Delta-debugging on Traces

by

Xueyuan Chen

A Creative Component Report submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
Master of Science

Major: Computer Science

Program of Study Committee:
Wei Le, Major Professor
Myra Cohen

Abstract

Debugging is challenging and time-consuming to find the cause of a failure by inspecting source code. In this creative component, we firstly apply delta-debugging to reduce C program traces for bug diagnosis. Taking a single failing run, we generate a trace. We use a tool from our lab, namely, Helium, to generate an executable program from the trace and reduce the program with the delta debugging tool C-Reduce. The results contain only the statements that are sufficient to reproduce the failure. We studied C-Reduce and used C-Reduce with two reduction settings. The size of the reduced program with reproduction setting 1 is smaller or equal to the size of the reduced program with reproduction setting 2 due to function merging. Compared with the traces, we totally reduced 74.38% of code with reduction setting 1 and 71.98% of the code with reduction setting 2.

1. Introduction

When the program fails, the developer must debug it to fix the fault. Reproducing the failure is essential for debugging. Without reproducing the failure, the developer will have trouble diagnosing the problem and finally proving that it has been fixed. Reproducing failure depends on the knowledge of the circumstances that led to the failure. If these are little known or difficult to recreate, reproducing can be a challenge. To improve the efficiency of debugging for programmers, program reduction techniques [7][8][9][10] are proposed to reduce failure-irrelevant event traces or statements.

Program slicing aims to identify the statements relevant to the bug. A program slice [15] is a subset of the program execution that is relevant to a specific statement or behavior. Slicing is based on dependencies between statements. Starting from a statement, the transitive closure over all dependencies forms a program slice. In debugging, applying the backward static slicing on a failing statement could generate all statements that could influence the failing statement. While static slicing generates statements to all possible runs, dynamic slicing just applies to the failing run with the fault-inducing input and thus is more precise. Dynamic trace of fault can be generated by running the program with the fault-inducing input. The trace contains the program dependence and removes the statements which are not depended by a failure. However, not all remaining statements in trace are necessary to reproduce the failure.

Delta debugging [1] is a general solution to automatically simplify failure-inducing inputs. Zeller and Hildebrandt [16] applied Delta Debugging algorithm on reducing test input. The algorithm can generate a minimal failure-inducing input but is not suitable for program reduction. Because in design, it neither utilizes nor respects the constraints of the syntax structure enforced by a programming language's formal syntax. To improve the effectiveness of Delta Debugging for

tree-structured inputs, Misherghi and Su [2] proposed Hierarchical Delta Debugging (HDD). HDD converts a test input into its tree representation by only using the formal syntax of a language but does not exploit the grammar further to guide reduction. HDD generates smaller results than Delta Debugging but it still generates several invalid results. Regehr et al. [9] proposed C-Reduce which was a highly customized reducer for C/C++. Although C-Reduce is powerful at the minimization of C/C++ programs, it takes a long time to reduce and may not guarantee the quality to generate the smallest reduction sizes.

In this creative component, we propose an approach that removes failure-irrelevant statements in the dynamic trace that leads to a failure by using delta debugging. At first, we generate the trace of a failing run so statements which are not exercised by the failure can be removed. The generated trace consists of the sequences of the statements which are not executable. Therefore, we use a tool which takes the trace as input and generates an executable test case. Finally, we run delta debugging tools to further remove failure-irrelevant statements. The reduced program can reproduce the failure.

Our goal is to use the final program to help programmers debug failure. Therefore, the final program should keep the dependencies between failure-relevant statements to make sure of the correct path of the fault. Without dependencies, the patch is hard to transfer between the original program and the reduced program. To help analyze the original program, the variable and functions should not be renamed; otherwise, the programmer could have difficulty to compare with the original program.

