CS412 Fuzzing Lab Report

Lars Waldvogel, Franziska von Albedyll, Emanuel Mairoll, Nives Križanec

April & May 2025

Abstract

To get familiar with the existing fuzzing setup of `binutils`, we analyzed the `addr2line` harness with and without a seed corpus. As expected, the run with corpus achieved slightly higher coverage. Afterwards, we identified two significant uncovered regions of the code base, namely parts of `objcopy` implementing its various options and `strings`. Although harnesses for both tools already existed we found that that they were not complete in case of `objcopy` and incorrectly implemented in case of `strings`. Next, we improved the existing fuzzers to increase the coverage. Among other things, the `objcopy` harness gained functionality to fuzz the program flags, not just the files they operate on. The improvements to the `strings` fuzzer were even more substantial since the original harness did almost not work at all. In the end, our new and improved fuzzers were able to find bugs that the previous fuzzing harnesses were not. The `strings` fuzzer found a bug. Although it cannot be exploited it would be possible to cause a DoS in very limited cases.

1. Introduction

Students

- Lars Waldvogel (404889)
- Franziska von Albedyll (404193)
- Emanuel Mairoll (404912)
- Nives Križanec (385796)

Chosen Project

`binutils` is a large suite of tools for the analysis of binary programs. It contains `ld`, `objcopy`, `strings`, and many more widely-used utilities. A large part of these tools is used to discover information about a given program without necessarily executing it. Such a program might be provided by a malicious actor (e.g. when analyzing malware). If a flaw in these tools were found, this would have grave consequences. The impact might range from making the investigation of a malicious binary more difficult up to gaining remote code execution on an analyst's machine. To reduce these risks, fuzzers are employed to find potential vulnerabilities. In our work, we discovered that the pre-existing fuzzers are not perfect and that we could still improve upon them.

2. Part 1

2.1. Scripts to run harness with and without seed corpus

The OSS-Fuzz integration for the `binutils` project[1] builds 27 fuzzing harnesses targeting different tools within the suite, such as `as`, `nm`, `objdump`, and `readelf`. For Part 1, we picked the harness `fuzz_addr2line`.

To run `fuzz_objdump` with the provided seed corpus, we used the following steps (to reproduce, simply run `run.w_corpus.sh` in an empty folder):

1. Build the `binutils` image
2. Build the `binutils` fuzzers
3. Run the fuzzer for 4 hours (will stop by itself)
4. Build fuzzers with coverage instrumentation
5. Create coverage report

To run `fuzz_addr2line` without a seed corpus, we followed these steps (this is done by executing the script `run.w_o_corpus.sh`):

1. Apply patch `binutils_no_corpus.diff` to modify the build script to not use the pre-existing corpus
2. Build the `binutils` image
3. Build the `binutils` fuzzers
4. Run the fuzzer for 4 hours (will stop by itself)
5. Build fuzzers with coverage instrumentation
6. Create coverage report

The only difference between the two scripts is that in `run.w_o_corpus`, we prevent the project's `build.sh` script from loading available seed corpora. The `diff` can be found in the appendix under Part 1 Git Diff.

2.2. Discussion of differences between runs with and without corpus

2.2.1. Chosen harness:

For this part, we decided to run a `addr2line` harness. `addr2line` takes hexadecimal addresses as input and translates them into file name and line number information using debugging data from an executable file.

The `addr2line` fuzzing harness focuses on testing binary file parsing functionality. As a first step, it saves

[1] GNU Binutils: https://www.gnu.org/software/binutils/

the fuzzing input to a binary file into `/tmp`. The fuzzer then starts `addr2line` with six different, but fixed addresses (some valid, some malformed).

2.2.1.1. Differences in branch coverage over time

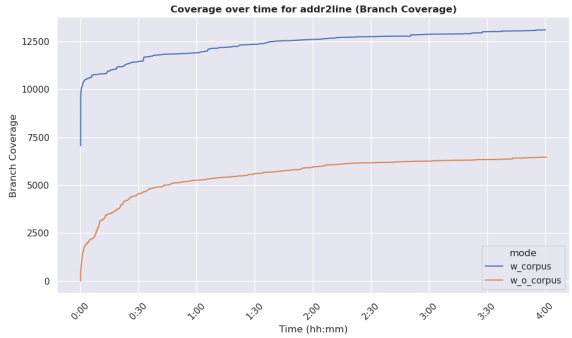

Figure 1: Coverage over time for `addr2line`[2]

Figure 1 shows that providing a corpus had a clear impact on performance. `w_corpus` starts at a coverage level of around 7200 branches, while `w_o_corpus` begins at 0, as it had no prior input help. Both runs show a rapid increase in the first 15 minutes, followed by a gradual slowdown. The final branch coverage of w_o_corpus remains below the initial coverage of `w_corpus`.

2.2.1.2. Difference in newly found functions over time

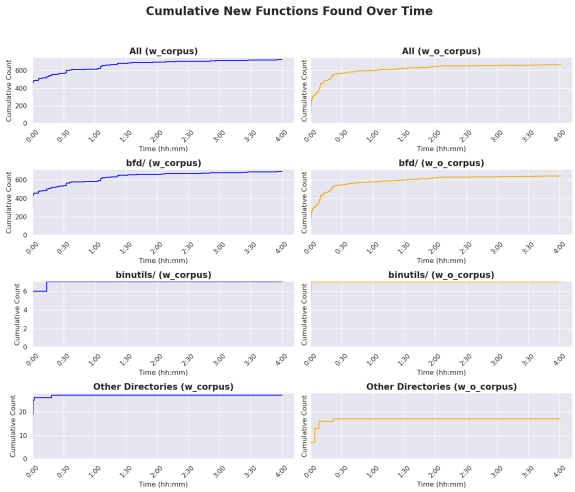

Figure 2: Found functions over time for `addr2line`[2]

Figure 2 shows that the overall count of functions found over time is similar, with the `w_corpus` run discovering slightly more functions. Most functions are found in the `bfd/` discovery. There is no meaningful difference in how many functions were found in `addr2line` itself.

