Poster: Debugging Inputs

Lukas Kirschner
CISPA – Helmholtz Center for Information Security
Saarbrücken, Germany
s8lukirs@stud.uni-saarland.de

Ezekiel Soremekun
CISPA – Helmholtz Center for Information Security
Saarbrücken, Germany
ezekiel.soremekun@cispa.saarland

Andreas Zeller
CISPA – Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.saarland

ABSTRACT

Program failures are often caused by *invalid inputs*, for instance due to input corruption. To obtain the *passing input*, one needs to *debug the data*. In this paper we present a generic technique called *ddmax* that (1) *identifies* which parts of the input data prevent processing, and (2) *recovers* as much of the (valuable) input data as possible. To the best of our knowledge, *ddmax* is the first approach that fixes faults in the input data without requiring program analysis. In our evaluation, *ddmax* repaired about 69% of input files and recovered about 78% of data within one minute per input.

1 INTRODUCTION

Several techniques for automated debugging have been developed in research and practice [5, 7]. Most of these techniques focus on *program code,* i.e. identifying fault locations in the code and synthesizing fixes for this code. However, program failures are often caused by invalid inputs, such *invalid inputs are prevalent in the wild*. In fact, four percent of input files in the wild are invalid, they could not be processed either by their grammar or program(s) [4].

If input data is corrupted, the easiest remedy is to use a backup. But if a backup does not exist, one may want to recover *as much data as possible* from the existing data—or in other words, *debug the data*. However, debugging input data can be very tedious. For instance, consider a large Wavefront OBJ file with one corrupted line (e.g. an invalid character inside a "usemtl" statement). To fix such an error by hand, one would have to scroll through thousands of lines of code to find the single corrupted character [4].

In this paper, we introduce a *generic input repair method* which does precisely this: Given a failure-inducing input, it automatically produces a *1-maximal* subset in which every single element (character) from the original input is added without producing a failure. We obtain this method by *inverting* the original *delta debugging* (*ddmin*) algorithm [8] into a *ddmax* algorithm, repeatedly adding data from the originally failing input (first larger pieces, then smaller pieces) as long as the failure does not occur.

The *ddmin* algorithm [8] focuses on identifying error causes in the input by simplifying a failure-inducing input down to a minimum that reproduces the error. Unfortunately, *ddmin* is not a good fit for input repair: It produces the smallest subset of the input that also produces an input error—typically a single character.

ICSE '20 Companion, October 5–11, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3390797

```
{ "item": "Apple", "price": **3.45 }
```

Figure 1: Failing JSON input

```
{ "item": "Apple", "price" 3.45 }
```

Figure 2: Failing JSON input with missing colon

However, *ddmax* identifies the invalid input fragment quickly (for debuggers) while also preserving a maximum of content (for users).

For instance, consider Figure 1, a JSON input with a syntax error; *ddmax* produces the "repaired" (passing) input subset — { "item": "Apple", "price": 3.45 }, where the faulty characters "**" are removed. The difference between the repaired input and the original input, gives a precise diagnosis of the cause of the failure (for debuggers), and serve as a debugging aid for developers. In contrast, *ddmin* produces the reduced input consisting of a single character ("{") which produces a syntax error. This *ddmin* output is neither useful for diagnosis nor data recovery.

2 APPROACH

We present two variants of the *ddmax* algorithm: (1) *lexical ddmax* which repairs arbitrary invalid inputs at the character level, and (2) *syntactic ddmax* which perform input repair on the parse tree.

Lexical ddmax. The *ddmax* algorithm uses the same setting as *ddmin*; however, rather than *minimizing* the failure-inducing input $c_{\text{✘}}$, it starts with a passing input $c'_{\text{✔}} = c_{\text{✔}}$. Like *ddmin*, it assumes for simplicity that $c_{\text{✔}} = \emptyset$ holds. It then *maximizes* $c'_{\text{✔}}$, systematically minimizing the difference between $c'_{\text{✔}}$ and $c_{\text{✘}}$ using the same techniques as *ddmin* (first progressing with large differences, then smaller differences), until every remaining difference would cause $c'_{\text{✔}}$ to fail. This makes *ddmax* act in exact symmetry to *ddmin*, and complements the original definitions of *dd* and *ddmin* [8].

Syntactic ddmax. We introduce the *syntactic ddmax* algorithm, which improves the performance of *ddmax* using the input grammar. The key insight is to execute *ddmax* on the *parse tree* of the input, instead of the input characters. This improves the runtime and general performance of the *ddmax* algorithm, since it can easily exclude corrupted parse tree nodes or subtrees during test runs. The knowledge of the input structure ensures that the resulting recovered inputs are syntactically valid. This helps in the case of syntax errors and multiple corruptions on the input (structure).

For instance, consider the corrupted JSON input in Figure 2. Repairing this input using the *lexical ddmax* algorithm results in the JSON input — { "item": "Apple" }, which would take over 100 test runs. For this example, *syntactic ddmax* reduces the number of test runs of *ddmax* to nine, improving performance by ten fold.

3 EXPERIMENTS

We evaluate our approaches on three input formats (namely, JSON, Wavefront OBJ and DOT [1]) using eight subject programs, namely `Blender`, `Assimp`, `Appleseed`, `JQ`, `JSON-Simple`, `Minimal-JSON`, `Graphviz`, and `Gephi` [4]. The *test oracle* for *ddmax* is a *crashing oracle*. An input is treated as *invalid* if it crashes the subject program, produces no output or the program runs for more than 10 seconds.