2. Approach

2.1. Overview

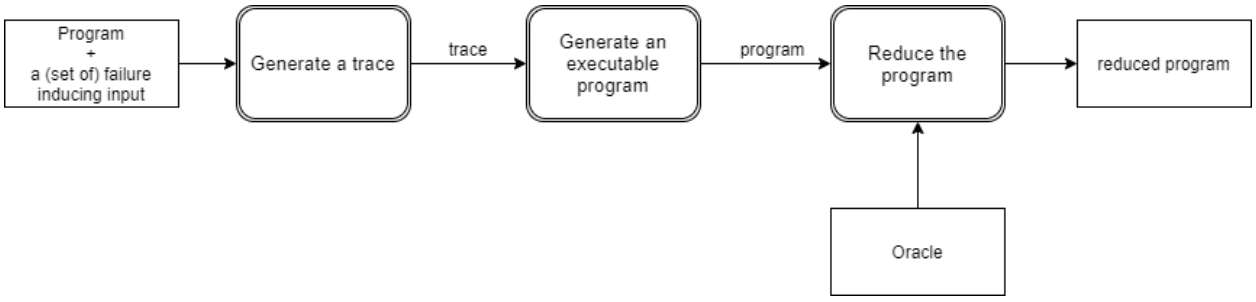

Figure 1. Workflow

We generate the dynamic traces of the faulty program with a (set of) failure inducing input. The failure inducing input can be found from the bug report. Otherwise, we find the input based on the test script of the program which triggers the fault. After generating the trace, we use a trace as input and generate the executable test case using a syntactic patching tool called *Helium.* Then

we apply Delta Debugging on the executable program. We use oracles associated with the executable test cases to ensure that the failure is reproduced during Delta-Debugging. For example, when we detect that the size of a string is greater than the buffer size, we report a buffer overflow. The reducer works automatically under a shell script, which returns 0 if a test is successful and non-zero otherwise. The correct reduced program should trigger the fault with expected output.

2.2. Generate the executable program from trace

We generate traces by using trace generation tool trace-pintool [13]. To generate an executable program from the trace, we use a tool from our lab, namely, Helium. The input file of Helium needs to be in a specific format. Therefore, we write a python script to reformat the output of trace-pintool. Taking the trace as input, Helium patches a code fragment and makes it syntactically valid. The paths in trace should be kept with the patch so that the fault in the can be reproduced. Then Helium resolves the dependencies and builds the code fragment to generate an executable program. For each C file related to trace, global variables, function declarations, included header files and type definitions are moved to a header file. Helium includes these header files in the code fragment. And by using these header files to figure out which files in the original project and which libraries should be linked to, Helium prepares a build script for the code fragment to make the program executable.

Using coreutils-7eff5901-6fc0ccf7 as an example, Helium takes input from the trace shown in Figure 2a together with the source code of coreutils-7eff5901-6fc0ccf7. Helium generated an executable program shown in Figure 2c. Helium adds the red lines (lines 12, 14, 15, 16, 23, 24, 26, 27, 29 and 33) to make the code syntactically valid. Some statements (line 339, 346, 349 and 350 in Figure 2b) of the original program are relevant to other statements in the trace but the trace-pintool does not include them in the trace. Without adding lines 15, 23, 26 and 27, the program cannot preserve the semantics of the original program. In addition, Helium adds the yellow lines to resolve the dependencies needed to build the code fragment, including variable dependencies, type dependencies and function dependencies such as function definition (line 7) and function declaration (line 5) as well as the header files (line 1-3). The patched program then can be compiled and executed.

```
((
("coreutils/src/expr.c" ... 336 337 341 343 344 345 351 347 352 357 359 ... )
("coreutils/lib/long-options.c" 54 55 56 73 58 61 63 64 83 86 90 91)
("coreutils/lib/xmalloc.c" 55 56 38 39 40 41 42 43 57 227 228 219 220 221 229)
))
```