Both runs show a relatively steep increase in discovered functions during the early phase, with the `w_corpus` run rising less sharply but continuing to increase over a longer period. In `bfd/`, this results in a later plateau compared tow_o_corpus.

The concentration of discovered functions in `bfd/` is expected, as `addr2line` relies heavily on the `bfd` library for core functionality such as symbol resolution and object file parsing, whereas the code in `binutils/` mostly handles command-line interface logic.

Coverage	Without Corpus	With Corpus	Δ (%)	Δ (Lines)
Line	23.82% (207/869)	25.20% (219/869)	1.38	12
Function	26.83% (11/41)	26.83% (11/41)	0	0
Region	23.45% (140/597)	24.79% (148/597)	1.34	8

Table 1: Coverage for `binutils/`

The coverage for the runs with and without corpus is nearly identical in `src/binutils-gdb/binutil`, with the only difference observed in `fuzz_addr2line.h`, where the run without corpus achieves slightly lower line coverage (-1.38%) and region coverage (-1.34%). These differences are limited to two regions:

Region 1

```c
148       if (symcount == 0
149           && ! dynamic
150           && (storage =
    bfd_get_dynamic_symtab_upper_bound
    (abfd)) > 0)
151         {
152           free (syms);
153           syms = xmalloc (storage);
154           symcount =
    bfd_canonicalize_dynamic_symtab (abfd,
    syms);
155         }
```

This code block is part of the function `slurp_symtab` (lines 117-164), which reads in the symbol table from the given binary file. This section handles the case where canonicalization of the regular symbol table succeeded but resulted in zero symbols.

For this section to be reached, we need very specific conditions:

- Have regular symbol tables that exist
- But after canonicalization result in zero symbols

[2]Enlarged version can be found in the appendix

- Haven't already tried dynamic symbols
- And have dynamic symbol tables

These conditions are difficult to reach with fuzzed binary files and might take longer to occur when fuzzing without a seed corpus.

Region 2

```c
316 {
317   const struct elf_backend_data *bed =
    get_elf_backend_data (abfd);
318   bfd_vma sign = (bfd_vma) 1 << (bed-
    >s->arch_size - 1);
319
320   pc &= (sign << 1) - 1;
321   if (bed->sign_extend_vma)
322     pc = (pc ^ sign) - sign;
323 }
```

This code block is part of the function `translate_addresses` (lines 286-421), which converts memory addresses into file name and line number information. This section specifically handles ELF-format address normalization. A possible reason for the zero coverage in the run without a corpus might be the absence of binary files in ELF format. Without such files, the condition `bfd_get_flavour(abfd) == bfd_target_elf_flavour` never evaluates to `true`.

Coverage statistics and a brief discussion for `bfd/` can be found in the appendix.

3. Part 2

An important thing to note before examining the OSS-Fuzz introspector coverage report is that the fuzzed `binutils` `.c` source files are converted into `.h` header files during the harness build process[3]:

```sh
1  # Patching. First do readelf.
2  cp ../../fuzz_*.c .
3  sed 's/main (int argc/old_main (int
   argc, char **argv);\nint old_main (int
   argc/' readelf.c >> readelf.h
4
5  # Special handling of dlltool
6  sed 's/main (int ac/old_main32 (int ac,
   char **av);\nint old_main32 (int ac/'
   dlltool.c > fuzz_dlltool.h
7
8  # Patch the rest
9  for i in objdump nm objcopy windres
   strings addr2line; do
10     sed -i 's/strip_main/strip_mian/g'
   $i.c
11     sed -i 's/copy_main/copy_mian/g'
   $i.c
12     sed 's/main (int argc/old_main32
   (int argc, char **argv);\nint old_main32
   (int argc/' $i.c > fuzz_$i.h
13     sed -i 's/copy_mian/copy_main/g'
   fuzz_$i.h
14 done
```

For example, the source file of `objdump` (`objdump.c`) will be compiled as `fuzz_objdump.h` and show up under this name in the OSS-Fuzz introspector. This is done due to the fact that its often necessary to apply patches to the original code, which is not possible if the file is not part of the same compilation unit. Patches are necessary in cases where we want to change the code such that it is possible to perform in-process fuzzing, e.g. we don't want to call `exit` or `abort`, but rather return back to the fuzzer.

With this knowledge we analyzed the coverage reports found on the `binutils` OSS-Fuzz introspector page and concluded that the following two significant regions were not covered.

3.1. Uncovered region 1: `objcopy` options

3.1.1. Explanation of shortcoming of existing harnesses

During a cross-reference of the existing harnesses with a list of all binary tools provided by `binutils` we noticed that `strip` seemingly has no harness yet. After further analysis we found that `strip` and `objcopy` are in essence the same program. The difference between the two is that for `strip` the source file `is-strip.c` is compiled while for `objcopy` the file `not-strip.c` is compiled. These two files only contain two preprocessing statements:

```c
1 #define is_strip x // x=1 in is-strip.c
  and x=0 in not-strip.c
2 #include "objcopy.c"
```

The crucial influence of the `#define is_strip` directive is on the following branching condition at the very end of the `main` function in `objcopy.c`:

```c
1 if (is_strip)
2   strip_main (argc, argv);
3 else
4   copy_main (argc, argv);
```

Functions `strip_main` and `copy_main` both handle the interpretation of options passed as command line arguments (both of them are essentially a huge `switch` statement) and call the relevant functions with the parsed option values. Comparing `strip_main`

[3] Fuzzer build.sh script, lines 84-96: https://github.com/google/oss-fuzz/blob/master/projects/binutils/build.sh#L84

and `copy_main` we concluded that the options and functionality of `strip_main` are roughly a subset of `copy_main`. To further verify this we confirmed that the options listed by `strip --help` are a strict subset of options listed by `objcopy --help`. Therefore fuzzing `objcopy` is sufficient to cover functionality of `strip` as well, although this approach could miss bugs within `strip_main` itself. Nevertheless, `strip_main` accounts for about 185 lines of code in `objcopy.c`, which contains in total 4176 lines of code[4]. This would imply that `strip_main` represents about 4.4% of the code. The other 95.6% should be either `objcopy` specific functions or functions used by both `strip` and `objcopy`. Due to this, we concluded that `objcopy` would be a more appropriate fuzzing target, as it can reach the majority of the code.

