TwoFuzzingLong: A Multi-threaded Approach to Fuzzing

Dhruv Swarup
UC Berkeley
Berkeley, CA, US
dswarup@berkeley.edu

Amit Jain
UC Berkeley
Berkeley, CA, US
amit_jain@berkeley.edu

Keywords: Fuzzing, Multi-threading, Parallelism, Synchronization, Performance

ACM Reference Format:
Dhruv Swarup and Amit Jain. 2021. TwoFuzzingLong: A Multi-threaded Approach to Fuzzing. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Fuzzing has become an integral part of the software development life cycle and has taken on popularity in the industry as a tool to help uncover software bugs and vulnerabilities. Fuzzing tools require minimal interaction allowing them to be run continuously for several days on end. The advantages of fuzzing have made the technique particularly beneficial for large programs with an extensive range of possible inputs.

One area of continuous improvement is how to make fuzzing tools more effective such that they can explore more code paths in the duration for which they are run. The more code coverage a fuzzer can obtain, the greater chance it has for exposing bugs and vulnerabilities such as security loopholes.

One enhancement to fuzzing has been to generate grammar based constraints from symbolic execution of the program [2]. These constraints act as a specification to ensure valid inputs are generated by the fuzzing algorithm. More recently, efforts in improving the effectiveness of fuzzer have been geared towards integrating Machine-Learning techniques with fuzzing. One approach has been to use reinforcement learning to guide the mutations for an input [4].

In a similar avenue, we introduce another mechanism for improving the effectiveness of a fuzzer. However, instead of improving the fuzzing algorithm itself by guiding the mutations, we focus our efforts on improving the computational efficiency achieved by the fuzzer. We explore how a fuzzer can be implemented as a multi-threaded application such that we can run several fuzzing loops in parallel on a multi-core system. By using multiple threads instead of multiple processes, we are able to share the test inputs easily, and ensure that each fuzzing loop is not overlapping the other in code coverage.

Our preliminary results show a dramatic performance improvement over a single-threaded fuzzing application. In our sample test program, the multi-threaded approach allowed us to explore all paths within a fraction of the time compared to the single-threaded approach. Across 100 runs, we measured on average a speedup of 22x when running the fuzzing loop as a multi-threaded application on 6 logical cores. We also tested a multi-process approach to fuzzing where we deploy independent fuzzing instances across each of the 6 cores. In the multi-process approach, the inputs are not shared and as a result each process may explore redundant paths. The multi-threaded implementation provided a speedup factor of approximately 7x when compared to the multi-process implementation.

2 Background and Motivation

2.1 Fuzzing

Fuzzing is a method of code testing and verification that uses randomness to search the program input space. The basic idea here is to run the code hundreds of thousands of times with random inputs, and measure characteristics of the code when run with that input. Some examples of these run-time characteristics are code coverage, crashes and the number of times basic blocks are run. For the purposes of this paper, lets consider code coverage as a running example. Every time a piece of code traverses a new branch, it is said to have different coverage. To "search" for higher coverage quickly and more efficiently, the search space is pruned by favoring those inputs that move the algorithm in that direction. This is done by storing and mutating those inputs with "interesting" results. The following paragraphs explain the actual process in more detail.

Fuzzing starts off with an initialization phase, where the program instrumentation is done first, and then the user provided seed inputs are processed to collect the coverage information. After this, the main loop begins, where the same few steps are followed during each iteration:

, ,
© 202 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

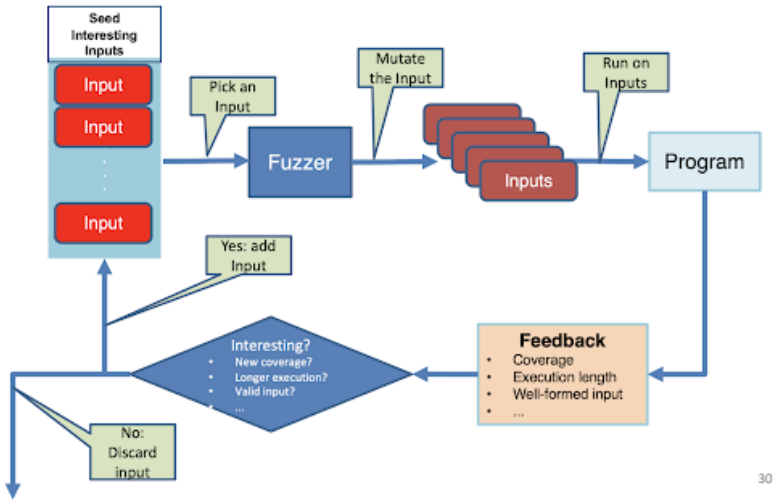

Figure 1. The fuzzing process

1. Pick an input from the queue
2. Mutate the input
3. Run the program with the mutated input
4. Check if the results are interesting
5. Conditionally add the mutated input to the queue

2.1.1 Instrumentation. Before the fuzzer can run, the program needs to instrument the code, because that allows us to collect coverage information automatically. This involves parsing the code file, and adding function calls that can track the coverage based on which branch the program takes.

