ITP 30002-02 Operating System, 2023-1

# Homework 2

Yang Sehyuk 21600415 21600415@handong.ac.kr Jang Jooyoung 21800637 21800637@handong.ac.kr

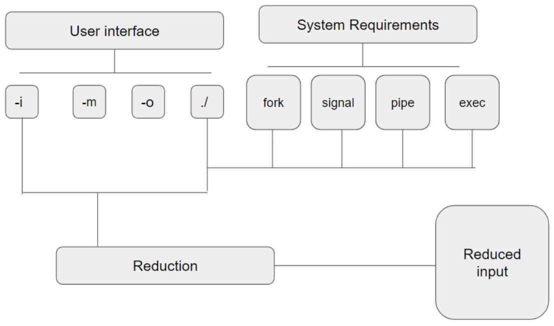
## 1. Introduction

According to research that was conducted by the Cambridge Judge Business School MBA project [1]. The annual cost for debugging is more than $61B. We cannot describe the importance of debugging during the development process. To solve this problem numerous algorithms and programs were invented. Delta Debugging is one of the efficient yet quite simple algorithms that was developed in 1999 by Andreas Zeller. Our goal for this homework is to implement Delta Debugging and test it on some examples that are out there on the internet. Before we implemented the basics of Delta Debugging, we had to deal with some system requirements. Firstly, implementation of the option handling method took a big part of the task. We had to think of ways to deal with at least four different options. Which are -i, -m, -o, and the executable binary program. We will explain what each option does and how we dealt with it in the approach part of the paper. The second requirement was to use the basic functions that make context switching possible which are fork(), exec(), pipe(), and signal(). Understanding these concepts was quite challenging but it was worth it. When every system requirement was set it was time to jump into the Delta Debugging algorithm.

The main goal of this program is to find out if the crash was caused by a crashing input(bug). After we reduce the input down to the root cause the output file will return that bug as a result. Above is the basic set of delta debugging. Now you can implement delta debugging. There are 3 cases of buggy programs and crashing inputs. The first one is ‘jsmn’ which is a JSON library related to heap-buffer overflow errors. When jsondump and jsmn/testcases/crash.json crash, you can check the crashing message “AddressSanitizer: heap-buffer-overflow”. The second one is ‘libxml2’ which is related to markup languages. xmllint is a utility program that detects syntax errors in given XML data. This case also generates an error message “AddressSanitizer: SEGV on unknown address …” Lastly, there is the case of ‘balance’. What if it falls into an infinite loop, cimin must kill a test.

Throughout this paper, we will discuss how we have tackled each requirement in detail. Starting from option to all the way down to the actual Delta Debugging. We had some challenges on our way and we’ll also make a discussion on that too.

## 2. Approach



**figure(1) Basic idea of our approach towards delta debugging**

### 2.1 User interface

One critical thing We need to tackle before implementing the delta debugging algorithm was to implement the interface of cimin. We were required to receive at least four options. The fundamental knowledge of implementing this was to use getopt() which was provided by <unistd.h>. It deals with delimiters that we required which were -i, -m, -o. Each of them was assigned file_path for crashing input, an error string that will be compared with the stderr result later, and output file_path for reduced crashing input. The last argument that we needed to receive was the executable binary program (target program) and the options for that specific program. We have used optind which is the index for options that were provided by getopt() and the char* argv[] array to receive the target program’s option. We made sure that the target program’s file path was to be secured into a different variable since it is going to be used throughout the program.

### 2.2 System Requirements

The crucial part of this assignment is to understand the basic knowledge of context switching, process creation, and IPC. Thus we had to know the usage of fork(), exec(), wait(),pipe(), signal(). The tasks between the parent and child process were clearly divided. We had to decide which one takes the writer role and which one takes the reader role. Based on our knowledge we agreed upon the fact that fork(), and exec() will be used to create the child process and to execute the target program on the child process. Following this decision to decide on the number of pipes to generate to tackle the requirements. According to the requirements we had to make two sets of pipes. One for the STDIN pipe which will be used to deliver the input for the target program. The other one was for the STDERR pipe which will be used to deliver the error message of the target program when it crashes. In order to execute target program that has additional options for itself. We have divided it into two situations where there are additional arguments or not. We detected this by checking the additional_args[1] is NULL or not, because the target programs file path will be stored in additional_args[0].