(a) Trace as input of Helium

file: coreutils/src/expr.c

```
323  static bool
324  toarith (VALUE *v)
325  {
335  ...
336      if (sign < 0)
337        cp++;
338
339      do
340        {
341          if (ISDIGIT (*cp))
342            {
343            intmax_t new_v = 10 * value + sign * (*cp - '0');
344            if (0 < sign
345                ? (INTMAX_MAX / 10 < value || new_v < 0)
346                : (value < INTMAX_MIN / 10 || 0 < new_v))
347              error (EXPR_FAILURE, 0,
348                 (0 < sign
349                  ? _("integer is too large: %s")
350                  : _("integer is too small: %s")),
351                 quotearg_colon (v->u.s));
352            value = new_v;
353            }
354          else
355            return false;
356        }
357      while (*++cp);
358
359      free (v->u.s);
360  ...
```

```
1   #include "he_long-options.h"
2   #include "he_xmalloc.h"
3   #include "he_expr.h"
4   ...
5   static bool helium_toarith(VALUE *v);
6   ...
7   static bool helium_toarith(VALUE *v)
8   {
9     ...
10
11      if (sign < 0)
12      {
13        cp++;
14      }
15      do
16      {
17        if (((unsigned int)(*cp) - 48 <= 9))
18        {
19          {
20            intmax_t new_v = 10 * value + sign * (*cp - 48);
21
22            if (0 < sign ? ((9223372036854775807) / 10 < value || new_v < 0)
23                         : (value < (-9223372036854775807 - 1) / 10 || 0 < new_v))
24            {
25              error(EXPR_FAILURE, 0,
26                    (0 < sign ? ((const char *)("integer is too large: %s"))
27                              : ((const char *)("integer is too small: %s"))),
28                    quotearg_colon(v->u.s));
29            }
30            value = new_v;
31          }
32        }
33      } while (*++cp);
34      free(v->u.s);
35      ...
36  }
37  ...
```

(b) Source code of coreutils-7eff5901-6fc0ccf7 **(c) Program generated by Helium**

Figure 2. The example of coreutils-7eff5901-6fc0ccf7

2.3. Reducer

2.3.1. Selection of reducers

We started with three delta-debugging tools, C-Reduce [9], delta [12] and multidelta [12] whose source code is available. All of them can handle c programs. All of them needed an “interesting” test script for reducing a program. The program needs to initially pass the test. With experimenting on simple programs, C-Reduce reduced the program to the smallest but might delete fault relevant statements. And it renamed the variable and function with simple names (eg. a, b, …). Multidelta could reduce the program but left some statements which should be removed. Delta reduced to a program which was smaller than the program of multidelta but

larger than the programs produced by C-Reduce. To get a smaller size reduced program, C-Reduce and delta are better choices.

C-Reduce transforms the file in a schedule of *passes*. Passes are a set of C/C++ program transformations designed by C-Reduce, and can be removed or added. The quality of the reduced program from C-Reduce was the smallest but not ideal. However, C-Reduce has a flag *--remove-pass* which can remove passes from the scheduled passes. By removing different passes, it gets different reduction settings. C-Reduce with different reduction settings can produce different reduction results. In addition, C-Reduce has passes that do everything that Delta does. Therefore, C-Reduce is a better reducer.

2.3.2. C-Reduce study

Our expected reduced program keeps only the statements relevant to the failure and the dependencies between failure-relevant statements. However, the program reduction tool C-Reduce does not produce the expected result with the original reduction setting. Using the results of ncompress-4.2.4 as an example. ncompress-4.2.4 contains a buffer overflow bug at the statement *strcpy(tempname, *fileptr);*. Without removing any passes, C-Reduce will produce a reduced program as Figure 3a. The reduction fails keeping the program dependencies related to fault. Compared with the good version as Figure 3b (this is generated with the reduction setting 2), the version with original setting contains several issues such as merging functions, renaming variables name, reducing the dimension of pointer and removing the if-else dependencies. The fault statement *strcpy(tempname, *fileptr);* should be in the function *helium_comprexx*. However, the buggy version merges the function *main* and the function *helium_comprexx*. The variables are renamed with simple names in the buggy version. Variable *tempname* is renamed as *b*. It's hard to analyze this version of the reduced program and map the results to the original program. The reduced program with the original setting is not helpful for debugging. The reduction setting can be modified by using flag *--remove-pass* to remove passes from the original setting. And the quality of the reduced programs can be improved.

```c
main(int argc, char *argv[]) {
  char *a;
  a = (char **)malloc(argc * sizeof(a[0]));
  char b;
  strcpy(b, *a);
}
```

```c
void helium_comprexx(char **fileptr) {
  {
    char tempname[1024];
    strcpy(tempname, *fileptr);
  }
}
void main(int argc, char *argv[]) {
  {
    char **filelist, **fileptr;
    filelist = fileptr = (char **)malloc(argc * sizeof(fileptr[0]));
    for (argc--, argv++; argc > 0; argc--, argv++) {
      if (**argv == '-')
        ;
      else {
        *fileptr++ = *argv;
      }
    }
    if (*filelist != (void *)0) {
      for (fileptr = filelist; *fileptr; fileptr++) {
        helium_comprexx(fileptr);
      }
    }
  }
}
```