2.1.2 Processing the seed inputs. The program starts with a small batch of inputs that the user provides. We will call this the "seed set". Each of the inputs in the seed set are first run by the program under test, and the coverage is collected for each input, and stored. Then, these inputs are added into a queue, which the fuzzer can now use.

2.1.3 Mutations. An input is extracted from the queue, and passed to the "mutator" function. The mutator uses several techniques to randomly change some small part of the selected input. Some common steps are:

- bit/byte flips
- bit/byte sets
- bit/byte clears
- set running bits/bytes
- deleting characters
- inserting characters

There are basically an infinite number of mutation functions that can be used. Currently, there is no "one perfect method", and a "standard" that determines the best set of operations that should be done on each input. Every fuzzer has its own set of mutations that can be very different from other programs.

2.1.4 Check if interesting. After the mutations, the program is run, and the statistics are collected. If these are deemed interesting, then the mutated input is added back into the queue. Using coverage as an example again, the inputs can be added back when they exceed some threshold of "good coverage", which can be the max coverage seen so far. This function can be customized based on what parameters are being measured, and on what kind of domain the program is being tested for.

2.2 Existing Parallelism

The existing methods that we encountered with parallelism involved deploying the single threaded fuzzer across all the logical cores available. In other words, several instances of the fuzzing program were created and the scheduler would be responsible for distributing these instances across cores. This approach is commonly known as multi-processing, where each instance of the program has its own separate virtual memory space. Since many fuzzing techniques use random mutations, some instances are able to discover new code paths faster than others. As a result, employing parallelism often increases performance many fold. This idea has also been extended to multiple clusters across a distributed server farm.

To take advantage of what the other fuzzers find, there is a need for inter process communication, so that each fuzzer can share its new and positive developments with the other fuzzers. In this synchronization process, the queues of each fuzzer are parsed, and the inputs with new coverage are added back to the master queue. Each fuzzing instance has access to the master queue and can retrieve a new set of inputs to once again begin the fuzzing loop resumes. This synchronization step is done after a relatively large amount of time: it can vary from once every few minutes, to the

order of hours. Some examples of popular fuzzers that do this are listed below.

2.2.1 ClusterFuzz. This is a tool developed by google, where the focus is to implement a "scalable fuzzing infrastructure that finds security and stability issues in software". Google's instance of this tool runs on almost 30,000 virtual machines, and has found 25,000+ bugs in popular applications like Google Chrome.

2.2.2 American Fuzzy Lop (AFL). Here, the implementation involves manually starting a process for each instance of the fuzzer. Each fuzzer stores interesting inputs in a file pipes, which is also used in the synchronization process.

3 TwoFuzzingLong: Fuzzing with Posix-threads

3.1 Goals and Expectations

The main goal of this program is to improve the efficiency of the fuzzer by leveraging a continuously shared queue instead of one that is synchronized at an interval. The easiest way to do this is to use multi-threading because of the shared program stack, instead of the multiprocessing approach. In the context of our project, we do not consider the normal differences in the creation of these two types of parallel processes, because they only occur once on creation, and this is negligible compared to the fuzzing times.

In multiprocessing, each fuzzer has its own queue that it adds interesting inputs to. To take advantage of each fuzzer, the synchronization process is done as described before. In multi-threading however, there is only one queue, that each instance of the fuzzers use. The interesting inputs are always synchronized, because each fuzzer adds its newly found inputs to the same queue. This means that theoretically, the fuzzers should be able to immediately take advantage of the results of another fuzzer.

However, there are a few concerns that might reduce the efficiency of such a method. For example, the use of locks to access the shared resources can block other threads. This could subsequently cause longer wait times as the number of threads increase. On the other hand, the amount of time that the fuzzers take could be long enough for the lock clashes to not matter. This can be measured by varying the number of threads and running the multiprocessing fuzzer. In the case where the locks do not clash, there should simply be a linear decrease in runtime from 1 to n cores to find a certain coverage. However, we predict that although performance will still improve, it will not be linearly. There are enough queue accesses in the code for it to slow down a little.

