CMPT 473 | SOFTWARE QUALITY ASSURANCE

Assignment III Project Report

NOVEMBER 21, 2015

SCHOOL OF COMPUTING SCIENCE, SIMON FRASER UNIVERSITY
Daphne Ordas, Matthew Chow, Young Yoon Choi (Justin), Wei Wang
Team DJW

Contents

Contact Information

Daphne Ordas	dordas@sfu.ca
Matthew Chow	mkc25 @sfu.ca
Young Yoon Choi (Justin)	yychoi@sfu.ca
Wei Wang	wwa53@sfu.ca

Software Specification

Name:	Gifsicle
Developers:	Eddie Kohler, Anne Dudfield, David Hedbor, Emil Mikulic, Hans Dinsen-Hansen
Stable release:	1.8.8 (Jul 1, 2015)
Written in:	C
Operating system:	Windows, Linux
Available in:	English, French
Type:	Utility
License:	Free open source license
Website:	https://github.com/kohler/Gifsicle

Gifsicle is a command-line tool for creating, editing, and getting information about GIF images and animations. Following are some sample commands:

Frame delay modification

`Gifsicle --delay=10 --loop *.gif > anim.gif`

Extracting frames from animations:

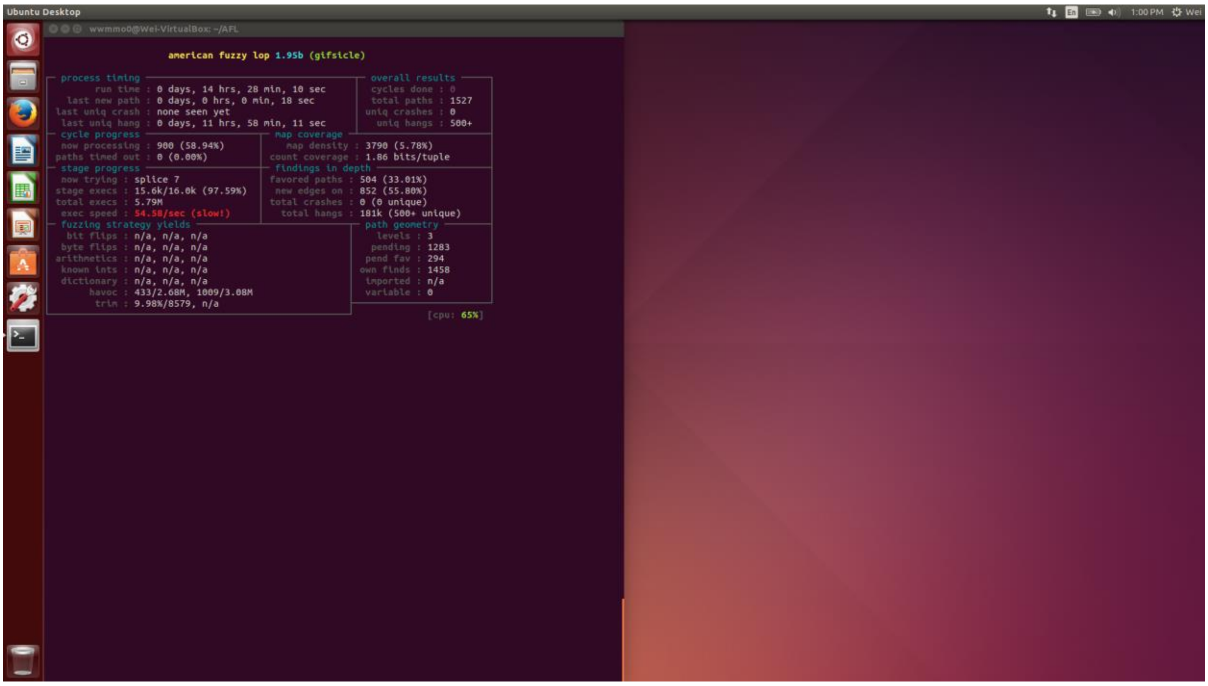
`Gifsicle anim.gif '#0' > firstframe.gif`

The original test subject is GNU Science Computation Library. However, after experimented with American Fuzzy Lop (AFL), the following concerns are added to require choosing a new test project:

