ITP 30002-02 Operating System, 2023 Spring

Homework 2

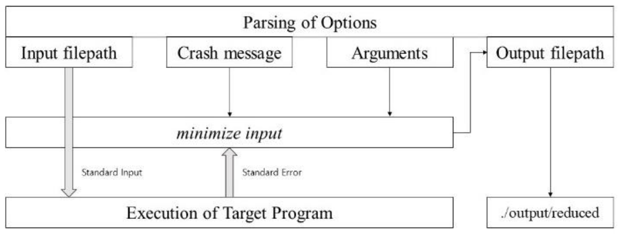
Seongbin Kim 22100113, Gunmin Yoo 21900458 {22100113@handong.ac.kr, 21900458@handong.ac.kr}

1. Introduction

This report describes the approach and evaluation of a program called `cimin`, which stands for Crash Input Minimizer. The program aims to produce a minimized input that preserves the crash status while communicating through system functions to provide functionality. The approach is presented in detail, including the requirements and the program's structure, which includes processing input options, minimizing the crash input, running the program, process communication, and ending the program. The evaluation of the program is also discussed, including a demonstration of `cimin` using a few test cases, dealing with *balance* using a global data structure, and finding the minimum crash input for *jsmn* and *libxml2*. We provide a comprehensive overview and evaluation of `cimin`, making it easier to understand the program's functionality and purpose.

2. Approach

In this section, we first introduce the requirements (Section 2.1). Then, we introduce the detailed approach of `cimin` (Crash Input Minimizer) (Section 2.2). Figure 1. represents the overall flow of `cimin`.



<Figure 1. Components of `cimin`>

2.1 Requirements

The requirements for `cimin`, our Crash Input Minimizer program, are based on two criteria, system functions and production of the minimized input. The goal is to produce an input that preserves the crash status while communicating through system functions to provide functionality (e.g., `fork()`, `pipe()`).

2.2 Structure of CIMIN

From the given requirements above, `cimin` is implemented by the following flow, processing input options, minimizing the crash input, running the program, communicating between processes, and ending the program.

2.2.1 Process of input options

The program receives input options via command-line arguments. Using the C library[1], the function `getopt()` parses the input options. The additional arguments required when executing the target program are listed after the binary execution file (e.g., ./a.out) of the target program. The program reads the list of additional arguments required for the target program when executing it and assigns them to a separate array. This is to send the list to the target program as `*argv[]` through `execv()`.

2.2.2 Minimization of the crash input

Our algorithm for minimizing the crash input, *minimize input,* is based on the *delta debugging*[2] algorithm. Based on the output of the execution of the target program with the modified input, the algorithm explores the subsequences. To compare the crashing messages, `strstr()` is used.

2.2.3 Execution of the target program

`cimin` executes the program to obtain the crashing message and compares it to the desired message (*-m* of our input option). Additional arguments are given through standard input via a pipe. These arguments are defined by the *-i* input option. Through the child process, created via `fork()`, `execv()` is executed.

2.2.4 Inter-communication between processes

As `execv()` replaces the image of the current process with the target program's process image[3], `fork()` is required to safely execute the target program. The main components used for the communication to work properly were `pipe()`, `signal`, and `dup2()`.

Two pipes were used for redirecting I/O, and `dup2()` to send and read via standard input or error. For the pipe from the parent to the child, the input is sent via standard input.

[1] unistd.h

[2] Andreas Zeller and Ralf Hildebrandt, Simplifying and Isolating Failure-Inducing Input, IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.

[3] https://linux.die.net/man/3/execv

In the case of the child to the parent, standard error outputs are redirected for the parent to read as a crash message. Exceptional cases such as an early end to the program or an infinite loop were dealt with signal communication. A signal alarm is reset for 3 seconds every execution, and every output is saved right away for preemption toward sudden termination signals.

2.2.5 Termination of the program

For the termination of `cimin`, the requirements are clear and concise. Our approach prints the specification of the result of the program, whether it was ended predetermined (via `Ctrl+C`), or not. The size of the final minimized input, and an output file with the input. Last but not least, the process identification number is saved for termination. This is to prevent a continuous program from executing whilst the parent is ended.

3. Evaluation

In this section, we provide a demonstration of `cimin` using a few test cases. Then, our evaluation of our program based on our implementation and results with follow.

3.1 Experimental setup

The criteria for correction using test cases are defined as the following: using the output file from `cimin`, which contains the derived reduced input, if the test case program produces the desired crash given in the message option, with the output file sent through standard input it is considered valid.

3.2 Dealing with *balance* using a global data structure

The *balance* test case provides a special condition to solve. The failing input requires the program to timeout the process when it passes the time limit (e.g., 3 seconds). When the program is signaled to end, it should provide a saved output of the crash-causing input. Preserving and updating the current data (reduced input) every iteration of the minimization process to a global data provided this functionality. Figure 2. describes the global data structure:

```
struct handler_args {
        int kill_pid;
        int length;
        char* output_string;
        char* output_filename;
} global_handler;
```

<Figure 2. Global data structure>

Through this global data structure, it is possible to solve both passing and failing test cases of *balance*. Saving the data before and throughout the execution of `cimin`, when the program is suddenly ended or given a non-crashing input, `cimin` provides the initial input.