Protocol: We collected a corpus of 7835 unique input files crawled from *Github* [2] and filtered them into sets of valid files and invalid files. Then, we randomly selected and mutated 50 valid *crawled files* to produce an additional set of corrupted input files. We fed an invalid file to each subject program (called *Baseline*), and the *ANTLR* parser framework [6]. *ANTLR* executed its default *error recovery strategy* while generating a parse tree for the input [3]. Finally, we fed the invalid file to *ddmax* to test the input under repair repeatedly using the feedback from the subject program.

RQ1 Effectiveness: *How effective is lexical and syntactic ddmax in repairing invalid input documents within a time budget of one minute per file?* For all input formats, *ddmax* repaired 69% of invalid inputs within a time budget of one minute per input. *Lexical ddmax* repaired about two-thirds (66%) of all invalid inputs and significantly outperforms both the baseline and ANTLR. *Syntactic ddmax* repaired about three-quarters (73%) of all invalid inputs, it is about 10% more effective than *lexical ddmax*. It significantly outperformed both the built-in repair strategies of the subject programs and *ANTLR*, it repaired five times as many files as the subject programs, and almost twice as many files as *ANTLR*. This confirms that using the input grammar improves the performance of *ddmax*.

RQ2 Data Recovery: *How much data is recovered by ddmax?* Overall, *ddmax* has a high data recovery rate, it recovered about 78% of data. On average, *syntactic ddmax* (89%) has a higher data recovery rate in comparison to *lexical ddmax* (58%). For all invalid inputs, the *baseline* and *ANTLR* maintain an almost perfect data recovery rate, but repair signifiantly fewer inputs (see RQ1).

RQ3 Diagnostic Quality: *How effective is ddmax in diagnosing the root cause of invalid inputs, especially in comparison to ddmin?* On average, only one-eighth (12%) of a *ddmin* diagnosis contains the minimal failure cause and about one-third (33%) of *ddmin* diagnosis contains the maximal passing input. Given that *ddmax* was completely executed without a timeout, the repair of *ddmax* is the *maximal passing input* and *ddmax* diagnosis is the *minimal failure cause*. As expected *ddmin* diagnosis is significantly larger (21 times more) than the *ddmax* diagnosis, hence, it contains a significant amount of noise (33%), i.e. portions of the maximal passing input.

4 LIMITATIONS

Both lexical and syntactic *ddmax* are limited in the following ways:

Context Sensitivity. If the input format has several context-sensitive dependencies, such as checksums, hashes, encryption, or references, a strict lexical or syntactical subset may not be sufficient to produce a valid repair. Learning and checking for such input context can improve the performance of *ddmax*.

Data repair, not information repair. Even though the resulting *ddmax* repair may be lexically or syntactically close to the original input, it can have very different semantics. We are exploring how to extend *ddmax* to account for (domain-specific) semantics.

Input Semantics. Although, *ddmax* obtains some "semantic" information from the feedback of the subject program itself, this feedback is limited to failure characteristics, i.e. "pass" or "fail". However, it is possible to extend *ddmax* to include (domain-specific) semantic checks, which could either be defined as the execution of specific program artifacts such as a specific branch, or programmatically defined by a developer (e.g. as an expected program output).

Multiple repairs. If there are multiple ways to repair an input, ddmax will produce only one of them. This property is shared with *dd* and *ddmin*, which also pick a local minimum rather than searching for a global one. However, it would be easy to modify *ddmax* to assess all alternative repairs rather than the first repair.

5 CONCLUSION AND FUTURE WORK

We have presented *ddmax*— the first *generic* technique for automatically repairing failure-inducing inputs, it recovers a maximal subset of the input that can still be processed by the program at hand. Our work opens the door for a number of exciting research opportunities in the following areas:

Synthesizing input structures. We are investigating grammar-based production strategies to synthesize missing input fragments, such input synthesis can improve *ddmax*'s performance.

Hybrid repair. Both *ddmax* variants can be combined such that after syntactic *ddmax* is executed on the parse tree, lexical *ddmax* further repairs the bytes in the faulty nodes. This improves the effectiveness of *ddmax* and reduces the number of iterations.

Semantic Input Repair. The *ddmax* test oracle can be extended to include checks for desirable "semantic" properties. For instance, to check if some function or output is observed during repair. These checks would ensure that the resulting maximized passing input is semantically similar to the original input and avoids the failure.

Fuzzing. Both variants of *ddmax* can be applied to improve software fuzzing. For instance, mutational fuzzers often produce malformed inputs that can be repaired by *ddmax*, to ensure validity.

A replication package of *ddmax*'s evaluation is available at:

https://tinyurl.com/debugging-inputs-icse-2020

REFERENCES

[1] GitHub. 2018. Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4

[2] GitHub Inc. 2018. REST API v3. https://developer.github.com/v3/

[3] ANTLR 4.7.1 API JavaDocs. 2018. Class DefaultErrorStrategy. https://www.antlr.org/api/Java/org/antlr/v4/runtime/DefaultErrorStrategy.html

[4] L Kirschner, E Soremekun, and A Zeller. 2020. Debugging Inputs. In *The 42nd International Conference on Software Engineering (ICSE 2020)*.

[5] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 269–278.

[6] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. *SIGPLAN Not.* 46, 6 (June 2011), 425–436. https://doi.org/10.1145/1993316.1993548

[7] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[8] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498