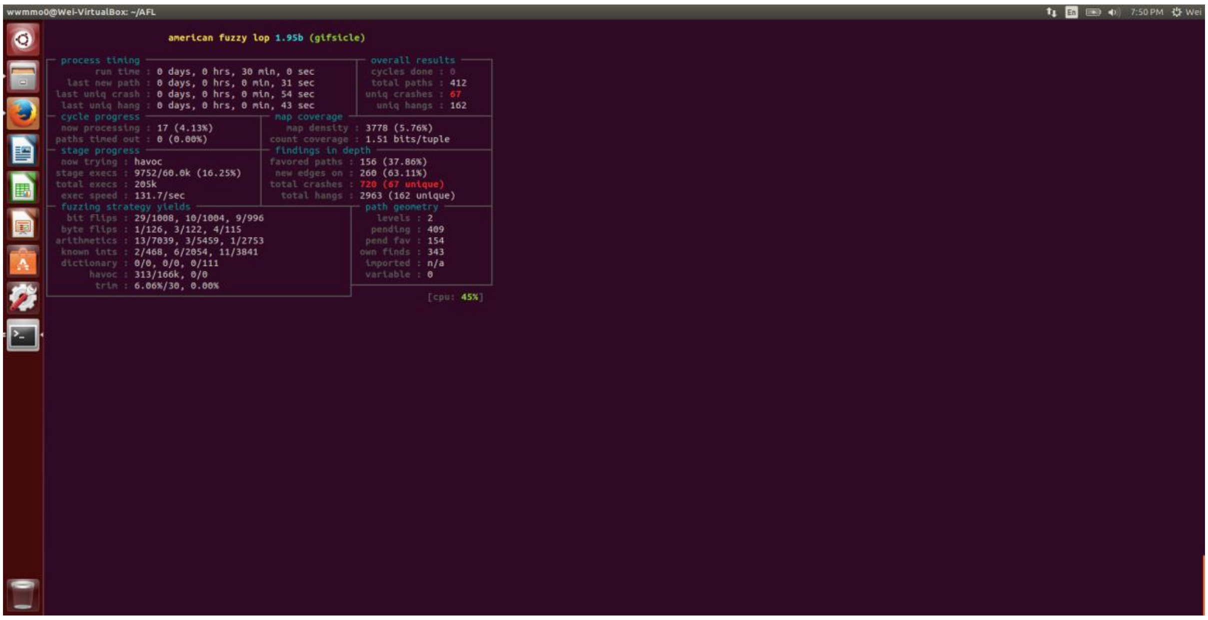
- The project shall handle some file input. This type of program can be multimedia player, database, format converter, compressor etc.
- The program shall conduct complex operation on the input. A good example can be optimize JPEG image. In contrast, a program practice mathematic computation may be regard as a bad example (The GNU Lib). Reasons are as follow:
    - Say the program compute integer addition. An arbitrary binary string can be interpreted as a number while any bits in it can be flipped which will not violate and interpretation.
    - In contrast, image compression require the program read and compute a pixel value in a strict format. Bit flip or trim can cause the format complete corrupted which much likely cause crushes.
- The subject shall implemented its own algorithm and libraries rather than use third party ones. The reason is since there are tons of personal or small scale project using well developed third party libraries to conduct routines, the fuzzing test is actually

testing the third party libraries rather than the subject program itself. For example, a media player decode XYZ (An uncommon one) video is much likely contain bugs compare with a general player using standard MPEG decoder library.

According to discussion above, dozens of programs are tested. These include libpng, poppler, redis, VLC, and other personal or small scale projects. AFL is configured with separate test cases so each team member can fuzz in parallel. The total accumulating fuzzing time is more than 60 hours and finally Gifsicle generate some crushes:



Example running generate nothing.



Example running on Gifsicle with generated crushes

Fuzzing Result

There are hundreds of crushes generated by the whole team. However, this report will focus on this specific instances. Results are as follow:

- Input Test Cases — Official GIF test cases from AFL image collection; Customized corrupted files
- Calling Command — **`Gifsicle -d 10 --optimize=10 --output=T.gif`**
  - -d — Delay between each frame is 10/1000 second
  - --optimize — Optimize each frame automatically
  - --output — Output file location
- Testing Command — **`$./afl-fuzz -i testcase_dir -o findings_dir -d -Q /Gifsicle -d 10 --optimize=10 --output=T.gif @@`**
  - -d — Dirty mode, save fuzzing time
  - -Q — For non-instrumented binary code
- Running Time — 0 day, 0 hours, 30 min, 0 sec
- Cycles Done — 0
- Total Paths — 412
- Total Crushes — 720
- Unique Crushes — 67
- Total Hangs — 2963
- Unique Hangs — 162