(a) The version without original setting **(b) Good version**

Figure 3. Reduced programs of ncompress-4.2.4

To determine which passes to keep or remove, we need to see what each pass is doing. The source code of C-Reduce has documented the usage of a few passes. To learn more about those passes and figure out the rest, we ran a small program with all passes on, and logged all the information about the passes. Then we categorized the passes into keep and remove.

After removing passes which we marked as remove after categorization, we found the result did not achieve our expectation. Therefore, we tried to remove more passes which might improve the property of the reduced program. The result became closer and closer to our expectation. After adjusting, we found the reduction setting that produced the reduced program satisfying our expectation.

C-Reduce has 135 passes. In the reduction setting, we remove 54 passes as following:

- Not rename variables and functions
  - *pass_clang: rename-fun, rename-param, rename-var; pass_clex: rename-toks*: Those passes rename the variables and functions to a simple name (eg. a, b, … for variables; fn1, fn2 … for functions)
  - *pass_balanced: square; pass_clang: aggregate-to-scalar, copy_propagation*: Those passes modify the names of variables and functions by removing or replacing the parts in square brackets.
- Keep original if-else dependency

    - *pass_clang: simplify-if*: This pass simplifies an if-else statement.
- Not merge functions
    - *pass_clang: callexpr-to-value, replace-callexpr*: Those passes replace a call expression with a value or variable. They merge functions by changing a function and function call to a variable.
    - *pass_clang: remove-nested-function*: This pass removes a nested function invocation from its enclosing expression. The transformation will create a temporary variable with the correct type, assign the return value of the selected nested function to the created variable, and replace the function invocation with this temporary variable.
    - *pass_lines: 0, 1, 2; pass_peep: a, c*: Those passes remove lines and cause functions to merge when the two functions are close. If we do not remove passes pass_lines: 0, 1, 2, functions will merge when the two functions are close.
- Keep original types of variables and functions
    - *pass_balanced: square; pass_clang*: aggregate-to-scalar, remove-array: Those passes make the array (eg. a[256]) become a variable (eg. a_256).
    - *pass_clang: copy_propagation*: Those passes replace the variable with a constant value.
    - *pass_clang: reduce-array-dim, reduce-array-size, reduce-pointer-level*: Those passes modify array or pointer by reducing their levels or size.
    - *pass_clex: rm-tok-pattern-4, rm-toks-[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 16]*: Those passes remove parts of a statement. They usually remove the type of a variable or function declaration.
- Keep the original value of variables
    - *pass_ints: a, b, c, d, e, 0*: Those passes change the values of integer constants.
- Not modify arguments of functions and calls
    - *pass_clang: local-to-global, para-to-global*: Those passes move the declarations of local variables or parameters of functions to global.
    - *pass_clang: lift-assignment-expr*: This pass lifts an assignment expression to an assignment statement.
    - *pass_balanced: parens, parens-inside, parens-to-zero, simplify-callexpr; pass_clex: delete-string*: Those passes modify the parts in paren.

3. Experiments and Analysis

Our experiments aim to answer the following questions:

- RQ1: How effectively can we reproduce failures in reduced programs using delta-debugging?
- RQ2: What are the performance and results of different reduction settings?