A fuzzer for `objcopy` already exists, namely `fuzz_objcopy.c`[5]. In OSS-Fuzz introspector, the reported coverage of `fuzz_objcopy.h` is 26.80%, or 1119 lines of code out of 4176. It is quickly noticeable looking at `copy_main`, that most of the cases in the large `switch` statement are never reached, therefore many of the specialized functions and conditional branches are never executed, hence not covered. But why are these `switch` cases never reached?

A look at the fuzzer revealed that it uses a static set of options:

```c
char *fakeArgv[12];
fakeArgv[0] = "fuzz_objcopy";
fakeArgv[1] = "-S";
fakeArgv[2] = "--decompress-debug-sections";
fakeArgv[3] = "--extract-dwo";
fakeArgv[4] = "--merge-notes";
fakeArgv[5] = "--pure";
fakeArgv[6] = "--debugging";
fakeArgv[7] = "--compress-debug-sections";
fakeArgv[8] = "--extract-symbol";
fakeArgv[9] = filename;
fakeArgv[10] = "/tmp/random.out";
fakeArgv[11] = NULL;
```

In OSS-Fuzz introspector we observed that these predefined options correspond directly with the covered cases of the large `switch` statement in `copy_main`. We concluded that by fuzzing the different options of `objcopy`, the coverage could be increased, not only within the `switch` case but the rest of the code as well. Here are a few examples of uncovered functions which can be attributed to the hard-coding of options in the fuzzing process:

- The function `add_specific_symbol` is not covered since the following options are never fuzzed: `--keep-symbol <name>`, `--strip-symbol <name>`, `--strip-unneeded-symbol <file>`, `--localize-symbol <name>`, `--globalize-symbol <name>`, `--keep-global-symbol <name>`, `--weaken-symbol <name>`.
- The function `add_specific_symbols` is not covered since the following options are never fuzzed: `--keep-symbols <file>`, `--strip-symbols <file>`, `--strip-unneeded-symbols <file>`, `--localize-symbols <file>`, `--globalize-symbols <file>`, `--keep-global-symbols <file>`, `--weaken-symbols <file>`.
- The function `init_section_add` is not covered since the following options are never fuzzed: `--add-section <name>=<file>`, `--update-section <name>=<file>`, `--dump-section <name>=<file>`.
- The function `section_add_load_file` is not covered since the following options are never fuzzed: `--add-section <name>=<file>`, `--update-section <name>=<file>`.
- The function `handle_remove_section_option` is not covered, since the following option is never fuzzed: `--remove-section <name>`.

3.1.2. Justification of significance

The available options `objcopy`, provide developers with a way to modify existing binaries without having to recompile the entire code base. This can be especially useful whenever the source code is not open-source or for reverse-engineering purposes. Additionally, having the ability to weaken or strengthen symbols as well as localize or globalize them, it makes it possible to modify the symbols in such a way that linking them would not result in conflicts.

`binutils` is implemented in C, which is infamous for being memory-unsafe. Therefore, it is likely that memory corruption bugs are present in the large codebase. If the programmers made a mistake, they might have inadvertently introduced a buffer overflow or a similar vulnerability. Such a vulnerability can potentially lead to arbitrary code execution. In the absence of memory corruption vulnerabilities, other common C bugs may be found such as improper initialization,

[4]The total number of lines was taken from the line coverage statistic shown in OSS-Fuzz introspector: https://storage.googleapis.com/oss-fuzz-coverage/binutils/reports/20250426/linux/src/binutils-gdb/binutils/report.html

[5]Objcopy fuzzer: https://github.com/google/oss-fuzz/blob/master/projects/binutils/fuzz_objcopy.c

scoping mistakes, unintended operator precedence, type confusion, etc. Conditional statements without curly braces as well as `goto` statements[6] are also common in `binutils`, increasing the likelihood of unintended paths in the control-flow.

3.2. Uncovered region 2: `strings`

3.2.1. Explanation of shortcoming of existing harnesses

Looking over the various other existing fuzzers, we found it curious that the `fuzz_strings` fuzzer achieves only 3.66% coverage of `fuzz_strings.h`. This accounts for only 32 lines of code out of 847. After further investigation, we found that the core functionality of `strings` was not actually being fuzzed at all.

The existing fuzzer found in `fuzz_string.c`[7] creates a temporary file using the fuzzing data and calls the main entry point of `strings -d`, namely `strings_object_file`. Using the `-d` flag assures the file is parsed using `bfd` and then only the initialized data section of the file is scanned for strings.

This is quite a short function, entirely covered by the fuzzer, of which the most relevant part is the invocation of the `strings_a_section` function:

```c
for (s = abfd->sections; s != NULL; s = s->next)
  strings_a_section (abfd, s, file, &got_a_section);
```

Naturally, we investigated further into what code is covered in `strings_a_section`. Interestingly, only the initial `if`-statements that check for error conditions are covered. This means that there must always be an error that is detected before the functions fully executes, hence the following function is never executed as indicated in the OSS-Fuzz introspector:

```c
print_strings (filename, NULL, sect->filepos, sectsize, (char *) mem);
```

The function `print_strings` is where the actual functionality of `strings` is implemented, i.e. the search for strings within the provided file and printing them. Since `print_strings` is not covered by the current fuzzer, this essentially means that we are never actually fuzzing the core functionality of the `strings` tool.