The fuzzing test begin with sample GIF files come from AFL. These collections are special designed file intentionally corrupted in a certain way so program reading them is much likely to generate a crush. The original purpose of these cases is for manually testing. However, the test team combine these input with few customized examples to accelerate the testing. Customized file include unmatched color channels, corrupted image size etc. AFL offered a user-friendly UI to monitor the test progress. Following anomalies are spotted during test:

- Unable to discover new test path. New test path means AFL spotted subject program generate error and will split the test flow afterwards. No newly discovered paths is majorly due to insufficiently designed test cases. Increase test case number will solve this problem.
- Extreme map density. According to manual, map density is a method to evaluate test coverage. Huge map density percentage is commonly due to the test subject is too complex while tiny one is in contrast. Both cases will dramatically reduce the test efficiency. Gifsicle has a proper one with around 5% density.
- Extreme low test execution speed. Depend on the subject program, testing complex UI such as VLC will reduce test execution speed below 50 cases/sec while other simple program can have an over 1000 cases/sec. To increase efficiency, test team reconfigured system resources to assign more memory and CPU cores to AFL.

Sanitizer Result

Sanitizer Usage

We used AddressSanitizer, ThreadSanitizer, and MemorySanitizer from Clang to attempt to analyze specific errors that caused the program to crash. To clarify, we grabbed the fuzzed inputs generated from AFL fuzzer that caused unique crashes. For example, to invoke the MemorySanitizer, the following commands were invoked:

```
CC = /usr/bin/clang-3.5
CCDEPMODE = depmode=gcc3
CFLAGS = -fsanitize=memory |-fno-omit-frame-pointer -g -02CPP =
/user/bin/clang-3.5 -E
```

Failure

The Clang sanitizer was able to compile successfully, but the program that was under test prevented the sanitizer from showing the backtraces for any errors found. Take for example the following code snippet:

```
int main(void) {char* a; return a[0]}
```

After running that code through our sanitizer, the result was as follows:

ASAN:SIGSEGV

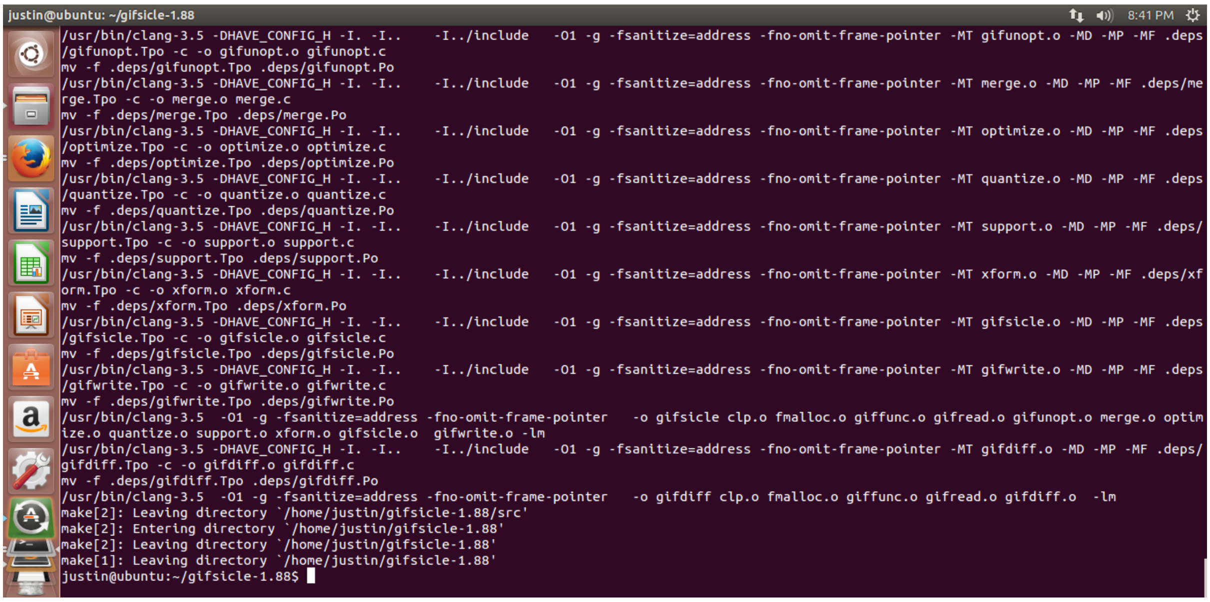
```
=================================================================
==2228== ERROR: AddressSanitizer: SEGV on unknown address
0x000000000000 (pc 0x000000400740 sp 0x7ffdb1b424a0 bp
0x7ffdb1b424b0 T0)
AddressSanitizer can not provide additional info.
    #0 0x40073f (/home/wwmmo0/T+0x40073f)
    #1 0x7fa5fefdbec4 (/lib/x86_64-linux-gnu/libc-2.19.so+0x21ec4)
    #2 0x400638 (/home/wwmmo0/T+0x400638)
==2228== ABORTING
```

