CS 453/698 Assignment 2

TA: Mehrad Haghshenas (m3haghsh@uwaterloo.ca)
Office hours: Wednesdays 2pm–3pm, online via BBB[1]

Due date: June 20, 2025

Program Reduction and Bug Discovery

Introduction

Modern software – including critical systems and compilers – are expected to be secure and trustworthy. *Software testing* is the process of verifying that software behaves as intended. It is a practice aimed at improving code-reliability by identifying bugs and defects through execution under specific scenarios. This process can be time-consuming and error-prone when done manually, motivating the need for automated techniques. Two powerful approaches have emerged to automate parts of this process: *differential testing* (randomized testing or fuzzing across multiple implementations) and *test-case reduction* (automatically simplifying failure-inducing inputs). In this assignment, we will explore these techniques in more detail.

Accordingly, *differential testing* uses multiple versions of a software to find discrepancies. For example, `csmith` is a tool that randomly generates C programs as test cases for compilers. By comparing the outputs of different compilers in the same random program, `csmith` can detect compiler bugs when one compiler crashes or produces different results than others. Note that in case there are multiple compilers, the output generated by most of the compilers is considered the correct output, and thus, any compiler which produces a different output is deemed to contain a bug. For more details, please read the references. Two key aspects of *csmith* are 1) that it is much more expressive compared to previous similar tools. 2) avoids undefined or unspecified behavior in the C programs it generates. This is crucial because undefined and unspecified behavior is treated differently in different implementations and would make the differences in compiler output useless.

While random testing can find bugs, the resulting test programs are often extremely large and complicated. To report the bugs and diagnose the underlying issue, developers need a minimal test case. This is where *test-case reduction* tools come into play. Delta debugging is an algorithmic approach (proposed by Zeller et al.) to automatically simplify a failure-inducing input. The basic delta debugging algorithm repeatedly removes parts of the input and checks whether the bug still occurs. If removing a certain subset of the input causes the failure to disappear, that subset is necessary; otherwise, it can be discarded. By systematically narrowing down the input, delta debugging finds a program where no single component can be removed without losing the bug. This process reduces the effort to manually isolate the cause of a failure. However, the algorithm might get stuck in a local minimum containing unnecessary code. `creduce` is a more advanced test-case reduction tool. While the classic delta works by removing chunks of text, creduce performs many semantic-aware transformations. This approach yields smaller and more useful results for test programs. Furthermore, creduce's reductions are validity-guided, meaning that the transformations do not introduce undefined behavior in the test cases.

Why do these tools matter for secure and trustworthy software development? First, they dramatically improve the efficiency and speed of debugging complex systems. Second, minimized test cases are critical for documenting when reporting a bug in an open-source project.

[1] Access code will be posted on Piazza.

In this assignment, you will work through a series of exercises to practice these concepts:

1. Part 1 involves a hands-on use of a delta debugging tool to reduce a failing input that triggers a bug in a C program.
2. Part 2 involves implementing a small fuzzer and using differential testing to identify discrepancies between multiple programs. The primary goal is to find the faulty implementation based on divergent outputs; the secondary is to minimize the input that exposed the bug using *creduce*.

By the end of the assignment, you will have gained experience with automated testing tools and how they contribute to building reliable systems.

Environment Setup

In Assignment 1, a virtual environment on the *ugster* machines was provided to responsibly experiment with security flaws. Theoretically, after completing Assignment 1, you should already have access to the virtual machine. In that case, you may continue using the currently running VM, or you can destroy it and build a new one from scratch. If you don't have access to the VM, you must (re)register an account on Virtus using your WatIAM and wait for approval. You can visit Virtus CS453 for instructions on creating a new VM, destroying an existing VM, or SSH-ing into a VM.

Once inside the VM, you should run the following command from the home directory to initialize the VM for the second assignment:

```
wget -q \
     -O a2_setup \
http://ugster72c.student.cs.uwaterloo.ca/a2/a2_setup.sh && \
chmod +x a2_setup && \
./a2_setup
```

This ensures that all tools—such as `csmith`, `creduce`, and `delta`—along with their required dependencies, are automatically installed on your system.

