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

When a program fails to process an input, it need not be the program code that is at fault. It can also be that the input data is faulty, for instance as result of data corruption. To get the data processed, one then has to *debug the input data*—that is, (1) *identify* which parts of the input data prevent processing, and (2) *recover* as much of the (valuable) input data as possible. In this paper, we present a general-purpose algorithm called *ddmax* that addresses these problems automatically. Through experiments, *ddmax* maximizes the subset of the input that can still be processed by the program, thus recovering and repairing as much data as possible; the difference between the original failing input and the "maximized" passing input includes all input fragments that could not be processed. 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.

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

Figure 1: Failing JSON input

```
{
```

Figure 2: Failing input reduced with *ddmin*

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

Figure 3: Failing input repaired with *ddmax*

```
**
```

Figure 4: Difference between failing and repaired input

1 INTRODUCTION

In the last decade, techniques for automated debugging and repair have seen great interest in research and practice. A recent survey [44] lists more than 100 papers on automatic fault localization and repair. Recently, social networking giant Facebook provided developers with automatically generated repair suggestions for every failure report of its apps [32]. Almost all of these techniques focus on *program code,* attempting to identify possible fault locations in the code and synthesizing fixes for this code. However, when a program fails on some input, it need not be the program code that is at fault. Hardware failures, hardware aging, transmission errors may all cause data to get corrupted. In computer hardware, radiation can impact memory cells, leading to bit flips and again data corruption. And finally, data can be corrupted through software bugs, with the processing software writing out malformed or incomplete data. If data is corrupted, the easiest remedy is to use a backup. But if a backup does not exist (or is too old, or fails to be processed), one may want to recover *as much data as possible* from the existing data—or in other words, *debug the data.*

Some programs come with application-specific means to recover data. Input parsers can recover from syntactical errors by applying sophisticated recovery strategies; in a programming language, this may involve skipping the current statement or function and resuming with the next one [21]. When detecting a corrupted or incomplete file, Microsoft Office programs may attempt to recover from the error, using a number of undisclosed approaches [43]. When a program does *not* implement a good recovery strategy, though, users are left to their own devices, using general-purpose editors to identify file contents and possible corrupted parts.

As listed above, general-purpose automated debugging techniques focus on faults in code and do not provide much help in such situations, as they would regularly identify the input parser and its error-handling code as being associated with the fault. The *delta debugging* (*ddmin*) algorithm [45], however, focuses on identifying error causes in the input; in repeated runs with reduced inputs, it simplifies a failure-inducing input down to a minimum that reproduces the error. Unfortunately, delta debugging is not a good fit: applied to invalid inputs, it produces the smallest subset of the input that also produces an input error—typically a single character. As an example, consider Figure 1, a JSON input with a syntax error; *ddmin* produces the reduced input in Figure 2, consisting of a single `{` character, which also produces a syntax error. This is neither helpful for diagnosis nor a basis for data recovery.

In this paper, we introduce a *generic input repair method* that automatically (1) *identifies* which parts of the input data prevent processing, and (2) *recovers* as much of the (valuable) input data as possible. Like *ddmin*, our approach runs the program under test repeatedly with different subsets of the input, assessing whether the subset can be processed or not. Also, it does not need any kind of program analysis and can thus be used in a wide range of settings. Unlike *ddmin*, however, which aims at minimizing the failure-inducing input, our *ddmax* algorithm aims at *maximizing the passing input.* Its result is a subset of the input that (1) can be successfully processed and (2) is *1-maximal:* no further element from the failing input can be added without causing the input to become invalid again.

ICSE '20, May 23–29, 2020, Seoul, South Korea
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Applied on our example from Figure 1, *ddmax* produces the "repaired" (passing) input subset in Figure 3, in which the confounding `**` characters (and nothing else) are removed. The difference between the original input (Figure 1) and the repaired input (Figure 3), listed in Figure 4, actually makes a precise diagnosis of the failure cause and can be given to developers for further debugging steps.

Note that while *ddmax* recovers a maximum of *data,* it does not recover a maximum of *information*; in our example, we do not know whether `3.45` actually is the correct price. However, the repaired input can now be read and processed by the program at hand, enabling humans to read and check their document and engage into additional recovery steps.