Our sanitizer was able to find a memory leak in the code snippet, but when running the sanitizer with Gifsicle, there were too many assertions that would block the program, and Gifsicle would abort itself after encountering erroneous input, resulting in the sanitizer not being able to analyze what the crashes were about. Gifsicle would return its own error output while we attempted to insert over 200 unique fuzzed inputs that were guaranteed to crash the program.

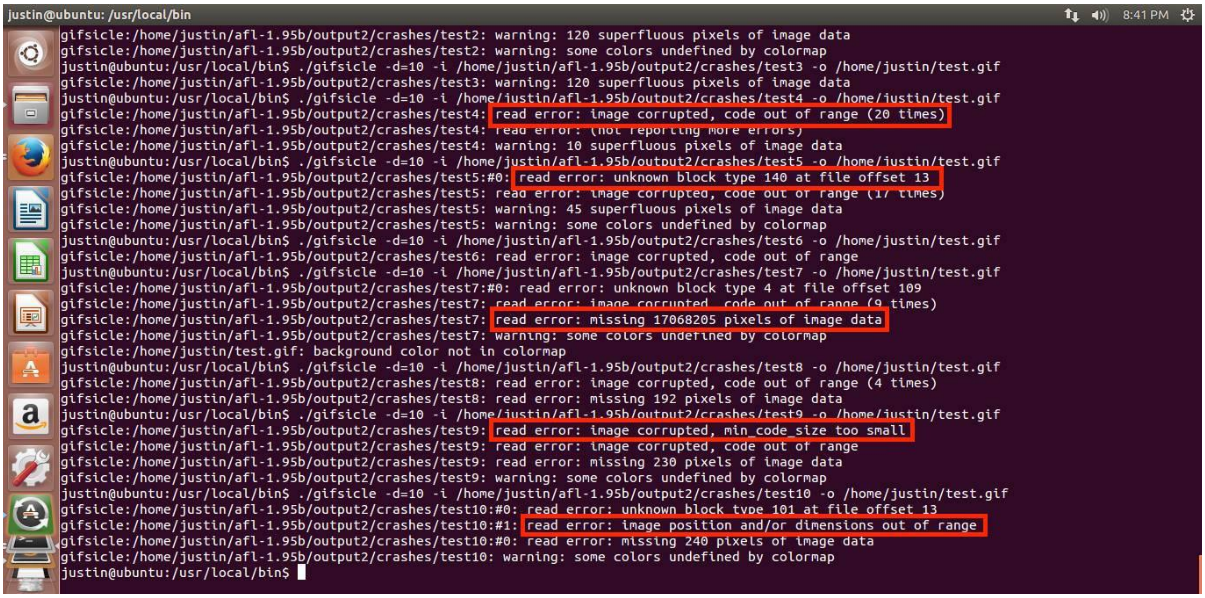
An assertion is a statement that is always expected to evaluate true at that point in code. If the assertion evaluates to false, the program will either crash or throw an exception. While assertions are useful when writing, testing, and debugging code, they are often not included or turned off in production builds of a program. This is because every assertion statement has to be evaluated and a large number of them can cause performance hindrances. Although Gifsicle was able to exit safely upon invalid input, it is better in practice to catch these errors by way of error handling in which a user friendly message can be displayed. It may also improve the performance of Gifsicle by reducing overhead.

From the screenshots below, it is clear that after inserting the erroneous input from the fuzzer, the program under test would crash and return output. This output was not returned from the sanitizer, but from the fuzzer. We will describe a subset of interesting error cases from these output and describe the cause of program failure. In general, the main types of errors that are present and are of interest are image corruption, missing pixels, unknown block type, and image position/dimensions out of range.

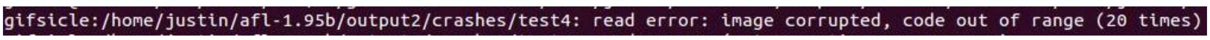
Applying Address Sanitizer



Executed make by setting CC and CFLAGS to be clang and AddressSanitizer, responsively



Assertions from project. Unique read errors and warnings are highlighted.