1. Delta Debugging for Test-Case Reduction

The objective of this assignment is to apply the automated method known as *delta debugging* to simplify a failure-inducing input for a given program. We provide a buggy program `bug.c` and a large test input `large_input.txt` on which this program fails. The file is *not* a minimal witness: it contains many superfluous edges. Your task is to use the delta tool to reduce `large_input.txt` that still triggers the *same* underlying reason. After obtaining the minimal failing test case, you will analyze the program's code to determine the cause of the bug, identify what kinds of inputs trigger the bug, and propose a fix.[2]

After setting up the environment, you should already have access to *delta*. You can use the following script to retrieve the faulty algorithm and the test file that exposes the bug:

```
wget -q \
     -O a2_part1 \
http://ugster72c.student.cs.uwaterloo.ca/a2/a2_part1.sh && \
chmod +x a2_part1 && \
./a2_part1
```

Note that this will give you the `bug.c` and `large_input.txt` files. You must first write an *interestingness.sh* script — a shell script that takes an input test case and returns exit code 0 if the bug is triggered and a non-zero exit code otherwise. Note that the bug must be triggered for the *exact same reason* (i.e., the same *type* of failure or behavior). Using the *delta* tool, generate a `min_failing_test.txt` — a minimal input file that causes the bug in `bug.c`.[3] Finally, write a `writeup.txt` — a short document answering the following: (1) Explain the root cause of the bug in `bug.c`. (2) Characterize all input lists on which `bug.c` will fail for the same reason. (3) Describe a fix that would resolve the bug.

[2] This assignment has been adopted from the *CS 520* course offering of UMass Amherst in Fall 2019. The course can be found on course website.

[3] You can invoke the delta tool by using the following command in the terminal `delta -test=interestingness.sh -cp_minimal=min_failing_test.txt < large_input.txt`

2. Differential Testing with Arithmetic Expression Evaluators

In this section, we provide three binaries `x1, x2, x3`, which can be obtained running the following script:

```
wget -q \
     -O a2_part2 \
http://ugster72c.student.cs.uwaterloo.ca/a2/a2_part2.sh && \
chmod +x a2_part2 && \
./a2_part2
```

These three programs implement different evaluators for arithmetic expressions. Each supports integer expressions with the operators +,-, ∗,/, ˆ (exponentiation), and parentheses. Specifically, each program reads an expression (as a single-line string) from standard input and prints the computed integer result. All three implementations handle the same grammar and operations so their outputs *should* match. There are several important points about these programs, which we will outline below:

- To emphasize, all three program represent arithmetic expressions evaluators.
- In theory, all three programs should accept the same set of valid arithmetic expressions and reject invalid ones. That is, they should conform to a consistent grammar. For example:
  - `x y` is invalid because it places two operands consecutively without an operator.
  - `x+*y` is invalid because it places two operators consecutively, violating proper syntax.