Initial experiments revealed that unlike the majority of the other fuzzers, the `fuzz_strings` fuzzer appears to not use an initial seed corpus. We further confirmed this by observing the absence of `strings` in the following lines of the harness build script[8]:

```sh
# Copy seeds out
for fuzzname in readelf_pef readelf_elf32_csky readelf_elf64_mmix readelf_elf32_littlearm readelf_elf32_bigarm objdump objdump_safe nm objcopy bfd windres addr2line dwarf; do
  cp $SRC/binary-samples/oss-fuzz-binutils/general_seeds.zip $OUT/fuzz_${fuzzname}_seed_corpus.zip
done
# Seed targeted the pef file format
cp $SRC/binary-samples/oss-fuzz-binutils/fuzz_bfd_ext_seed_corpus.zip $OUT/fuzz_bfd_ext_seed_corpus.zip
```

Following this revelation, we decided to test the performance of `fuzz_strings` with a simple seed file. Although other issues were revealed, which will be discussed shortly, the fuzzer was able to reach the previously uncovered `print_strings` function. However, it was stuck in an infinite loop printing blank lines. Thus, we figured that the fuzzer must have reached the function but never saved the seed, probably because it ran into a timeout. The saved coverage comes from generated binary files which were deemed invalid when parsed, leading to an early return.

In addition to the lack of a saved seed corpus, we found that the `fuzz_strings` fuzzer had not been implemented correctly. The fuzzer did strike us as oddly short compared to the others, and in fact a noticeable difference we found was that `fuzz_string` never calls any setup functions before invoking the entry point `strings_object_file` and it also never initializes any global variables. A look at the `main` function in the `strings.c` source code[9] revealed that there are in fact functions that need to be called as well as global variables that need to be initialized before the entry point is called in order to simulate the expected behaviour of `strings`. This discrepancy is ultimately what causes the fuzzer to get stuck printing the infinite number of empty strings found to the terminal.

3.2.2. Justification of significance

In general, the `strings` tool is used to extract printable character sequences from non-text files[10]. It is commonly used in reverse engineering and malware

[6] See e.g. the `goto fail;` vulnerability: https://www.blackduck.com/blog/understanding-apple-goto-fail-vulnerability-2.html

[7] Strings fuzzer: https://github.com/google/oss-fuzz/blob/master/projects/binutils/fuzz_strings.c

[8] Fuzzer build.sh script, lines 174-179: https://github.com/google/oss-fuzz/blob/master/projects/binutils/build.sh#L174

[9] Strings main function: https://github.com/bminor/binutils-gdb/blob/master/binutils/strings.c#L200

[10] Man page for `strings`: https://linux.die.net/man/1/strings

analysis to identify human-readable content within binary files. This can reveal information such as function names, error messages, format strings, as well as other strings that may reveal secret data, indicators of compromise and/or commands indicating malicious intent, such as connecting to remote servers or executing shell commands. Overall, `strings` can give insight into the purpose, the origin as well as the behavior of a binary.[11][12]. As students we have also commonly used `strings` for reverse engineering. It is one of the first tools one is introduced to when starting out with reverse engineering and a CTF player's best friend. We would argue that is one of the most famous utilities out there.

The bugs we previously described in `objcopy` can once again occur in `strings`, specifically in the case when the option `-d` to print only the data section is enabled. The usage of the flag is not unlikely, since the tools is popular among professionals and researchers who are likely to fiddle with various options during binary analysis. Whenever this flag is enabled, the `bfd` library of `binutils` is used to parse the file and identify different sections within it, similarly to what is done in `objcopy`. The reason why bugs are less likely without the flag is because in that case the code is much simpler and straightforward. The whole file is scanned only once as a whole and only `string_min` printable characters are stored in a buffer of appropriate size. To increase coverage and therefore the likelihood of discovering a bug we set `datasection_only = true` in the fuzzer, which corresponds to `strings` being invoked with the `-d` flag. This ensures that before searching for strings, `bfd` attempt to parse the file into sections, i.e. more code is covered overall.

4. Part 3

4.1. Uncovered region 1: `objcopy` options

4.1.1. Changes for the new harness

The original harness has the options given to `objcopy` hard-coded. This means it is impossible to reach all the code which requires these hard-coded options to be disabled or which need other options to be present. Instead of modifying the original harness, we chose to create a new harness. This was done primarily for two reasons: For one, this allowed us to simply create a new corpus with 256 bytes prepended. These 256 bytes in the beginning are used as the fake `argv` for `objcopy`. The other reason we did this is to preserve the higher speed of the original fuzzer for the region it can explore. Since the new fuzzer might potentially explore many invalid option names (especially in the beginning), it will not explore the originally reachable region as quickly. However, the new fuzzing harness clearly wins out in terms of theoretically achievable coverage, since it will eventually try all possible combinations of options. Thus, we end up with the best of both worlds: The original fuzzer keeps on fuzzing its region with raw speed, while the new harness casts a wider net for the parts it tries to cover.

We also did consider ways to fuzz without updating the corpus: For example, we could have generated the seed for the fake `argv` from some sort of hash over the data. While this would have worked, the fuzzer would have had a hard time purposefully going down a certain path in the domain of arguments. Changing a single byte might completely change the options. To avoid this, we opted to reserve a dedicated region in the corpus.

One of the main pain points with building a fuzzing harness for `objcopy` was the fact that it voluntarily exits in many places, which kills the in-process fuzzer. These crashes can be distinguished from proper crashes because they do not generate a crash report, unlike e.g. an `ASan` crash. The voluntary exits serve as false positive crashes and slow down the fuzzer since it has to be restarted. We tried to avoid this from happening by patching the source code to replace the `abort`, `exit`, `print_version`, `parse_vma` functions with stand-ins containing `longjmp` calls. If one of the listed functions were to be called, it would eventually stop execution of the original program. Now, in the places the unmodified `objcopy` were to crash, it instead jumps back to the main fuzzing loop (where we originally called `setjmp`), as if the run had terminated normally. This is done by using `#undefine` statements for the original functions imported from header files and defining the stand-in functions in the source files.

