**1. Introduction**

ITP 30002-02 Operating System, 2023 Spring

**Homework 2**

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

Our minimizing delta debugging algorithm *ddmin* is fed with a failing test case, which it simplifies by successive testing. It stops when a minimal test case is reached, where removing any single input entity would cause the failure to disappear. (original paper) This is surprising, because re-testing a program under changed circumstances is a common debugging approach and the only way to prove that the circumstances actually cause the failure. Eventually, we expect that several debugging tasks can in fact be stated as search and minimization problems, based on automated testing and thus be solved automatically.

**2. Approach**

In this section, we first introduce the necessary requirements (Section 2.1). Then, we introduce the detailed approach of cimin (Crash Input Minimizer) (Section 2.2).

**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, creating a minimized crash input file, 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]](#footnote-1), the function getopt() parses the input options. The additional arguments required when executing the target program are listed after the binary execution file 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 of minimizing the crash input is based on the *delta debugging[[2]](#footnote-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 target program and inputs the crash input to obtain the crashing message produced by the target program. cimin also receives this message and compare to the desired message (*-m* of our 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]](#footnote-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. 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 delt 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 specification of the result of the program, whether it was ended predetermined (via Ctrl+C), or not. The size of the minimized input, and an output with the minimized input are printed when the program is terminated while running. File with minimized crash input is created when program exits. Last but not least, the process identification number is saved for termination. This is to prevent a continuous program 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 Demonstration of CIMIN with test cases**

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 1. describes the global data structure:

**struct** handler\_args {

**int** kill\_pid;

**int** length;

**char\*** output\_string;

**char\*** output\_filename;

} global\_handler;

<Figure 1. Global data structure>

Through this global data structure, it is possible to solve both passing and failing test cases of *balance*. Saving the data prior 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]](#footnote-4)***

Our approach for *jsmn* provided the clearest view of the process of reducing the crash input. Figure 2. illustrates a snippet of the input being minimized to the minimal.

**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”glossar**y”**: {\n\t\t”title”: …**}**\n **5 {**\n\t”glossar**y**”: {\n\t\t”title”: …**}**\n

<Figure 2. Crash input reduction of *jsmn*>

Starting off 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]](#footnote-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 with a reduced input. Figure 3. shows the result of producing the desired crash in the original test case is as follows:

$ cd target\_programs/libxml2

$ ./xmllint -–recover -–postvalid - < testcases/crash.xml

…

==ERROR: AddressSanitizer: SEGV on unknown address…

…

<Figure 3. Crash result of provided test case>

Figure 2. 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 4. Crash result of reduced crash input>

The results show after numerous iterations, the initial count of 143 characters are reduced to approximately 120. Using the test demonstrated in Figure 4. the resulting crash input is proven valid.

Applying these various techniques were 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 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]](#footnote-6)*, we left it unsolved. Although executing the program with *libpng* options given produces a reduced input, we tested to be invalid using the proof described in Section 3.1. To deal with .png files, we had to change our approach of reading files from 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 5. 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 5. Observation of execution of *libpng*>

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

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 onto *jsmn* raised an observation to 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 would it we 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 exhaustive search of the paths may quickly grow infeasible.

One of the problems we found with this program is that it may not work properly if we run it in our personal ubuntu environment instead of running it on a peace server. When we ran the cimin program with *jsmn* in our personal Ubuntu environment, there was an error that the program did not terminate itself and could not escape the loop. When we terminated the program, the output was different from the result of running it on the peace server. We expect that this is probably the result of the difference in memory capacity of execution environment.

**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, (3) and successfully minimizes crash inputs for various test cases.

1. unistd.h [↑](#footnote-ref-1)
2. Andreas Zeller and Ralf Hildebrandt, Simplifying and Isolating Failure-Inducing Input, IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200. [↑](#footnote-ref-2)
3. https://linux.die.net/man/3/execv [↑](#footnote-ref-3)
4. https://zserge.com/jsmn/ [↑](#footnote-ref-4)
5. https://gitlab.gnome.org/GNOME/libxml2/-/wikis/home [↑](#footnote-ref-5)
6. http://www.libpng.org/pub/png/ [↑](#footnote-ref-6)