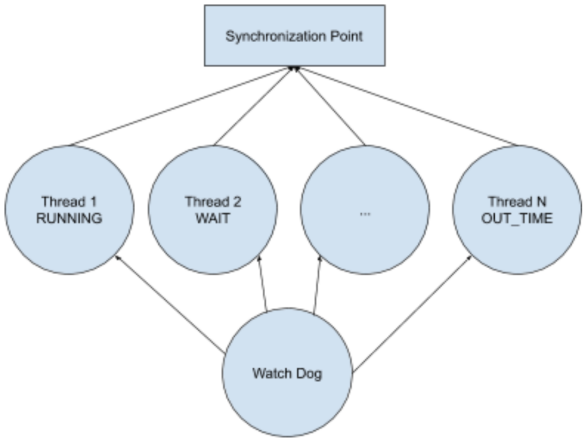

Fig. 3. Scheduler architecture

The system has N threads, where N can increase or decrease. WatchDog selects execution threads through SynchronizationPoint, except for the OUT_TIME state processing. A special WatchDog thread monitors all states and durations of threads' work. It sets OUT_TIME state if the thread has been running longer than a defined period of time. Depending on the choice of the length of this time period, we can balance the quality and time of work.

The pseudo-code for the SynchronizationPoint method implementation is:

tsan_mutex_lock

```
int tsan_mutex_lock(void* mutex) {
   IScheduler::SyncronizationPoint();
   //...
   pthread_mutex_lock(mutex);
   //...
   IScheduler::SynchronizationPoint();
}
```

In the example above, we do two synchronization points: before the mutex capture and after the capture. This is how IScheduler is introduced into TSAN.

Now let's go to the description of how to build a fuzzing algorithm for testing multithreaded applications, which would bypass the thread sagging cases described above.

Let's start by describing the states in which each thread can be inside IScheduler:

SynchronizationPoint pseudo-code

```
SynchronizationPoint():
    tid = GetTid();
    oldState = state[tid];
    state[tid] = WAIT;
    if (oldState = RUNNING) {
```

```
6           nextTid = GetNextTid();
7           state[nextTid] = nextTid;
8       }
9       While (state[tid] == Wait) Yield();
```

The pseudo-code above sets the (initial) state for each thread to WAIT unless it was in OUT_TIME state; and then the next thread is selected for execution, so there can only be one RUNNING thread on the execution. The rest of the work for OUT_TIME state processing and preserving the invariant of just one RUNNING thread takes place in the WatchDog thread.

3 Conclusion

This work improved the module for fuzz testing of multithreaded applications in Google TSAN. We added support for the correct processing of application hang-up on non-atomic variables. This result allows TSAN to test any multithreaded algorithms. For example, if we want to improve the quality of lock-free algorithm testing, such as the libcds [5] library, it suffices to set an infinite time for the state of OUT_TIME. We can see the relevance of the results through the test cases where the use of fuzz testing infrastructure led to a hanging state, but now it works fine. We have created a review request for integration into google TSAN main branch: https://reviews.llvm.org/D66235

References

1. Doronin O., Dergun K., Dergachev A. Automatic fuzzy-scheduling of threads in Google Thread Sanitizer to detect errors in multithreaded code // CEUR Workshop Proceedings - 2019, Vol. 2344, pp. 1-12
2. Derghun K.I., Doronin O.V. Fazzing testirovanie fine-grained algoritmov, Sbornik tezisov dokladov kongressa molodyh uchenyh. Elektronnoe izdanie. – SPb: Universitet ITMO, [2019]
3. ThreadSanitizer project: documentation, source code, dynamic annotations, unit tests. http://code.google.com/p/data-race-test
4. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: data race detection in practice. WBIA (2009)
5. M. Khizhinsky, CDS C++ library, https://github.com/khizmax/libcds
6. Majkl Satton, Adam Grin, FUZZING. Issledovanie uyazvimostej metodom gruboj sily, 2009
7. Patrick Carribault, Marc Pérache, Hervé Jourdren, Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications, 2011
8. Maged M. Michael, Michael Wong, Hazard Pointers. Safe Resource Reclamation for Optimistic Concurrency, 2016
9. Keijo Ruohonen, GRAPH THEORY, 2013
10. Dmitry Vyukov, Relacy Race Detector, http://www.1024cores.net/home/relacy-race-detector