4.1.2. Possible further improvements

Letting it freely create the command line arguments also has a trade-off: It might accidentally overwrite existing files. For example, this could be done if it specified a path to an existing file as the output file. However, we assume the probability of it guessing a pre-existing filename is very slim, so we did not address this issue. If a clean solution to avoid this is found, it can easily be implemented into the harness. Another possible improvement would be to replace the `setjmp` and `longjmp` (which we use to avoid calling `abort`) with proper return chains. This would allow

[11] Reverse engineering a simple C program part 1: `strings`, `ltrace`, `hexdump`: https://1500bytes.blogspot.com/2018/04/reverse-engineering-simple-c-program.html

[12] Static Malware Analysis. Strings analysis: https://medium.com/@1200km/static-malware-analysis-e876640cfdb0

a proper cleanup of newly allocated memory without leaking it. Also, this can avoid some other unintended side effects of these notoriously dangerous functions. It should also be noted that the program still crashes sometimes. Patching out these last `abort` calls would further increase the effectiveness of the fuzzer by eliminating false positives, but this process is very tedious and would require adaption whenever `objcopy` is updated. The cleanest solution would probably be if the `objcopy` source files would avoid these `abort` altogether and always fully return up to the main function instead.

4.1.3. Reproducing our results

Running our script at `submission/part3/objcopy/run.fuzz_objcopy_options.sh` in a new folder will automatically reproduce our results. The script is self-contained and downloads all required files into the folder it is run in. It will also automatically stop after 4 hours.

4.1.4. Comparison

The new harness aims to target all regions that were previously unexplored because they were dependent on a certain option being present or absent. After running the old and the new harness each 3 times for four hours per run, we merged their corpora and produced a coverage report for each harness: The old harness achieved a line coverage of 26.91% (1137/4225), while the new harness got a line coverage of 43.57% (1841/4225). This confirms our improvement over the old harness. It was able to reach many functions that the old fuzzer could not find. Here is a selection of the newly reached functions in `fuzz_objcopy.h`:

- `copy_usage`
- `parse_flags`
- `parse_symflags`
- `eq_string_redefnode`
- `htab_hash_redefnode`
- `add_specific_symbol`
- `add_specific_symbol_node`
- `add_specific_symbols`

Additionally, it managed to reach large chunks of code that the original fuzzer was unable to explore. For example, the function `find_section_list` was called by the original harness, but most lines were never touched (e.g. because of early returns or never exploring an `if`-branch): In this analysis, we only looked at the first 1300 lines of the 6296 lines total in `fuzz_objcopy.h`. Even if we looked at less than a fourth of the total source code, we were already able to find many new covered regions. But it also has to be said that the new harness did not reach all code regions that the original fuzzer did. For example, the function `is_dwo_section` is never explored by the new fuzzer, but the old fuzzer managed to reach it. We have two theories as to why that is:

- The seeds were simply in favor of the older harness in this case
- Or the older fuzzer just reaches some regions faster because its search is more focused

Given enough time, our new fuzzer should be able to cover these regions too, since it fuzzes a superset of the functionality the original fuzzer did. Nevertheless, this also shows that it might be beneficial to keep both harnesses, as they have different strengths (see the discussion in Changes for the new harness).

4.2. Uncovered region 2: `strings`

4.2.1. Changes for the new harness

Looking at the original `fuzz_strings.c` fuzzer, we found it to be quite lacking in terms of coverage. It only achieved a line coverage of about 3% (31/847 lines) in `fuzz_strings.h`, which is surprising for a tool so popular. We started by checking the code for the `strings` fuzzer in the OSS-Fuzz repository and noticed a few things, which we addressed step by step.

The first thing we found was that, compared to the other fuzzers, the strings fuzzer was missing a seed corpus. We added the same seed corpus that we used for the other fuzzers (from the `DavidKorczynski/binary-samples`[13] repository). This simple addition already increased the coverage significantly.

After this, we ran the fuzzer again and found that it was able to reach the function `print_strings`. But instead of consistently printing the strings it was supposed to print, it quickly got stuck in a loop printing an infinite number of newlines. We had to look into the code to understand what was going on.

As already outlined in Part 2, the `strings` fuzzer was not properly initializing the global variables that would have been set in the `main` function. Most importantly, the global variable `string_min` was initialized to 0, which would mean that the fuzzer would search for strings of length 0. This would lead to the fuzzer printing an infinite number of empty strings, which would lead to the infinite loop we observed.

We fixed this by porting the initialization code from the `main` function, properly initializing all necessary global variables, and calling the `bfd_init` function to initialize the `bfd` library as well. Now we were able to reach the `print_strings` function without the

[13] Binary samples: https://github.com/DavidKorczynski/binary-samples

fuzzer getting stuck. However, running our first small fuzzing campaign, we quickly found yet another issue: The fuzzer would abort without writing an artefact, and sometimes get stuck on some inputs for a very long time. Both of these issues were not reproducible with the original `strings` binary.

Uncovering the last issue took us quite a while. After tinkering around a lot with the build flags, we fond that the `--enable-targets=all` flag was the culprit. This enables all targets (binary platforms, meaning binary file format + architecture) in the `bfd` library, which also enables more "unstable" backends, were we quickly found multiple bugs.

Some crashes were reported due to the library calling `abort`. The crash is invoked in `xsym.c`, lines 206-211:

```c
int
bfd_sym_read_header_v34 (bfd *abfd
  ATTRIBUTE_UNUSED,

  bfd_sym_header_block *header
  ATTRIBUTE_UNUSED)
{
  abort ();
}
```

It seems that this functionality is not implemented yet, which is why it aborts execution.