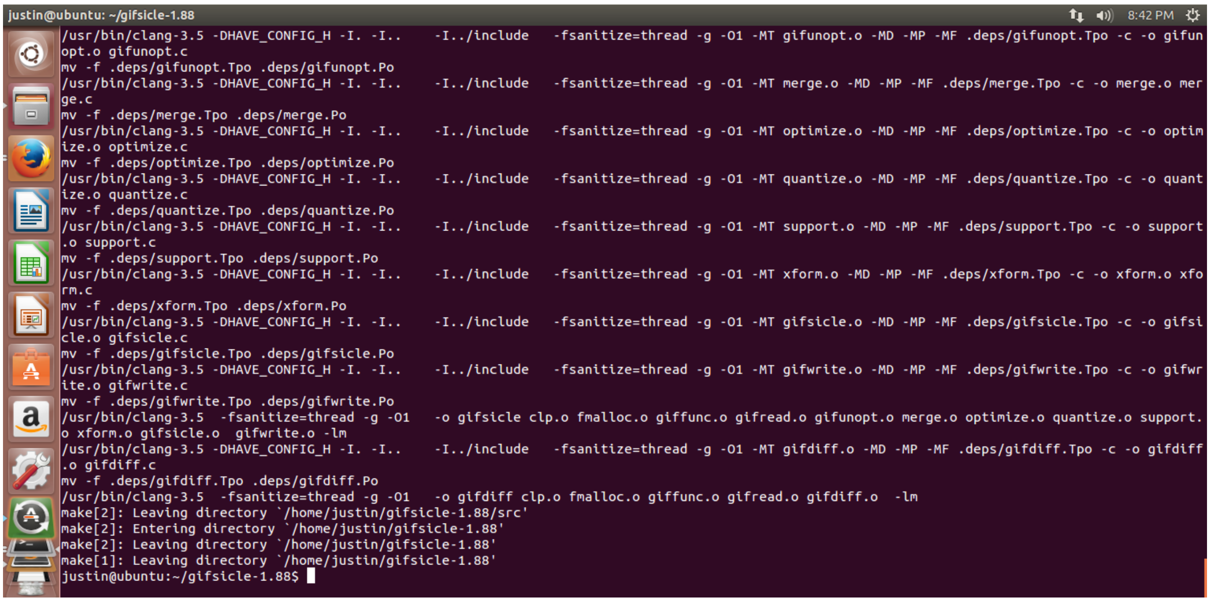


The above is an example of a image corruption error found from the output files of the afl fuzzer, when applied to Gifsicle. Based on the output of Gifsicle, after applying the fuzzed input files, this unique crash happened because of a read error, where the input image was corrupted, thus sending the code out of range.

Another example taken from this screenshot is where, after applying the fuzzed input files, this crash happened because the input image was missing pixels of image data:

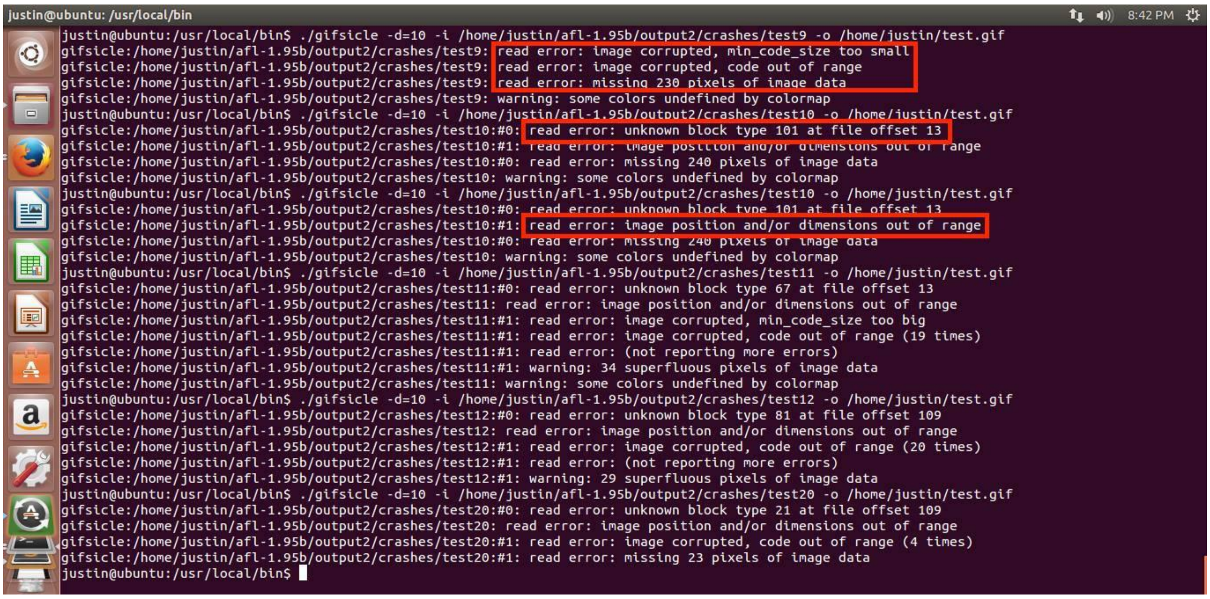
```
gifsicle:/home/justin/afl-1.95b/output2/crashes/test7: read error: missing 17068205 pixels of image data
```

