ITP 30002-{01, 02} Operating Systems, Spring 2023

Homework 2. CIMIN: Crashing Input Minimizer

Shin Hong
hongshin@handong.edu

1. Introduction

Fixing bugs or *debugging* is the most time-consuming activities in software development lifecycle since it is difficult to mechanize reasoning for identifying the locations of bugs in a program when crashes occur. To alleviate this challenge, many automated approaches that assist developers to diagnose buggy programs and test inputs, including test input generation, crash reproduction, test minimization and fault localization.

Delta debugging[1] is a test input minimization technique that takes a target program together with a crashing test input, and then automatically derives a part of the test input that still make the program crash. Reducing test input is helpful for developers because a smaller input covers less code regions, thus the scope to review the source code to find the root cause of the problem (i.e., buggy code elements) is reduced.

For Homework 2, you are asked to write down `cimin` to implement a delta debugging algorithm (Section 2) as a C program. This homework poses design requirements (Section 3) that specify system functions (e.g., `fork()`, `pipe()`) that you must use for implementing certain functionalities of `cimin`.

Each team must submit the resulting program, and a report by 9 PM, Apr 8th (Sat). Late submissions will be accepted within the next three days (by 9 PM, Apr 11th) at the penalty of 25%. Homework 2 is assigned as a teamwork; thus, team members must collaborate in all tasks of the homework.

2. Delta Debugging

Testing is to run a target program with a given test input and check if the program does not crash and produces the correct output for the given test input. In general cases, an input of an application program is defined as a byte stream given to standard input. And we can check if the output is valid (i.e., non-crashing) and correct by observing the byte stream from standard output.

Figure 1 describes a variant of the delta debugging algorithm that gradually reduces a given test input while preserving crashing result. This algorithm receives three inputs: (1) p, an executable binary of a target program, (2) t, a test input that crashes the target program, and (3) c, a condition to determine the crash. This algorithm repeatedly executes p with a different subsequence of t to find out a small subsequence (i.e., a reduced input) that crashes p and shows m in the crash message (i.e., preserve the faulty result as the same with t).

Since it is infeasible to explore all input subsequences, the algorithm explores only a part of them in a greedy-search manner. Given minimal crashing input so far (t_m at Line 1), the algorithm iteratively derives input subsequences by removing substrings of size s (Lines 5-6) [2] and runs p with these (Line 7). Initially, t_m is defined as the given crashing input t (Line 22), and s is defined as $|t_m| - 1$. If a derived input crashes p and satisfies c, it updates t_m as the derived input and continues the reduction (Lines 8-10). If no derived input exhibits the crash, the algorithm moves on to explore another set of input

Input: p, a target program
t, a test input that crashes p
c, a predicate over output to check if the program crashed

Output: t_m, a reduced crashing input

```
    REDUCE(t_m):
1.    t_m ← t
2.    s ← |t_m| − 1
3.    while s > 0 begin
4.      foreach i ∈ [0, |t_m| − s] begin
5.        head ← t_m[0..i − 1]
6.        tail ← t_m[i + s..|t_m| − 1]
7.        o ← p(head + tail)
8.        if o satisfies c then
9.          return REDUCE(head + tail)
10.       endif
11.     end
12.     foreach i ∈ [0, |t_m| − s − 1] begin
13.       mid ← t_m[i..i + s − 1]
14.       o ← p(mid)
15.       if o satisfies c then
16.         return REDUCE(mid)
17.       endif
18.     end
19.     s = s − 1
20.   end
21.   return t_m

    MINIMIZE(t):
22.   return REDUCE(t)
```

<Figure 1. Delta Debugging Algorithm >

subsequences: it checks if the substrings of size s (Line 13) crashes p while preserving the crash behavior. If found, the algorithm updates t_m and continues the reduction with it (Lines 14-17). If the algorithm could not find a reduced crashing input with s, it decrements s by one (Line 19), and repeats the input subsequence exploration until s becomes zero (Line 3).

Note that this algorithm does not guarantee to produce an optimal solution (i.e., a minimal crashing input) because it does not check all possible subsequences of a given test input. And, since the same algorithm may be implemented in different ways, the reduction results may be different for the same given input.

3. Requirements

3.1. User Interface

You are asked to implement the algorithm of Figure 1 as `cimin`. `cimin` receives command-line arguments as following example:

```
$ls
cimin    a.out    crash
$./cimin -i crash -m "SEGV" -o reduced ./a.out
$ls
cimin    a.out    crash    reduced
```

Initially, three parameters must be given with options. The `-i` option is followed by a file path of the crashing input (i.e., the subject of the reduction). The `-m` option is followed by a string whose appearance in standard error determines whether the expected crash occurs or not. The `-o` option is followed by a new file path to store the reduced crashing input. `cimin` must return

* Last update: 2 Apr, 2023

[1] The original delta debugging algorithm is found in the following paper: Andreas Zeller and Ralf Hildebrandt, Simplifying and Isolating Failure-Inducing Input, IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.

[2] $t[b..e]$ is the substring of t from index b to index e inclusively if b is less than equal to e; otherwise, an empty string.

proper error messages if a given argument is invalid.