3.3 Finding the minimum crash input for *jsmn*[4]

Our approach for *jsmn* provided the clearest view of the process of reducing the crash input. Figure 3. illustrates a snippet of the input being minimized to the minimum.

```
1 {\n\t″glossary″: {\n\t\t″title″: …}\n
2 {\n\t″glossary″: {\n\t\t″title″: …}\n
3 {\n\t″glossary″: {\n\t\t″title″: …}\n
4 {\n\t″glossary″: {\n\t\t″title″: …}\n
5 {\n\t″glossary″: {\n\t\t″title″: …}\n
```

<Figure 3. Crash input reduction of *jsmn*>

Starting with the initial *crash.json* file, the *minimize input* algorithm reduces the message string character-by-character.

3.4 Reducing the crash input for *libxml2*[5]

libxml2 had additional arguments to process. We had to deal with tighter constraints such as having to add a `NULL` value at the end of the 2D array of arguments. The execution of the program results in a reduced input. Figure 4. shows the result of producing the desired crash in the original test case as follows:

```
$ cd target_programs/libxml2
$ ./xmllint --recover --postvalid - < testcases/crash.xml
…
==ERROR: AddressSanitizer: SEGV on unknown address…
…
```

<Figure 4. Crash result of provided test case>

Figure 5. is an actual implementation of reproducing the crash using the output file *reduced* and is as follows:

```
$ cd target_programs/libxml2
$ ./xmllint --recover --postvalid - < ../../output/reduced
…
==ERROR: AddressSanitizer: SEGV on unknown address…
…
```

<Figure 5. Crash result of reduced crash input>

The results show after numerous iterations, the initial count of 143 characters is reduced to approximately 120. Using the test demonstrated in Figure 5. the resulting crash input is proven valid (Section 3.1).

Applying these various techniques was crucial for our journey to develop a program that minimizes a crash input.

4. Discussion

In this section, we further explore the possibilities of `cimin`. Related to our question, we provide ideas and

[4] https://zserge.com/jsmn/

[5] https://gitlab.gnome.org/GNOME/libxml2/-/wikis/home

thoughts for improvement to the process.

According to the results of the program, `cimin` successfully minimized the crash input of 3 out of 4 test cases provided. In testing using the test case *libpng*[6], we left it unsolved. Although executing the program with *libpng* options given produces a reduced input, we tested it to be invalid using the proof described in Section 3.1. To deal with `.png` files, we had to change our approach from reading files using `getline()` and `r` as the `fopen()` option to `fread()` and `rb` as the option.

The conclusion we reached with *libpng* was after an interesting test. Figure 6. shows executing the *libpng* program just from a higher directory:

```
$ cd target_programs
$ ./libpng/libpng/test_pngfix < ./libpng/crash.
$ ./libpng/libpng/testdata!: No such file or directory
```

<Figure 6. Observation of execution of *libpng*>

We decided there were more complex bugs hidden in *libpng* concerning directory paths.

Another limitation of our program, when executing using the *libxml2* test case, *minimize input* gets stuck on an infinite loop. It constantly produces results but in a repetitive fashion. To mitigate this threat, we confirmed the preemptive result (`Ctrl+C`) proves to successfully induce the exact crash condition.

Our initial approach for the *minimize input* algorithm was to slice character by character. While our approach was effective for the case of *balance*, moving on to *jsmn* raised an observation for us. Cutting a curly bracket of a `.json` file caused a `fread(): unexpected EOF` error. Affected by this error, we tried out cutting the input based on the given case's syntax. However, soon did we realize this was not effective for the *minimize input* algorithm. The breakthrough that needed to happen was to ignore the error other than the given crash message.

One of the main ideas we had throughout, was whether it would be possible to verify a reduced input is the minimal input. The partial answer we arrived at was that our current algorithm of *minimize input* is done in a greedy manner, it will not always be possible to converge to the minimal crash input. However, further work on this topic is needed as the exhaustive search of the paths may quickly grow infeasible.

Related to our question, we provide ideas and thoughts for improvement to the process.

One of the ideas suggested was to split inputs based on the specific syntax. For example, wrappers such as parentheses could be used to manage the reductions.

Zeller and Hildebrandt states that isolating a failure is much more efficient than simplifying the test case[7]. A particular instance aims to try obtaining the *1-minimal* difference. This can be done by applying the set of the difference between failed and passed cases ($\Delta = c^{\times} - c^{\checkmark}$). Unlike the original algorithm which derives its subsets simply from the failed cases, the difference is further used to apply union with the passing case ($c^{\checkmark} \cup \Delta_i$). If the union of the two results in a contradicting behavior of failing the test case, this proves the difference is at least a subset of the entire failure-inducing case. Gathering the multiple scattered subsets will construct a minimum (global, unlike the original approach) case. Constructing the minimum will be simple because the initial crash input is considered given.

5. Conclusion

We have proposed a Crash Input Minimizer, CIMIN, which aims to reduce the crash input, while preserving the same error message, till its minimum. Our study provides an automation of simplifying the crash input. The simplified input will have a larger portion of characters that induce the failure. Therefore, isolating the failure.

Our `cimin` program accomplishes (1) a user-friendly interface to execute the program, (2) inter-process communication via pipes and signals, and (3) successfully minimizes crash inputs for various test cases.

[6] http://www.libpng.org/pub/png/

[7] Zeller, Andreas, and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." IEEE Transactions on Software Engineering 28.2 (2002): 183-200.