Applying Thread Sanitizer



```
justin@ubuntu: ~/gifsicle-1.88
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT gifunopt.o -MD -MP -MF .deps/gifunopt.Tpo -c -o gifunopt.o gifunopt.c
mv -f .deps/gifunopt.Tpo .deps/gifunopt.Po
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT merge.o -MD -MP -MF .deps/merge.Tpo -c -o merge.o merge.c
mv -f .deps/merge.Tpo .deps/merge.Po
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT optimize.o -MD -MP -MF .deps/optimize.Tpo -c -o optimize.o optimize.c
mv -f .deps/optimize.Tpo .deps/optimize.Po
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT quantize.o -MD -MP -MF .deps/quantize.Tpo -c -o quantize.o quantize.c
mv -f .deps/quantize.Tpo .deps/quantize.Po
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT support.o -MD -MP -MF .deps/support.Tpo -c -o support.o support.c
mv -f .deps/support.Tpo .deps/support.Po
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT xform.o -MD -MP -MF .deps/xform.Tpo -c -o xform.o xform.c
mv -f .deps/xform.Tpo .deps/xform.Po
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT gifsicle.o -MD -MP -MF .deps/gifsicle.Tpo -c -o gifsicle.o gifsicle.c
mv -f .deps/gifsicle.Tpo .deps/gifsicle.Po
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT gifwrite.o -MD -MP -MF .deps/gifwrite.Tpo -c -o gifwrite.o gifwrite.c
mv -f .deps/gifwrite.Tpo .deps/gifwrite.Po
/usr/bin/clang-3.5  -fsanitize=thread -g -O1   -o gifsicle clp.o fmalloc.o giffunc.o gifread.o gifunopt.o merge.o optimize.o quantize.o support.o xform.o gifsicle.o  gifwrite.o -lm
/usr/bin/clang-3.5 -DHAVE_CONFIG_H -I. -I..    -I../include   -fsanitize=thread -g -O1 -MT gifdiff.o -MD -MP -MF .deps/gifdiff.Tpo -c -o gifdiff.o gifdiff.c
mv -f .deps/gifdiff.Tpo .deps/gifdiff.Po
/usr/bin/clang-3.5  -fsanitize=thread -g -O1   -o gifdiff clp.o fmalloc.o giffunc.o gifread.o gifdiff.o  -lm
make[2]: Leaving directory `/home/justin/gifsicle-1.88/src'
make[2]: Entering directory `/home/justin/gifsicle-1.88'
make[2]: Leaving directory `/home/justin/gifsicle-1.88'
make[1]: Leaving directory `/home/justin/gifsicle-1.88'
justin@ubuntu:~/gifsicle-1.88$
```