3.1.Set Up

To answer the research questions, we used C projects in BugBench [17] (gzip-1.2.4, man-1.5h1, ncompress-4.2.4, and polymorph-0.4.0), DBGBench [18], and CoreReBench [19] (we used the first listed buggy version of coreutils-7eff5901-6fc0ccf7 project). The error type of projects find.b445af98, find.24e2271e and coreutils-7eff5901-6fc0ccf7 are functional bug. A functional bug needs an assertion to trigger the fault. We generated an assertion for coreutils-7eff5901-6fc0ccf7 to trigger the functional bug. Triggering assertion failure is used as expected output when we write a test script for the reducer. While we failed to find assertions to trigger the fault for functional bug programs find.b445af98 and find.24e2271e, we could use the expected failing output of the program to run reduction.

To answer RQ1, we used two metrics: 1) trace size and the number of files related to the trace (for showing the complexity of the project) 2) program size after reduction and the time taken by reduction (for demonstrating the delta debugging on trace). We used the trace generation tool trace-pintool [13][14] to generate traces. We calculated the length of the output from trace-pintool as trace size and reported the number of files related to trace in Table 1. We used the newest version of C-Reduce [9] to reduce the output of Helium and cloc [20] to count the lines of code as program size reported in Table 1.

To answer RQ2, we run C-Reduce with two reduction settings chosen from different versions of the reduction settings we tried. Reduction setting 2 removes all 54 passes to guarantee the property of the reduction result as expected while reduction setting 1 keeps the passes of *pass_lines* which cause functions merging when the two functions are close. In Table 1, we report the program size and time of reduction for each reduction setting.

3.2.Result

Table 1. Results of the experimental evaluation

Bug	Trace size	Number of files related to trace	Program size before reduction	Reduction setting 1		Reduction setting 2	
				size	time	size	time
gzip-1.2.4	80	2	192	20	3m31s	25	4m11s
man-1.5h1	609	9	1327	87	6m47s	116	8m58s
ncompress-4.2.4	51	1	116	24	58s	24	1m2s
polymorph-0.4.0	61	1	158	14	56s	17	1m14s
find.c8491c11	350	7	737	103	22m24s	129	23m58s
find.b445af98	1708	13	2404	380	30m19s	409	20m29s
find.24e2271e	1050	15	2078	290	18m9s	296	20m4s
coreutils-7eff5901-6fc0ccf7	166	3	346	126	4m52s	126	3m42s

Results of RQ1

Table 1 shows that we totally reduced 74.38% of code with reduction setting 1 and 71.98% of the code with reduction setting 2. In general, the program with a larger trace takes more time for the reduction. The trace of a small project is related to 1 original c file. However, the trace of the larger project is related to around 15 original c files. If we manually analyze those c files based on the trace for debugging, it will take time and be inefficient. Using a delta debugging tool to reduce the program generated from trace saves much time.

In addition, we tried to directly apply delta debugging on our smallest project ncompress-4.2.4 using C-Reduce with reduction setting 2. ncompress-4.2.4 only contains only one source code file (C-Reduce works on a file at a time). However, compared with applying delta debugging on trace, it took about 3 times of our time, and the reduced program is larger. That is, the trace-based reduction helps narrow down the fault-relevant statements.

Our approach handles functional bugs in addition to memory bugs. Project coreutils-7eff5901-6fc0ccf7 contains a functional bug which can be triggered by an assertion. We used the assertion as a test oracle. Projects find.b445af98 and find.24e2271e do not have the fault-triggering assertions. We used their test input and output as oracles.

Results for RQ2

Compared with the reduction size with setting 2, the reduction with setting 1 reduces 8.58% more in total, and 11.49% in average. The reduction time is irrelevant to the number of passes in the setting. The reduction setting 2 has less passes than the reduction setting 1. But the reduction time with setting 2 is not always shorter than the reduction time with setting 1.

```c
#include "he_gzip.h"
#include <assert.h>
static void helium_treat_file(char *iname) {
  {
    strcpy(ifname, iname);
  }
}
int main(int argc, char **argv) {
  {
    int file_count;
    file_count = argc - optind;
    if (file_count != 0) {
      {
        while (optind < argc) {
          helium_treat_file(argv[optind++]);
        }
      }
    }
  }
}