For the slow-units, we looked further into the `bfd` library to see what is actually happening. This has been a bit of a rabbit hole, which we discussed in Part 4.

We were able to counteract these crashes and timeouts by removing the build flag `--enable-targets=all` and applying a few other smaller changes in `build.sh` to make it compile successfully again.

On paper, disabling all targets could reduce the coverage somewhat. However, it is important to note that we are trying to fuzz uncovered lines in the `strings` program, but the targets are part of the `bfd` library, which already has more specialized fuzzers. Thus, there is little to be gained from having all targets enabled, but the fuzzer performance suffers from having them enabled. It follows that we lose only very little important coverage, but the fuzzer can now effectively work without crashing regularly.

Running it with all these changes, we were not only able to reach the `print_strings` function, but also a lot of other functions that were previously unreachable. We discuss the results in more detail in the upcoming section.

4.2.2. Possible further improvements

A lot of functionality regarding Unicode is still uncovered. To reach this functionality, one could initialize the `unicode_display` with a set of values. We have tried this to some success, but the result was not particularly stable. Thus we decided against shipping this experimental harness. In the future, somebody could properly implement this to work with all kinds of values assigned to the `unicode_display` variable. Another possible improvement is the patching out of `abort` instructions and replacing them with proper return chains. Doing so would end the fuzzer run without crashing the entire program if some unsupported functionality is reached. This would re-enable us to fuzz all targets and potentially reach higher coverage. Additionally, one could implement a way to fuzz command line options, similar to how we did it in `objcopy`. This could discover bugs in the parsing of the options[14] or find new combinations which cover new paths.

4.2.3. Reproducing our results

Running our script at `submission/part3/objcopy/run.fuzz_strings.sh` in a new folder will automatically reproduce our results. The script is self-contained and downloads all required files into the folder it is run in. It will also automatically stop after 4 hours.

4.2.4. Comparison

The original harness in OSS-Fuzz only explored 31/847 lines (3.66%). With the better initialization code we added, and by removing `--enable-targets=all`, our new harness was able to cover 112/847 lines (13.22%). Please keep in mind that the OSS-Fuzz project has been running for quite some time. The first commit to `fuzz_strings.c` was almost 4 years ago, we assume it has been running continuously since then. Our fuzzer only ran for 4 hours, with much better results.

Now it's actually able to reach the following new functions:

- `print_strings`
- `print_filename_and_address`
- `get_char`
- `STRING_ISGRAPHIC`

[14] Relevant vulnerability in command line option parsing: https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit

5. Part 4

5.1. Root cause analysis of the crash and fix proposal

The object file generated by the fuzzer using the `--enable-targets=all` flag causes the `strings` tool to print the same strings over and over again. This causes the fuzzer to timeout. In addition, `strings` continuously reports errors of the following form:

```
1 error: ../slow-unit-file(ed %u, f) is
  too large (0x2e7365 bytes)
2 ../slow-unit-file: Reading section ed
  %u, f failed: file truncated
3 error: ../slow-unit-file(tended a) is
  too large (0x65636f72 bytes)
4 ../slow-unit-file: Reading section
  tended a failed: file truncated
5 error: ../slow-unit-file( length ) is
  too large (0x20486578 bytes)
6 ../slow-unit-file: Reading section
  length  failed: file truncated
```

These errors already give us a hint as to what is happening. Firstly, we note that `strings` is telling us it is trying to read different sections of the file. It would be our expectation that `strings` would try to print only one section, namely the `.data` section, since the fuzzer sets `datasection_only = true`. If we take a look at the sections names `'ed %u, f'`, `'tended a'`, `' length '`, they appear strange and malformed, i.e. incorrect. Secondly, the error messages tell us that these sections are way too large. Curiously, the size of the buggy object file is 900KiB, while the reported section sizes are 2973KiB, 1622MiB and 517MiB respectively. An attempt at reproducing this behaviour with the default `strings` build was unsuccessful. We will come back to both of these points later on in the analysis.

By not including the flag, the resulting `strings` would behave the same as standard `strings`. Having access to both versions of the tool unstripped, we examined them in `gdb` with the buggy file as input and searched for the divergence in execution. At the top level, the execution path diverged within `string.c` in the function `strings_file` on the following condition:

c

```c
1 if (!datasection_only || !
  strings_object_file (file))
2 {
3   FILE *stream = fopen (file,
  FOPEN_RB); ...
4   print_strings (file, stream,
  (file_ptr) 0, 0, (char *) NULL); ...
5 }
```

Clearly, if `datasection_only` is set to `false`, the function `print_strings` is called only a single time, with the entire file as input. This was confirmed also by `gdb` when we executed `strings` without the `-d` flag. In the presence of the flag, we observer that the standard `strings` tool would similarly enter the `if` block and call `print_strings` only once. Therefore, `strings_object_file` returned `false`. On the other hand, the version of `strings` built with `--enable-targets=all` did not execute the `if` block, implying `strings_object_file` in this case returned `true`. The function `strings_object_file` internally performs the following steps:

c

```c
1   abfd = bfd_openr (file, target);
2   if (abfd == NULL) return false;
3
4   if (!bfd_check_format (abfd,
  bfd_object)) // side effect of reading
  in the sections
5   {
6     bfd_close (abfd);
7     return false;
8   }
9
10  for (s = abfd->sections; s != NULL; s
  = s->next)
11    strings_a_section (abfd, s, file,
  &got_a_section);
```

In the standard `strings` tool, we found that the function `bfd_check_format` would return `false`, and consequentially so would the function `strings_object_file`. In the `--enable-targets=all` build, the function `bfd_check_format` would return `true`, and continue with the execution of the for loop below. Without going into too much detail, `strings_a_section` internally checks that the section is a data section, that its size is larger than 0 and that the reading of the section into memory was successful before ultimately calling `print_strings` providing a section of the file as input instead of the whole file. With the help of `gdb` we further found that:

- `strings_file` function would get invoked once,
- `strings_a_section` function would get invoked 9664 times, and
- `print_strings` function would get invoked 551.