After the option arguments, the command to run the target program follows. In the given example, the command is the file path to the executable binary of the target program (i.e., `./a.out`). If the target program needs to receive its own command-line arguments, multiple arguments must be given.

When the algorithm terminates, the program prints out the size of the final reduced crashing input, and produces output file. A user can send a signal by `Ctrl+C` to stop `cimin` anytime. for this case, `cimin` must stop the running test if exists, and prints out the size of the shortest crashing input found so far, produces the output with it, and terminates the execution.

3.2. System Design and Requirements

`cimin` repeatedly runs the target program to check if a reduced input shows the same crash. To run the target program, `cimin` must use `fork()` to create a new process, and `exec()` (or its variants) to load the target program in the created process.

We assume that the target program receives input via standard input and sends out crash message to standard error. To pass input to the target program, `cimin` must use unnamed pipe (i.e., `pipe()`) to redirect standard input. For simplicity, we assume that a crashing input does not exceed 4096 bytes. Thus, the full content of a crashing input can be loaded in memory.

`cimin` must use unnamed pipe as well to redirect the standard error to receive the error message from the target program execution. We assume that the target program produces crash message to standard error, and we can find a keyword to determine if the same crash happens with reduced inputs. This assumption is strong because, in general, erroneous behavior may not be observed as crash message, or the same bug may result different crash messages depending on input; nonetheless, we employ this strong assumption because it is acceptable in common cases, and it keeps the simplicity of your assignment.

`cimin` must reject the initial crashing input if its execution takes more than 3 seconds. In exploring input subsequences, if a test execution runs over this time limit (i.e., 3 seconds), `cimin` must kill the process and consider that the input does not reproduce the expected crash. You must use signal mechanisms to provide this functionality. Although infinite loop is an important kind of software error, we exclude the case of infinite loop in Homework 2.

3.3. Buggy Program Examples

You are given three cases of buggy programs and crashing inputs[3]. More cases may be given for your understanding.

jsmn: *jsmn* is a popular open-source JSON library. This program has a bug resulting heap-buffer overflow errors. You can build the program by running `jsmn/build.sh`. This build script uses LLVM AddressSanitizer to explicitly detect an occurrence of heap-buffer overflow. As the build result, `jsondump` will be generated. The crashing input is given at `jsmn/testcases/crash.json`. `jsondump` will crash if it receives `jsmn/testcases/crash.json` via standard input. You can characterize this crashing message by checking if it has "`AddressSanitizer: heap-buffer-overflow`". Note that the stack trace or memory address values in crashing message may change depending on executions.

libxml2: *libxml2* is an open-source library for parsing and manipulating formatted data in multiple markup languages such as XML and HTML. By running `libxml2/build.sh`, all libraries and utility programs of libxml2 can be built. `xmllint` is a utility program that detects syntax errors of given XML data. The given version of `xmllint` has a bug that results null pointer dereference errors. You can trigger this bug by executing xmllint with option "`--recover --postvalid -`" and passing `libxml2/testcases/crash.xml` to standard input. The instruction for building and reproducing the crash is as follows:

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

The crashing symptom can be identified by checking if "`SEGV on unknown address`" is printed to standard error. This error message is produced by LLVM AddressSanitizer when a crash occurs by null pointer dereference.

balance: *balance* is a toy example program that checks if the given input has well-balanced parenthesis or not. You can build the program by running `balance/build.sh`. When this program receives `balance/testcases/fail` via standard input, it will fall into infinite loop. `cimin` is expected to kill a testing run if it exceeds the time limit.

3.4. Report Writing

Your report must follow the structure of the given template, and it must not exceed 3 pages. In the evaluation, the descriptions on the following points are expected to be found from your report:

- an overview of the program design,
- a description on how to use system functions to achieve expected functionalities,
- demonstration of your program with given examples, and
- discussion: your findings, the challenges, unsolved questions, lessons learned, or any other interesting discussions related to this homework as well.

You can achieve extra points if you write interesting discussions. For example, you may propose a sound and interesting idea to improve the delta debugging algorithm, or an idea to implement the algorithm in a more efficient way.

Note that the evaluation is primary based on your report, and your implementation will be tested to check whether it consistently works as described in the report.

3.5. Evaluation

Your result will be evaluated against the following criteria:

- the report effectively delivers how you designed and implemented the program to achieve the requirements.
- the report clearly demonstrates that your implementation is effective with buggy program examples.
- the discussion in the report is relevant and interesting.
- your program properly system libraries following the given instructions.
- your program successfully runs with test cases.

4. Submission Instruction

Upload a zip file to HDLMS, containing all results including all files of your implementation and your report. Your report must be in PDF for compatibility. Your program must contain a build script (e.g., `Makefile`) and documentation (e.g., `README.md`) on how to build and run the program. Your program must be built and run successfully on the peace server (`peace.handong.edu`) because your program will be tested on this environment.

The submission deadline is 9 PM, Apr 8. You can make late submission by 9 PM, Apr 11. A late submission will get 75% of the score after evaluation.

[3] `https://github.com/hongshin/OperatingSystem/tree/hw2`