The last part was to deal with signal interrupts. We had to configure ways to cope with SIGINT and SIGALRM. To do this we made int_handler and alrm_handler functions. This function reacts to each signal based on the signal value that each signal generates. SIGINT was generated by CTRLC and when it happens, we made sure to kill the ongoing child process, return the size of the reduced input and save the postponed result of the reduced output and write it to the designated file_path. Then safely exit. SIGALRM was a different type of signal that was caused by a timer. We have used a simple alarm function to do so. We have set the timer to three seconds at the parent process to make sure the execution time does not exceed its limit. When SIGALRM occurs it will kill the ongoing child process and will deliver a SIGINT signal to use the previous function that we provided, which is to store the current output and safely exit the program.

**2.3 Delta Debugging and additional requirements.**

Delta debugging is an efficient algorithm that isolates and identifies the root cause of an error in a program. Our goal was to successfully implement this algorithm and test it on three different types of error-causing programs to find out the root cause of it. The structure of the algorithm was simple to follow. Once the input is received the algorithm will divide the input into two parts and then greedily do the searching. Although it was easy to understand the basic concept of an algorithm it was challenging to implement it. 2.1 and 2.2 were all the cornerstones that will be used to deliver this algorithm successfully. After we received users' input for the initial crashing file_path. We read what was in the file and stored it in a buffer. Then divide this into three parts which are head, tail, and mid. We used the strndup() function to deliver the designated characters to each part. The most challenging part was to think of the idea to run the program inside the algorithm. This was because at first, we thought the order was to fork - exec - reduce algorithm. However, we found out that the step of fork and exec needs to go inside the reducing algorithm. Hence, we forked a new child process and loaded the targeted program on it. When we tested with the ./balance it successfully limited its inputs down to ][]*[][ \n \n and got into time out. The key concept to implement delta debugging was to deliver inputs and outputs of STDIN , STRERR , etc. We made it possible to achieve this goal by using pipes that we generated previously. Knowing when to close and open the pipe was important.

**3. Evaluation**

By executing ./balance we could prove we have successfully implemented several requirements. Firstly, the process creation. As stated in the Approach section we have used fork() to create a new process. Then we loaded the target program "./balance" with function exec(). We used unnamed pipes to receive and send crash messages and inputs for the program. When we printed out the buffer it was successfully delivering the assigned contents. While doing this we made sure not to exceed the buffer size of 4096. The reducing algorithm worked well in this program, and we checked it by printing the result on the console and reviewing the intermediate outcome of the reduced algorithm. We also did simple logic checking by hand by following the code. Then compared the expected result with the actual result to see if it was successful. Lastly, the fundamental reason why the balance was provided as the testing example was to observe the timer interrupt. When the crashing input was given the program has done some reduction then stopped for 3 seconds and returned the saving output with the ][]*[][ \n \n . Hence, we can say that the program implemented the timer interrupt. Since the test cases for balance had a low number of characters to reduce, we tried cimin on jsondump. It successfully fetched its initial input and then did the reduction all the way down to {y}. The last testing that we need to deal with was libxml2's xmllint and its crashing input. This task wanted us to demonstrate whether we had implemented a method to receive addtional arguments for target program's options. As we stated above we made another type of buffer just to store addtional arguments which is named addtional_args[]. To deliver these arguments we used execv which receives the target programs file path and addtional argument as the parameter. It successfully executed and reduced its crahing input down to <!DOCTYPE[<!ELEMENT\n:(□,()><:. It took some time to reducing the input, however, still obtain it. We double checked if we got the correct result by giving the outputs as the crashing input and it gave me the error message. Thus, we can say that our program was successful.

**4. Discussion**

**4.1 Limits on Delta Debugging and Solutions**

Delta Debugging is a very useful software debugging technique, but it seems not perfect. First, as it focuses on finding the least set of inputs, it might not be possible to find the fundamental cause of the bug. So sometimes, it is more difficult to find and understand the cause of the bug. To overcome this problem, you need to keep analyzing test cases to find hidden reasons for bugs. Or you can do code reviews or use static analysis tools.

Secondly, what if the test case is too heavy, prodigious time can be spent. So, it is hard to use in a massive system and distributed system. To overcome it, you must optimize the input distribution algorithm of delta debugging. For example, you can find an area where's importance is higher than others or diminish the complexity of time of the distribution algorithm.