The reduction in number of calls from `strings_a_section` to `print_strings` is due to the aforementioned checks done within `strings_a_section`. One of these is also responsible for the error messages we observed, namely:

c

```c
1 if (!bfd_malloc_and_get_section (abfd,
  sect, &mem))
```

```
2 {
3   non_fatal (_("%s: Reading section %s
  failed: %s"),
4     filename, sect->name, bfd_errmsg
  (bfd_get_error ()));
5   return;
6 }
```

Next, we decided to take a closer look at the sections that were being found in the file. Following is a very small subset:

```
1 name:uiltin_e, size:0x6e5f7465,
  filepos:0x6d705f65
2 name:node, size:0x70697065,
  filepos:0x2d66696c
3 name:op_redir, size:0x65786563,
  filepos:0x7574655f
4 name:, size:0x0, filepos:0x0
5 name:, size:0xffffffff, filepos:0x0
6 name:, size:0x0, filepos:0x0
```

All of sections are clearly non-coherent, indicated not just by their nonsensical names but also by the fact that many of them are way too large and overlapping. On a side note, `size` and `filepos` are both unsigned integers, which makes sense, since in a valid file they would never be negative. But there appear to no upper bounds imposed. Considering there are 9664 sections found and 551 of them pass all checks and are deemed as "valid" data sections, it is no wonder that the same strings get found and printed to the terminal repeatedly. Indeed if we only look at the "valid" sections, they are overlapping:

```
1 name:, size:0xbb480, filepos:0x14
2 name:, size:0xd61d4, filepos:0x8
3 name:, size:0x77ac0, filepos:0x54
```

Although why are so many sections being found? To find the answer, we went back to the `bfd_check_format`. We found that it tries to identify the target architecture and uses the architecture specific parser to identify distinct sections. In standard `strings` no target architecture is identified as a match, due to the fact it has a reduced set of 18 targets that it can match against. Enabling all targets expanded the set to 260 possible targets. One of these additional targets, which our particular buggy file matched, is `aixcoff-rs6000`.

With further examination of the section parser of the `aixcoff-rs6000` architecture and a lot more knowledge of the expected file structure we might be able to find exactly where the implementation is faulty. We believe that the parser relies on `TLV` (type, length, value) encoding, but does not validate the plausibility of identified sections. This would immediately explain the incoherent names of sections, their impossibly large sizes compared to the file size and the fact that most sections have significant overlap.

The solution we propose is to find add sanity checks within the parser that would identify these errors and reject the object file. First, based on the architecture standard, the section names may be verifiable. Second, an upper limit on the section size could be set at the file size. Third, a sweep-line approach could be used to efficiently identify overlapping sections. After the application of this fix, `strings` with all targets enabled would behave exactly the same as when they are disabled, since `aixcoff-rs6000` was the only architecture that was identified as a possible match among all 260 targets.

5.2. Analysis of the severity of the crash

The bug is only reachable with the usage of the `-d` option with a specifically built `strings` tool configured to have all targets enabled, which includes the problematic architecture `aixcoff-rs6000`. The commonly distributed `strings` binary is therefore not impacted. This limits the real-world feasibility of exploiting the bug. Nevertheless, although limited it could have an impact on systems and individuals dealing with exotic architectures that would require the enabling of all targets. People that could be impacted are security researches, CTF enthusiasts and developers of applications targeted at architectures that are only part of the extended target set.

The behaviour of `strings` on similar files could be used to perform a DoS attack. A maliciously crafted binary `bin` which would get recognized as a `aixcoff-rs6000` target could take advantage of the lack of verification in the parser. One could encode in the `TLV`s multiple tens of thousands (or more) of sections, all pointing to the beginning of the file and having the maximum possible size. Additionally, the inclusion of printable characters should be ensured that form strings which are at least 4 characters long. Providing such a file to an unsuspecting victim that is likely to execute `strings -d bin` could overwhelm their system leading to a crash. While testing our PoC, this exact situation has affected also one of our laptops.

In conclusion, the bug is unlikely to be exploitable. Only in very limited cases it could be used to perform a DoS but there are many other more acessible, impactful and easier ways that could be done.

6. Appendix

6.1. Part 1 Git Diff

binutils_no_corpus.diff diff

```
diff --git a/projects/binutils/build.sh b/projects/binutils/build.sh
index 04fb516dc..7ba5df7d8 100755
--- a/projects/binutils/build.sh
+++ b/projects/binutils/build.sh
@@ -171,10 +171,6 @@ then
       -L/src/binutils-gdb/zlib ../libsframe/.libs/libsframe.a ../libiberty/libiberty.a -lz
 fi

-# Copy seeds out
-for fuzzname in readelf_pef readelf_elf32_csky readelf_elf64_mmix readelf_elf32_littlearm readelf_elf32_bigarm objdump objdump_safe nm objcopy bfd windres addr2line dwarf; do
-  cp $SRC/binary-samples/oss-fuzz-binutils/general_seeds.zip $OUT/fuzz_${fuzzname}_seed_corpus.zip
-done
 # Seed targeted the pef file format
 cp $SRC/binary-samples/oss-fuzz-binutils/fuzz_bfd_ext_seed_corpus.zip $OUT/fuzz_bfd_ext_seed_corpus.zip
```