Regarding the comparison between the current methods of parallelization and ours, we predict that there will be a 2-4x improvement in performance (for 6 threads). The main reason for this would be that there are 6 threads contributing to the interesting inputs queue, so the queue will get new inputs at 6x the rate of non shared queues.

3.2 Attempts

To implement the multi-threaded queue, we had several possibilities each with their own pros and cons. They are outlined here.

3.2.1 Editing AFL. The first approach we tried for to edit the American Fuzzy Lop (AFL) source code. The main issue that we faced was the fact that there were hundreds of queue accessed interspersed within only one file in the repository. This presented a huge challenge as we would have to lock each of these accesses. Not just that, but having to debug such a large piece of code with parallelism is simply not feasible given our time frame.

3.2.2 Fuzzer from scratch (Python). The next approach that made sense was building our own fuzzer from scratch with very limited functionality. A python fuzzer would have been the quickest to implement. However, the Python interpreter uses a Global Interpreter Lock (GIL) that blocks multiple threads to use it at the same time. This means that we can achieve concurrency, but not true parallelism.

3.2.3 Fuzzer from scratch (C). The path we chose was to write a small fuzzer from scratch. It would use a small set of mutation functions, with hard-coded instrumentation, and use a small test program as a proof of concept. The next section digs into the details of the implementation of this program.

3.3 Final structure

3.3.1 Parent Thread. The parent thread is responsible for the setup of the program. At the beginning of the program, it will count the number of logical cores in the system. This thread is responsible for the initialization of the fuzzing process, where it will take the seed inputs, run them on the test program, collect feedback, and construct the preliminary queue, by inserting each of these inputs after sorting them by coverage.

Then it will create as many "child" threads as logical cores. By using the Linux POSIX API, we then bind each of these child threads to a specific core. This increases the utilization of each of the cores because each thread can now run on different cores, as opposed to the kernel scheduling a few threads to the same core. Although there is no guarantee that the cores won't be handling other processes, such as system background activities, and higher priority kernel tasks, we have added the capability to increase the priority of each thread we create. Finally, the parent thread is responsible for garbage collection as well, once the other threads have converged to a desired coverage level.

Algorithm 1 Parent thread: Main Loop

Input: Fuzzing domain "D" and seed inputs
Output: Inputs with largest coverage or WCET
$y \leftarrow CountLogicalCores()$
$X \leftarrow createQueue()$
$X \leftarrow runSeedInput(X)$
$CreateThread(X, performanceStatistics)$
for i in y **do**
    $CreateThread(core_i, X, fuzzloop, D)$
end for
$JoinThreads()$

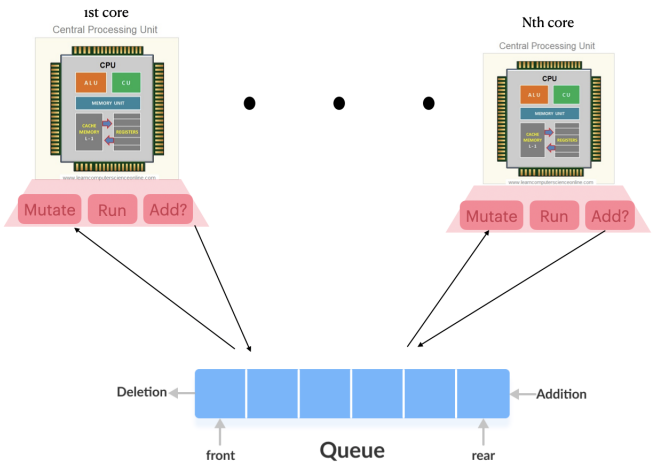

Figure 2. Multi-Threading

3.3.2 Child-fuzzing Threads. Each child thread can now run the main fuzzing loop, which involves all the steps mentioned in section 2.2 (mutating input, running on test program, etc.). All enqueues and dequeues from this common queue are executed solely by the child threads in their own fuzzing loop. Since the queue is sorted based on the domain specific feedback (such as largest code coverage), each child thread will always dequeue and fuzz the first element it sees, which will have the largest coverage.