```

(a) Reduction setting 1

```c
#include "he_gzip.h"
#include <assert.h>
static int helium_get_istat(char *iname, struct stat *sbuf);
static void helium_treat_file(char *iname) {
  {
    if (helium_get_istat(iname, &istat) != 0)
      ;
  }
}
static int helium_get_istat(char *iname, struct stat *sbuf) {
  strcpy(ifname, iname);
}
int main(int argc, char **argv) {
  {
    int file_count;
    file_count = argc - optind;
    if (file_count != 0) {
      {
        while (optind < argc) {
          helium_treat_file(argv[optind++]);
        }
      }
    }
  }
}

```

(b) Reduction setting 2

Figure 4. Reduced programs of gzip-1.2.4 with two reduction settings

We manually validate the reduced programs of different reduction settings with original programs. We found reduction setting 2 kept failure-relevant statements and the dependencies between the failure-relevant statements, and reduction setting 1 satisfied requirements except not merging functions. Due to merging functions, the size of the reduced program which used reduction setting 1 is smaller than the size of reduced program with reduction setting 2 for most projects. For example, Figure 4a is the reduction result of C-Reduce with reduction setting 1. Figure 4b is the reduction result with reduction setting 2. In the original code, the statement causing the fault is *strcpy(ifname, iname);* which is in function *get_istate*. As we can see, the reduction setting 1 merges function *helium_treat_file* and function *helium_get_istate* which are close in the program before reduction. Reduction 2 keeps the dependencies of the functions. But the size of the reduced program with setting 1 is not always smaller than the size with setting 2. For ncompress-4.2.4 and coreutils-7eff5901-6fc0ccf7, the reduced programs with two settings have the same size because there is no function merging during reduction with reduction setting 1.

4. Related Work

Delta-debugging [1] has been used to minimize inputs to locate the bugs in the program. Misherghi and Su [2] proposed Hierarchical Delta Debugging (HDD) which ran faster and produced better reduction on tree structured inputs. Hodován and Kiss [3] proposed an approach using extended context-free grammars with HDD to help spread the adaptation of HDD in practice. They[4][5] focused on improving HDD while keeping its minimality guarantees intact. Herfert et al. [6] proposed the Generalized Tree Reduction (GTR) algorithm which reduces tree-structured test inputs. A given test input is reduced by hierarchical tree transformations. The algorithm applies the transformation chosen from Delta Debugging and backtracking-based search.

Minimizing unit tests by using Delta Debugging makes the unit tests become easier to understand, and keep all failure-relevant steps to reproduce. Leitner et al. [7] proposed a technique which automatically reduces the sequence of failure-inducing method calls by applying the combination of static slicing and Delta Debugging. Burger and Zeller [8] proposed JINSI which takes a single failing run, records and minimizes the interaction between objects to the set of failure-inducing calls. It generates a minimal unit test that reproduces the failure.

Delta Debugging is implemented to build more effective, more efficient, and more general approaches for test cases reductions. Regehr et al. [9] proposed C-Reduce which took a large C/C++ program and automatically reduced it to a much smaller program that had the same property. Although C-Reduce provided high-quality reductions for C/C++ programs, it lacked generality and took much reduction time. Sun et al. [10] proposed a framework Perses for test program reduction. Perses took the formal syntax of a programming language as a guide for reduction. In each iteration of reduction, it ensured that only smaller and syntactically valid program variants were considered so the syntactically invalid program variants could be avoided.

5. Limitations

For different projects, programs generated from trace might have different reasons for failing the fault reproduction. It might be time-consuming to fix. For example, the programs of *find* projects do not reproduce the fault from our program but from the original program. Because those projects use a function pointer which the function call points to the original function. We need to manually change this function pointer to an expression calling the helium function. After fixing, the fault reproduced with helium functions, and we applied reduction on the programs.

The reduction setting guarantees the reduction result of the current projects. However, there might be new unexpected behavior in the result of programs with larger traces. The reduction setting might need to be modified to solve the new unexpected behaviors.