  In short, the BNF grammar of all three implementations is supposed to be as following:

```
<Expr>   ::= <Term> { ("+" | "-") <Term> }
<Term>   ::= <Power> { ("*" | "/") <Power> }
<Power>  ::= <Factor> [ "^" <Power> ]
<Factor> ::= <Number> | "(" <Expr> ")"
<Number> ::= <Digit> { <Digit> }
<Digit>  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

However, one of the implementations contains a subtle bug. By performing *differential testing* – running all implementations on many random inputs and comparing their outputs – you should detect the discrepancy caused by the bug. This approach uses the fact that any divergence in results (with the majority agreeing) signals a likely bug in the outlier implementation.[4]

Specifically, to find such a bug, we use fuzzing with differential testing. The idea is to automatically generate many random expressions and evaluate each on all three programs, then compare the outputs:

- If all three agree, we move on.
- If one implementation disagrees with the other two, we have found a potential bug.

[4]Note that there may be more than one bug, but you are only required to find one. Importantly, discrepancies caused by unspecified behavior — such as providing invalid syntax or dividing by zero— do not count. Even if our binaries do not crash or report an error, such cases are excluded. This design choice intentionally mirrors how undefined behavior can lead to divergent outputs across compiler implementations, as seen in tools like csmith.

Differential testing is powerful here because we don't need to know the expected correct result for each expression – we only need to check for consistency across implementations. The two agreeing results serve as a baseline, and the differing result is flagged as a bug in that implementation. This is a *plurality voting* kind of strategy which has also been used in related work. To sum up, your task is the following:

1. First, implement a random input generator (fuzzer) that meets two essential requirements:

   (1) **Expressiveness**: it must cover a wide variety of valid inputs to exercise different code paths. As a rule of thumb, hard-coded inputs are discouraged, as they limit coverage and reduce the likelihood of uncovering subtle bugs.

   (2) **Safety**: it must avoid inputs that trigger undefined or unspecified behavior—otherwise, differential testing breaks down as each implementation may handle such cases arbitrarily. Examples include division by zero or semantically invalid programs as each implementation may handle these situations differently.

   This list of concerns is illustrative, not exhaustive. In short, your random generator should produce syntactically valid strings of expressions.

2. After implementing the fuzzer, apply *differential testing* to identify discrepancies in behavior across the different implementations. This should ideally be an automated script running your fuzzer until the input that triggered the divergent output is generated. Report the specific binary along with the input that triggered the divergent output.

3. Finally, use *creduce* to minimize the test input that exposes the bug, producing a minimal test case.

3. Deliverables

Submit a single archive named `<userid>_assignment2.zip` (replace `<userid>` with your WatIAM). The archive must contain two folders, one for part one named as `part_one` and another for part two named as `part_two`:

- In the folder corresponding to Part One, you should submit 3 files:
  - `interestingness.sh` — A shell script that takes an input test case and returns exit code 0 if the same bug is triggered and non-zero otherwise.
  - `min_failing_test.txt` — A minimal input file that causes the bug in `bug.c`. Note that this needs to be obtained by running *delta*.
  - `writeup.txt` — A short document answering the following:
    1. Explain the root cause of the bug in `bug.c`.
    2. Characterize all input lists on which `bug.c` will fail for the same reason.
    3. Describe a fix that would resolve the bug.
- In the folder corresponding to Part Two, include the following files:
  - `fuzzer.c` or `fuzzer.py` — your random input generator.
  - `interestingness.sh` — After finding the input that reveals a discrepancy between implementations, write this script to minimize the failure-inducing input.
  - `writeup.txt` — This should include the following:
    * A brief explanation of how you ensured both expressiveness (diverse inputs) and safety (validity and termination) in your fuzzer design.
    * Providing a demo/screenshot demonstrating your fuzzer running and identifying a test case where the outputs of the implementations differ.
    * The specific input that exposes the discrepancy between the implementations.
    * The name of the buggy implementation (binary).
    * The minimized version of the triggering input after using *creduce*.
    * Which input patterns trigger the bug? Explain.

4. Grading & Submission

This assignment is worth 25% of the course grade. Your submission should be in your own words. Ensure you cite any external resources if used. Plagiarism or directly copying someone else's work is strictly prohibited. You may use large language models to assist with answering the questions, but clearly indicate the context in which they were used. Please make the final submission on LEARN by the due date. Lastly, the entire assignment has 30 marks in total. The following is the grading scheme for the assignment:

1. Section 1 (12 marks):
   (a) **interestingness.sh script (4 marks)**
   (b) **min_failing_test.txt (2 marks)**
      - 2 marks: A small, valid input list that reliably triggers the bug.
   (c) **Write-up (6 marks)**
      - 2 marks: Clear explanation of the root cause in the source code.
      - 2 marks: Correct and complete characterization of all inputs affected.
      - 2 marks: Sensible and correct fix for the bug.
2. Section 2 (18 marks):
   (a) **Fuzzer Implementation (8 marks)**
      - 2 marks: Input expressiveness — handles a wide variety of valid inputs and syntax structures.
      - 2 marks: Input safety — avoids crashes, invalid syntax, and undefined behavior.
      - 4 marks: Demonstrates that the fuzzer successfully triggered a test input where the implementations produced different outputs (i.e., exposed a bug).
   (b) **Interestingness Test Script (4 marks)**
   (c) **Write-up and Supporting Files (6 marks)**
      - 1 mark: Clear explanation of how expressiveness was achieved.
      - 1 mark: Clear explanation of how input safety was maintained.
      - 1 mark: A valid input that triggers a discrepancy.
      - 1 mark: Correct identification of the buggy implementation.
      - 1 mark: A reduced version of the triggering input that still reproduces the bug.
      - 1 mark: Defining the *input patterns* that trigger the different behaviour in the implementations.

Good luck, and have fun exploring these tools!

References

[1] Yang, Xuejun, et al. ”Finding and understanding bugs in C compilers.” Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011.

[2] Zeller, Andreas. ”Yesterday, my program worked. Today, it does not. Why?.” ACM SIGSOFT Software engineering notes 24.6 (1999): 253-267.

[3] Regehr, John, et al. ”Test-case reduction for C compiler bugs.” Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. 2012.

[4] Groce, Alex, et al. ”Cause reduction: delta debugging, even without bugs.” Software Testing, Verification and Reliability 26.1 (2016): 40-68.

[5] Kumbhar, P. (2024, April 4). creduce: Systematically Tackling (Not Only) Compiler Bugs - Performance engineering. Performance Engineering. https://pramodkumbhar.com/2024/01/c-reduce-systematically-tackling-not-only-compiler-bugs-in-hpc-too/

[6] Dsw. GitHub - dsw/delta: Delta assists you in minimizing “interesting” files subject to a test of their interestingness. GitHub. https://github.com/dsw/delta?tab=readme-ov-file

[7] csmith-Project. GitHub - csmith-project/creduce: creduce, a C and C++ program reducer. GitHub. https://github.com/csmith-project/creduce