Third, as Delta Debugging is still an automated debugging skill, you can't find or change every bug through Delta Debugging. In case, human interference is required. The problems in which the minimum set of inputs exists only for certain inputs usually arise when those inputs have very different structures from other inputs. In this case, it can be effective to analyze the structure of the input values before delta debugging and find input values with similar structures.

### 4.2 Delta Debugging's advantages

But there also are reasons why Delta Debugging is efficient as well. Delta Debugging has many advantages over other debugging techniques.

First, Delta Debugging is an automated minimum set of input extraction techniques that are much more efficient than manually analyzing and minimizing test cases. It is very useful for finding complex bugs in large systems. Second, Delta Debugging can quickly resolve a bug by finding a set of minimum input values that can reproduce the bug. This is particularly effective for relatively simple bugs. Third, Delta Debugging can be used in conjunction with other debugging techniques in minimizing input values. For example, when used with code coverage optimization, static analysis tools, etc., more effective debugging is possible. Fourth, Delta Debugging is open-source-based software that allows you to freely use the library or source code and is customizable.

Delta Debugging is therefore faster and more efficient than other debugging techniques and has several advantages, including the ability to resolve bugs quickly.

### 4.3 Related works

There are a few related works about delta debugging. One interesting study is Andreas Zeller et al [2] "Simplifying and Isolating Failure-Inducing Input" which is an improved version of Delta Debugging, which presents a method of simplifying and isolating inputs that determine the cause of failure. In this paper, we refine the Delta Debugging algorithm and present a way to extract useful information in the process of simplifying input. It also describes how to remove input elements unrelated to the cause of failure, and how to handle different types of input data. Experiments present that the improved Delta Debugging algorithm is more effective than the previous version and performs better than other debugging techniques. These results show that Delta Debugging can be useful in real-world software development.

## 5. Conclusion

### 5.1 Summary

Delta debugging is a very useful technology for detecting software bugs quickly. This technique is used to reduce and simplify input values to find bugs. Delta debugging first divides the input value into minimal units and executes this divided input value to find out where the bug is. Afterward, the divided input values are combined again and repeated until the bug is found. This allows you to quickly find bugs and saves you the time and money required to perform the tests.

One of the advantages of Delta debugging is that it can automate debugging. Debugging is an important step in the process of program development, but performing debugging manually is very inefficient and time-consuming. Delta debugging allows developers to solve bugs more efficiently while saving time and money.

### 5.2 Limitations of Delta Debugging,

However, Delta debugging also has its limits. If there are too many divided input values, the execution time may be longer, and the divided input value may be excessively simplified, resulting in missing important parts. Delta debugging is also applicable only for a given input and may not work properly for other input values.

Therefore, Delta debugging is a very useful technology in the debugging process, but it should be recognized that there are also precautions when using it. In particular, if input values are complex, prior work such as refining data before applying Delta debugging may be required.

Delta debugging, however, does not detect all bugs. Delta debugging may fail, depending on the complexity of the bug or the development environment. In addition, Delta debugging can cost additional money and time.

### 5.3 Meaning of Delta Debugging and Human's role,

However, Delta debugging is still a very useful and important technology. Research and development are being conducted to find better ways to debug bugs, and various techniques and methods are being studied to overcome Delta debugging's limitations.

Therefore, Delta debugging is one of the most important technologies you need to know in software engineering, such as software developers, debuggers, and testers.

In addition, while using automation tools such as Delta debugging is very effective, it does not completely rule out human roles in the overall software development process. Human knowledge and experience are still important, and the best results can be achieved when automated tools and human efforts work complementarily.

Therefore, we need to value human effort and experience while making the most of the benefits of automation tools such as delta debugging and pursue continuous improvement and innovation in the overall software development process.

### Reference

[1] "Study: Software Failures Cost the Enterprise Software Market $61B Annually." *PR Newswire: Press Release Distribution, Targeting, Monitoring and Marketing*, 28 May 2020, https://www.prnewswire.com/news-releases/study-software-failures-cost-the-enterprise-software-market-61b-annually-301066579.html.

[2] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," in *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183-200, Feb. 2002, doi: 10.1109/32.988498.