6. Conclusions and Future Work

We presented an approach which can reproduce the failure by delta-debugging on traces. It generates a dynamic trace of fault program with failure inducing input at first, and then uses Helium to generate an executable program from the trace. Finally, we use delta debugging tool C-Reduce to apply reduction on the output of Helium. We used two reduction settings to produce two kinds of reduced programs. The reduced program with reproduction setting 1 has function merging but satisfies other goals. The size of the reduced program with reproduction setting 1 is smaller or equal to the size of the reduced program with reproduction setting 2. The reduced program with reproduction setting 2 satisfies our requirement. Our experiment shows that we totally reduced 74.38% of code with reduction setting 1 and 71.98% of the code with reduction setting 2. The reduction with setting 1 reduces 8.58 % more of codes in total and 11.49% more of codes in average than the reduction with setting 2 does. In the future, we will use our approach to generate bug benchmarks and apply the reduced program on test input generation and automatic program repair (APR).

Acknowledgements

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research. First and foremost, Dr. Wei Le for her guidance, patience and support throughout this research. I would also like to thank Helium team members Ashwin Kallingal Joshy, Benjamin Steenhoek and Xiuyuan Guo for assistance during the entire research and development process. Additionally, I would like to extend my gratitude towards my committee members Dr. Myra Cohen and Dr. Hongyang Gao.

References

[1] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? SIGSOFT Softw. Eng. Notes 24, 6 (Nov. 1999), 253–267. DOI:https://doi.org/10.1145/318774.318946
[2] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In Proceedings of the 28th international conference on Software engineering (ICSE '06). Association for Computing Machinery, New York, NY, USA, 142–151. DOI:https://doi.org/10.1145/1134285.1134307

[3] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2016). Association for Computing Machinery, New York, NY, USA, 31–37. DOI:https://doi.org/10.1145/2994291.2994296
[4] R. Hodován, Á. Kiss and T. Gyimóthy, "Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging," 2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST), Buenos Aires, Argentina, 2017, pp. 23-29, doi: 10.1109/AST.2017.4.
[5] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical Delta debugging algorithm. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST 2018). Association for Computing Machinery, New York, NY, USA, 16–22. DOI:https://doi.org/10.1145/3278186.3278189
[6] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, 861–871.
[7] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient unit test case minimization. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07). Association for Computing Machinery, New York, NY, USA, 417–420. DOI:https://doi.org/10.1145/1321631.1321698
[8] Martin Burger and Andreas Zeller. 2011. Minimizing reproduction of software failures. In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). Association for Computing Machinery, New York, NY, USA, 221–231. DOI:https://doi.org/10.1145/2001420.2001447
[9] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. SIGPLAN Not. 47, 6 (June 2012), 335–346. DOI:https://doi.org/10.1145/2345156.2254104
[10] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). Association for Computing Machinery, New York, NY, USA, 361–371. DOI:https://doi.org/10.1145/3180155.3180236
[11] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. 2014. Using test case reduction and prioritization to improve symbolic execution. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 160–170. DOI:https://doi.org/10.1145/2610384.2610392
[12] Scott McPeak, Daniel S. Wilkerson, and Simon Goldsmith. https://github.com/csmith-project/creduce/tree/master/delta, accessed: 2020-12-07.

[13] Benjamin Steenhoek. trace-pintool. https://github.com/bstee615/trace-pintool, accessed: 2020-11-24.
[14] Benjamin Steenhoek. pal-tools. https://github.com/bstee615/pal-tools, accessed: 2021-01-26.
[15] F. Tip. A survey of program slicing techniques. Journal of programming languages, 3(3):121–189, 1995.
[16] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating FailureInducing Input. IEEE Trans. Softw. Eng. 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498
[17] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools."
[18] Böhme, M., Soremekun, E., Chattopadhyay, S., Ugherughe, E., & Zeller, A. (2017). Where is the Bug and How is it Fixed? An Experiment with Practitioners. In Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (pp. 1-11).
[19] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 105–115
[20] Al Danial (2006). cloc (1.74). https://github.com/AlDanial/cloc#license-