6.2. Part 1 Visualizations (Enlarged)

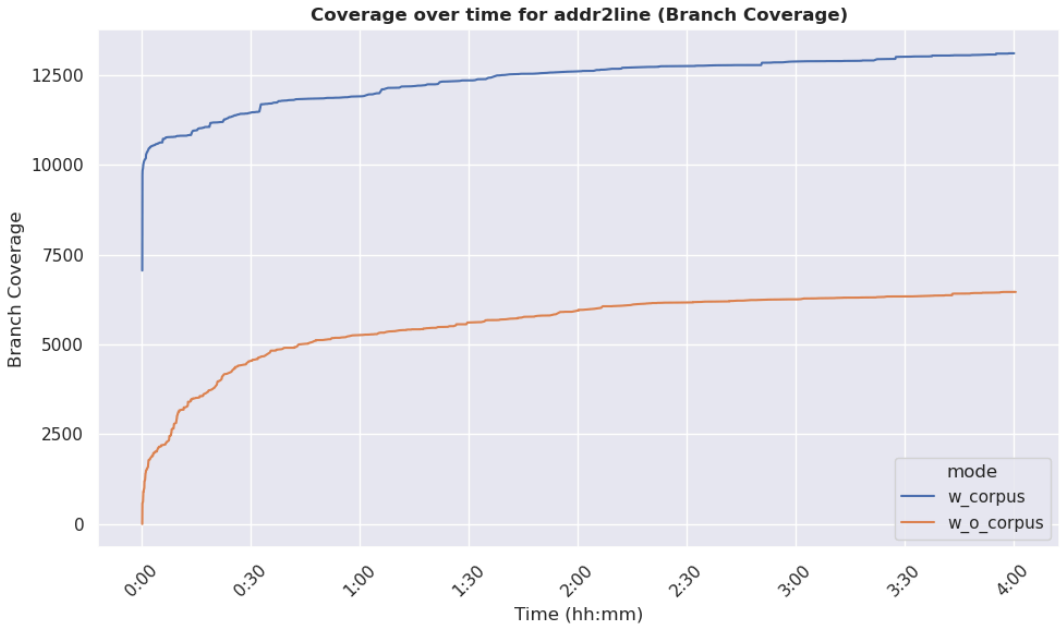

Figure 3: Coverage over time for `addr2line`

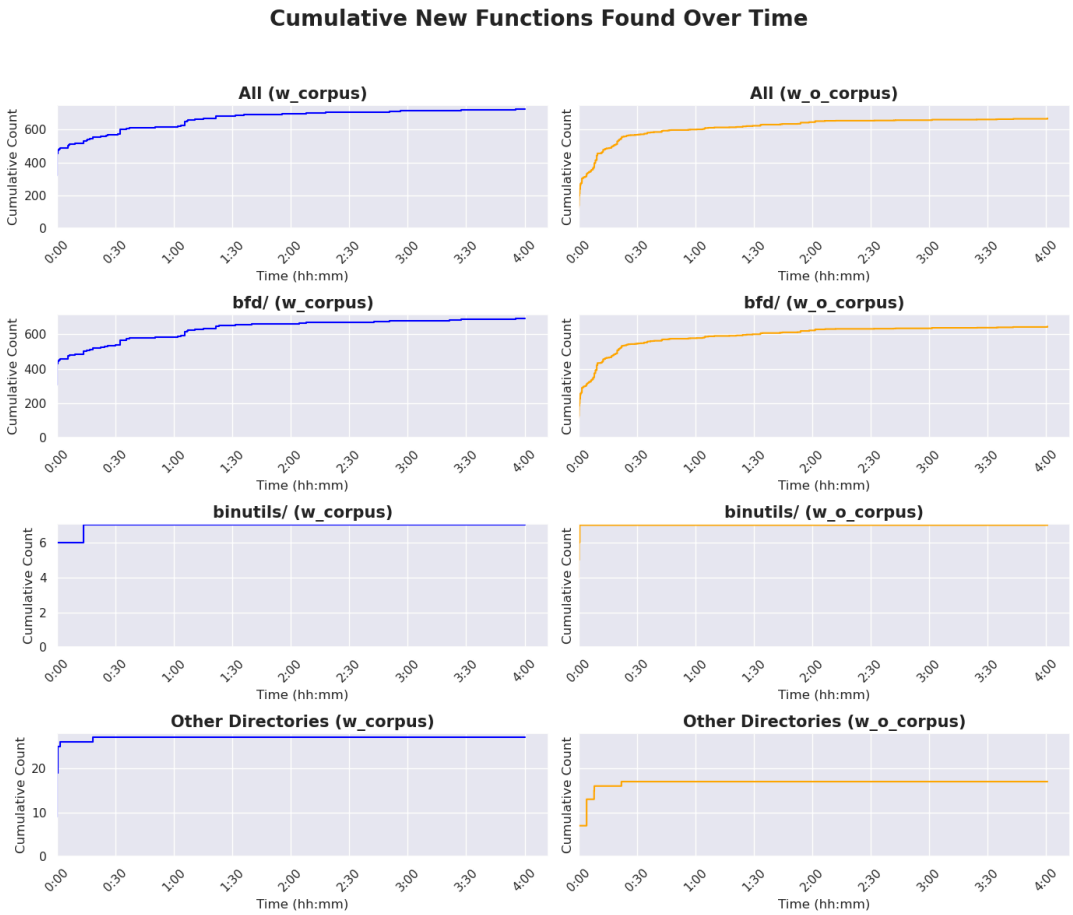

Figure 4: Found functions over time for `addr2line`

6.3. Coverage for `src/binutils-gdb/bfd`

Coverage	Without Corpus	With Corpus	Δ (%)	Δ (Lines)
Line	3.29% (11008/334485)	4.59% (15351/334485)	1.3	4343
Function	5.73% (506/8837)	8.15% (720/8837)	2.42	214
Region	3.36% (10625/315962)	4.42% (13950/315962)	1.06	3325

Table 2: Coverage for `bfd/`

The relative coverage differences in `bfd/` are small in percentage terms (1–3%), though they involve a large number of additional lines due to the size of the codebase. In comparison, `binutils/` shows similar percentage improvements on a much smaller code base. Since our analysis focuses on the `addr2line` harness behavior, we do not examine the broader coverage effects in `bfd/` further.