3.3.3 Progress Thread. The main parent thread also creates one more thread that is responsible for displaying the real-time information and progress of the fuzzers. This includes:

1. CPU time
2. Wall time
3. Maximum coverage reached
4. Total number of fuzzing loops executed
5. Size of the queue

3.3.4 Mutation Function. Our mutation function uses six mutations that it selects at random. It chooses between bit and byte flip, set, and clear. For each function, the bit, and byte selected is also at random. This is good enough for the purposes of the project, because we care more about the comparison between the different types of parallelization rather than the absolute performance.

Algorithm 2 Child Thread: FuzzLoop

Input: Fuzzing domain "D"
Output: Test inputs into common queue
while *true* **do**
    $Mutex\_lock$
    $R \leftarrow getFirstQueueElemenet(X)$
    $Mutex\_unlock$
    $T \leftarrow Mutate(R)$
    $F \leftarrow RunTestProgram(T)$
    if $isInteresting(F)$ **then**
        $Mutex\_lock$
        $addToQueue(F, X)$
        $Mutex\_unlock$
    else
        $DiscardF$
    end if
end while

3.3.5 Instrumentation. For program instrumentation, we use a hardcoded implementation into the test program, where we use a bit array to store the coverage information. Each bit in the array represents a different branch. At the start of each conditional statement, we add some code to set the bit corresponding to that branch. Since the test program that we use is relatively small with around 7-12 branches this was relatively straightforward to instrument.

3.3.6 Queue. The queue that we use is a sorted linked list. Each node in the queue stores an input and the information gathered after running the program with that input. After collecting the domain specific feedback, each child thread will also do a sorted insertion back into this queue, thus we are effectively prioritizing the fuzzing of interesting inputs. As previously mentioned, each of these queue accesses will be locked by a single mutex.

3.4 Results

3.4.1 Single-core vs Multi-core. After the implementation of all the functionality, we used a Ubuntu 20.04 VM, running on a 8-core intel i7 macbook pro to test our results. The preliminary test runs on our instrumented program show compelling results for fuzzing with a multi-threaded approach. Compared to a single threaded application, the results are better than expected. A dual core system with two threads already provides a speedup factor of 7x (Table 1). The single threaded fuzzing program took roughly 22s to explore all branches in our test program, whereas the program with two threads and two cores enabled took only 2.93s. As we increased the number of cores and corresponding threads, the

Single Core (s)	Mutli-Core (s)	Cores enabled	Speed-up Factor
22.23	2.93	2	7.58
22.23	2.75	4	8.08
22.23	0.844	6	26.46

Table 1. CPU Time required for Max Code Coverage : Single core vs multi-threading with Multi-Core

Number of Cores	Multi-Processing (s)	Multi-Threading (s)	Speed-up Factor
2	6.46	2.93	2.204
4	5.7	2.75	2.072
6	5.97	0.844	7.07

Table 2. CPU Time required for Max Code Coverage: Multiprocessing vs multi-threading

Number of Cores	Multi-Processing (s)	Multi-Threading (s)	Speed-up Factor
2	97.12	22.19	4.37
4	102.10	15.67	6.509
6	197.07	1.267	84.51

Table 3. Wall Time required for Max Code Coverage: Multiprocessing vs multi-threading

speedup factor becomes even more noticeable. With 6-cores enabled and 6 fuzzing threads, the speedup factor is roughly 26x. One might ask why the speedup factor is greater than the the number of cores enabled. The reason is that because we have continuous synchronization of the inputs, cores start immediately benefiting from the work done by other cores.

3.4.2 Multi-threading vs Multi-processing.

In order to show the advantage of TwoFuzzingLong (TFL) over ClusterFuzz we also experimented with a multi-process approach where we deployed an independent fuzzing loop on each core. Similar to ClusterFuzz, there is a duration over which synchronization of inputs does not occur. We wanted to examine what the benefits were during this interval before synchronization takes place and hence it was not implemented in our testing.

Figure 2 shows the performance improvements based on CPU time, and Figure 3 shows the same algorithms run in the previous figure, but in wall time. With two cores enabled in the system, the multi-threaded approach showed a speedup factor (in CPU time) of about 2x over the multi-process approach. With 6 cores enabled in the system, the speedup factor was 7x. Furthermore, the speed up factor in wall time was almost 85x with all cores enabled. The speedup between these two approaches is once again attributed to the continuous synchronization, as it allows cores to build off one another and also avoid redundant coverage.