Executed make by setting CC and CFLAGS to be clang and ThreadSanitizer, responsively



```
justin@ubuntu: /usr/local/bin
justin@ubuntu:/usr/local/bin$ ./gifsicle -d=10 -i /home/justin/afl-1.95b/output2/crashes/test9 -o /home/justin/test.gif
gifsicle:/home/justin/afl-1.95b/output2/crashes/test9: read error: image corrupted, min_code_size too small
gifsicle:/home/justin/afl-1.95b/output2/crashes/test9: read error: image corrupted, code out of range
gifsicle:/home/justin/afl-1.95b/output2/crashes/test9: read error: missing 230 pixels of image data
gifsicle:/home/justin/afl-1.95b/output2/crashes/test9: warning: some colors undefined by colormap
justin@ubuntu:/usr/local/bin$ ./gifsicle -d=10 -i /home/justin/afl-1.95b/output2/crashes/test10 -o /home/justin/test.gif
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10:#0: read error: unknown block type 101 at file offset 13
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10:#1: read error: image position and/or dimensions out of range
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10:#0: read error: missing 240 pixels of image data
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10: warning: some colors undefined by colormap
justin@ubuntu:/usr/local/bin$ ./gifsicle -d=10 -i /home/justin/afl-1.95b/output2/crashes/test10 -o /home/justin/test.gif
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10:#0: read error: unknown block type 101 at file offset 13
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10:#1: read error: image position and/or dimensions out of range
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10:#0: read error: missing 240 pixels of image data
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10: warning: some colors undefined by colormap
justin@ubuntu:/usr/local/bin$ ./gifsicle -d=10 -i /home/justin/afl-1.95b/output2/crashes/test11 -o /home/justin/test.gif
gifsicle:/home/justin/afl-1.95b/output2/crashes/test11:#0: read error: unknown block type 67 at file offset 13
gifsicle:/home/justin/afl-1.95b/output2/crashes/test11: read error: image position and/or dimensions out of range
gifsicle:/home/justin/afl-1.95b/output2/crashes/test11:#1: read error: image corrupted, min_code_size too big
gifsicle:/home/justin/afl-1.95b/output2/crashes/test11:#1: read error: image corrupted, code out of range (19 times)
gifsicle:/home/justin/afl-1.95b/output2/crashes/test11:#1: read error: (not reporting more errors)
gifsicle:/home/justin/afl-1.95b/output2/crashes/test11:#1: warning: 34 superfluous pixels of image data
gifsicle:/home/justin/afl-1.95b/output2/crashes/test11: warning: some colors undefined by colormap
justin@ubuntu:/usr/local/bin$ ./gifsicle -d=10 -i /home/justin/afl-1.95b/output2/crashes/test12 -o /home/justin/test.gif
gifsicle:/home/justin/afl-1.95b/output2/crashes/test12:#0: read error: unknown block type 81 at file offset 109
gifsicle:/home/justin/afl-1.95b/output2/crashes/test12: read error: image position and/or dimensions out of range
gifsicle:/home/justin/afl-1.95b/output2/crashes/test12:#1: read error: image corrupted, code out of range (20 times)
gifsicle:/home/justin/afl-1.95b/output2/crashes/test12:#1: read error: (not reporting more errors)
gifsicle:/home/justin/afl-1.95b/output2/crashes/test12:#1: warning: 29 superfluous pixels of image data
justin@ubuntu:/usr/local/bin$ ./gifsicle -d=10 -i /home/justin/afl-1.95b/output2/crashes/test20 -o /home/justin/test.gif
gifsicle:/home/justin/afl-1.95b/output2/crashes/test20:#0: read error: unknown block type 21 at file offset 109
gifsicle:/home/justin/afl-1.95b/output2/crashes/test20: read error: image position and/or dimensions out of range
gifsicle:/home/justin/afl-1.95b/output2/crashes/test20:#1: read error: image corrupted, code out of range (4 times)
gifsicle:/home/justin/afl-1.95b/output2/crashes/test20:#1: read error: missing 23 pixels of image data
justin@ubuntu:/usr/local/bin$
```

Assertions from project. Unique read errors and warnings are highlighted.

```
gifsicle:/home/justin/afl-1.95b/output2/crashes/test10:#0: read error: unknown block type 101 at file offset 13
```

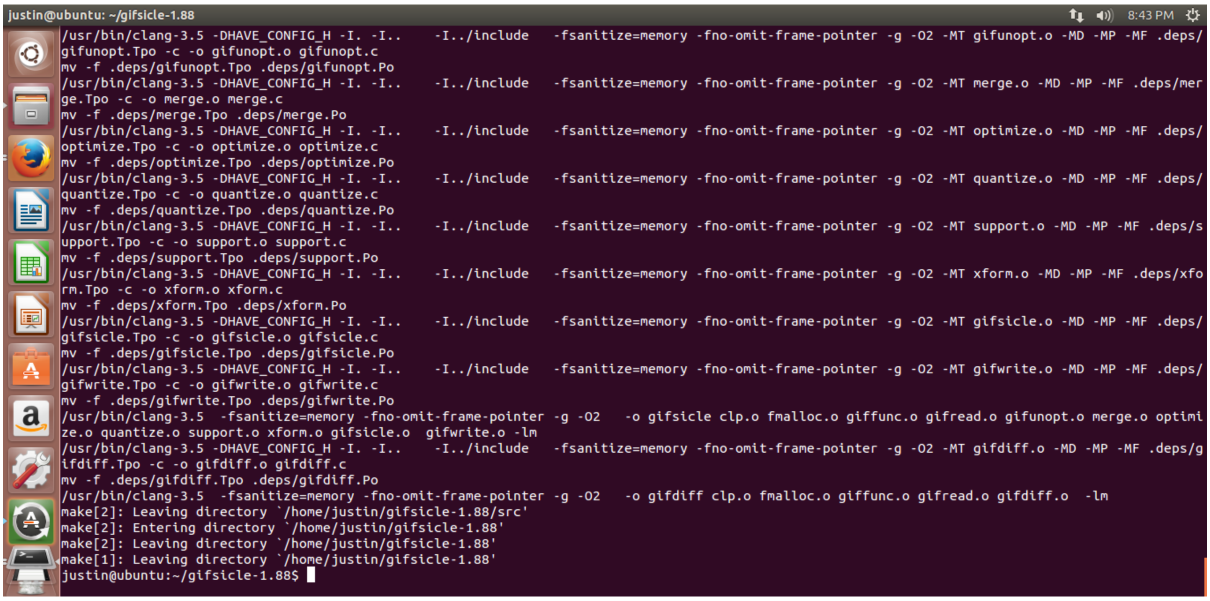
The above example shows an unknown block type error, which caused the program under test to fail and crash after a fuzzed input from AFL was put into the program.