4 Future Work

4.1 Execution Time Fuzzing

To the best of our knowledge, there is currently no fuzzing tool that implements execution time as a fuzzing domain. American Fuzzy Lop (AFL) looks for coverage, SlowFuzz [5] looks for the longest code path, and perfFuzz [3] looks to maximize the execution count of basic blocks. Neither of these fuzzing domains will directly provide maximum execution time, since the runtime of a program can vary with different caching mechanisms and different instructions. As a result of these variances, it is not necessary that the longest code path provide the longest execution time.

We believe execution time is an important domain for fuzzing particularly when it comes to programs with real-time constraints. To be able to accurately determine the worst-case execution time (WCET) is critical for characterizing real time performance and reliability. In order to address this current gap in the capability of existing fuzzing tools, we have structured TwoFuzzingLong to collect program runtime as domain specific feedback. A user can switch to execution time as the fuzzing domain by providing a command line argument, similar to AFL. We have not yet examined the benefits of multi-threading with execution time as the fuzzing domain but believe this would be an interesting test to conduct.

4.2 Official Benchmarking

In order to prove the benefits of multi-threading over multi-processing formally, future work should include testing on benchmarks such as libpng, libarchive or openssl rather than testing on our own instrumented test program. This would require setting up automatic instrumentation, as well as using a real Linux machine, as opposed to a Virtual Machine. Because the number of branches we had was small, it is possible that the speedup would not be as prevalent with large test programs, and this test would help to measure the scalability of this idea.

5 Related Works

As mentioned previously, there do currently exist implementations of fuzzing tools that leverage parallelization. Namely AFL [7] provides the capability to deploy several fuzzing jobs simultaneously across a different cores in a system or several clusters in a fleet. By setting up a shared file, each fuzzing job is also able to synchronize its input periodically. This synchronization step achieves the same goal of multi-threading in TwoFuzzingLong, but is not as efficient because the synchronization only happens periodically and it uses a file which has more overhead for read/writes than a shared data structure.

ClusterFuzz [1], was another tool developed at google targeted towards implementing fuzzing at scale. It has a similar infrastructure to AFL's parallelization technique, where it's able to deploy several independent instances of fuzzing across a fleet of clusters. On a periodic basis, these inputs will be added back to the global corpus, and a distillation process occurs on a separate interval to remove the redundant inputs.

Another parallel fuzzing method ParallelFuzz (P-Fuzz) [6] is perhaps the most similar in implementation to TwoFuzzingLong. P-Fuzz does a better job than AFL and ClusterFuzz at maintaining synchronicity with other fuzzing processes but the implementation is targeted towards a large distributed computing system. P-Fuzz framework uses a database-centric architecture. There is a server, analogous to the queue we use in TwoFuzzingLong, which maintains a global set of fuzzing inputs. The server is then responsible for distributing the fuzzing tasks across all of its clients, while ensuring that each client does not access the same input. When a new input is determined the client will push the input back to the server, similar to how TwoFuzzingLong will add an interesting input back to the queue. Using the client-server model, P-Fuzz achieves better efficiency than AFL and ClusterFuzz, but the frequent client-server communication will carry a significant amount of overhead. This approach is well-suited for a large distributed system, but perhaps combining it with multi-threading could provide even greater performance results.

6 Conclusion

In this paper we looked at one of the most popular code testing method and proposed a way to improve its performance. The method we demonstrated involved using multi-threading as a method of parallelism instead of the multi-processing used today. Using a small test program on a virtual machine, our preliminary test results showed up to a 7 times speedup in CPU time, and and 84x speedup in wall time. This demonstrates the potential of how using a single shared queue to share the interesting inputs across fuzzers can result in a much more effective fuzzing process.

References

[1] Abhishek Arya. 2019. Clusterfuzz: fuzzing at Google Scale. https://i.blackhat.com/eu-19/Wednesday/eu-19-Arya-ClusterFuzz-Fuzzing-At-Google-Scale.pdf

[2] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. [n. d.]. Grammar-based Whitebox Fuzzing. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.187.7988&rep=rep1&type=pdf

[3] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[4] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. 2021. RiverFuzzRL - an open-source tool to experiment with reinforcement learning for fuzzing. https://doi.org/10.1109/ICST49551.2021.00055

[5] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[6] Congxi Song, Xu Zhou, Yin Qidi, Xinglu He, Hangwei Zhang, and Kai Lu. 2019. P-Fuzz: A Parallel Grey-Box Fuzzing Framework. *Applied Sciences* 9 (11 2019), 5100. https://doi.org/10.3390/app9235100

[7] Michał Zalewski. 2017. American fuzzy lop technical details. http://lcamtuf.coredump.cx/afl/technical_details.txt