Another cause of failure for the program was a certain fuzzed input that caused this specific error to be produced:

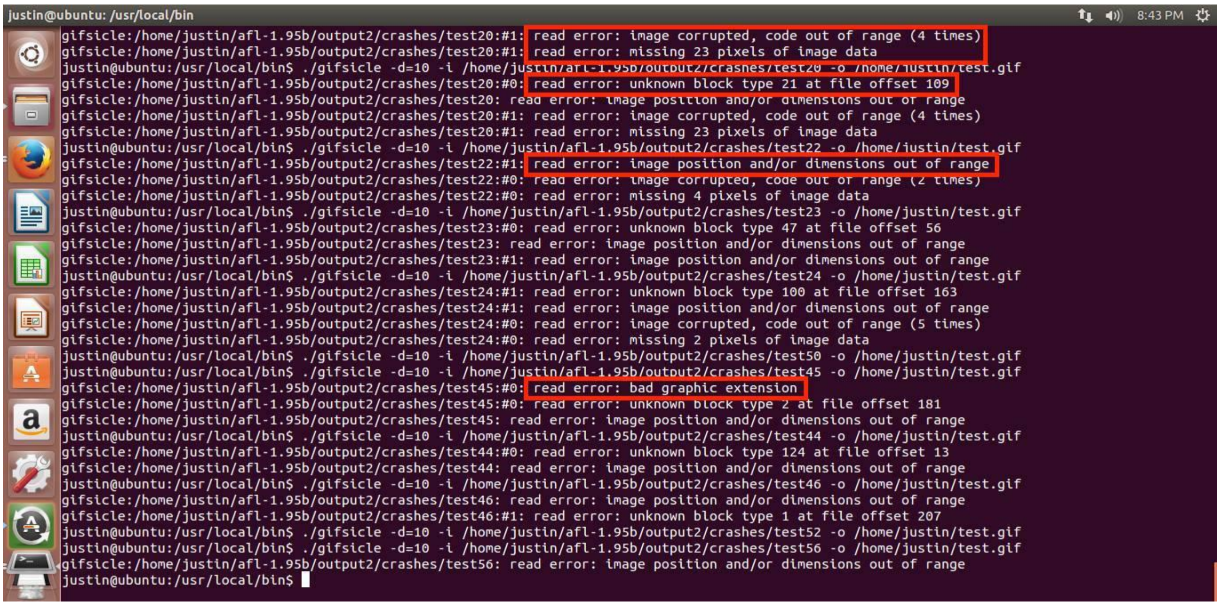
```
gifsicle:/home/justin/afl-1.95b/output2/crashes/test12: read error: image position and/or dimensions out of range
```

The proposed cause of failure was that the image position or dimensions were out of range, and because the sanitizer wasn't working properly with the project it was hard to determine whether it was the image position or the image dimension that caused the failure.

Applying Memory Sanitizer



Executed make by setting CC and CFLAGS to be clang and MemorySanitizer, responsively



Assertions from project. Unique read errors and warnings are highlighted.

All in all, the types of errors that were commonly encountered are as follows:

- read error: image corrupted, code out of range
- read error: unknown block type at file offset
- read error: missing pixels of image data
- read error: image corrupted, min_code_size too small

With only one occurrence of:

- read error: bad graphic extension