The remainder of this paper makes the following contributions:

An empirical study of invalid inputs in practice. We evaluate the prevalence of invalid input in the wild (Section 2). We crawled thousands of input files from *github* and determine the set of valid and invalid files. We find that invalid inputs are common in practice, about four percent (295 files) of all input files (7835 files) crawled from github were invalid.

Generic input repair with minimal data loss. We introduce the *ddmax* algorithm, automatically recovering a maximum of data from a given failure-inducing input (Section 3). To the best of our knowledge, *ddmax* is the first input repair technique that can be applied to arbitrary inputs and programs without additional knowledge on input formats or program code. In its evaluation on eight subjects and three input formats, using real-world invalid inputs as well as synthetic corruptions, we find that *ddmax* is effective: It repairs 69% of corrupted inputs and recovers about 78% of data, within a time budget of one minute per input.

An efficient syntactical input repair technique. We introduce a variant of *ddmax* that makes use of a *grammar* to parse inputs into derivation trees and to maximize inputs by pruning parts of the tree that could not be read (Section 4); this vastly speeds up input repair. In its evaluation, syntactic *ddmax* is faster and more efficient than the lexical variant.

Identifying faults in input data. The difference between the "repaired" input by *ddmax* and the original input contains all parts of the input that prevented the data from being processed in the first place. Section 5 shows that this difference precisely characterizes the fault in the input.

After discussing limitations (Section 6), threats to validity (Section 7) and related work (Section 8), we close with conclusion and future work (Section 9).

2 PREVALENCE OF INVALID INPUTS

Before we start *repairing* inputs, let us first answer the question of how *relevant* the problem is. Is it actually possible that some application cannot open a data file? And would there be files claiming to adhere to some format if in fact, they are not? To answer such questions, let us go and catch some invalid inputs in the wild.

2.1 Evaluation Setup

Subject Programs. In this paper, we use eight programs as test subjects, namely `Blender` [17], `Assimp` [5], `Appleseed` [36], `JQ` [15], `JSON-Simple` [26], `Minimal-JSON` [42], `Graphviz` [38], and finally `Gephi` [18]. Each input format was evaluated with three subjects, except for *DOT* which was evaluated with two programs. All our subject programs are open source `C`, `C++` or `Java` programs. On average, these programs have 478 KLOC and a maturity of over 14 years. Table 1 highlights the properties of our subject programs.

Table 1: Subject Programs

Subject Program	Input Format	Prog. Lang.	Size (in KLOC)	Maturity (1st Commit)
`Blender`	`OBJ`	`C/C++`	1800	Jan. 1994
`Assimp`	`OBJ`	`C++`	88.9	July 2002
`Appleseed`	`OBJ`	`C++`	600.1	May 2009
`JQ`	`JSON`	`C`	20.2	July 2012
`JSONSimple`	`JSON`	`Java`	2.6	Nov. 2008
`Minimal-JSON`	`JSON`	`Java`	6.4	Feb. 2013
`Graphviz`	`DOT`	`C`	1140	Sep. 1991
`Gephi`	`DOT`	`Java`	166.1	July 2008

Table 2: Input Grammar Details

Grammar	Size (LOC)	#ParserRules	#LexerRules
`JSON`	79	5	9
`Wave. OBJ`	271	13	42
`DOT`	181	14	15

Grammars. We have collected the grammars for our subjects from the *ANTLR Grammar repository* [19]. We chose complex and large grammars for data-rich input formats used in two popular domains, namely graphics domain (i.e. Wave. OBJ and DOT) and data exchange domain (i.e. JSON). To ensure the grammars were sound, we tested them with 50 valid crawled files for each input format. We modified the `Wavefront OBJ` grammar since its ANTLR grammar was only a subset of the official Wavefront OBJ specificiation [35]. The `JSON` and `DOT` were used unmodified since they matched the official specificiations [11, 27]. On average, the grammars are written in 177 LOC, with 11 parser rules and 22 lexer rules (*cf. Table 2*).

Mining and Filtering Input Files. Table 3 highlights the details of the input files in our corpus. We crawled for a specific file format using the file extension (e.g. "*.json*" for the JSON input format). In total, we collected a corpus of 9544 input files (*cf. #Crawled Files in Table 3*) using the *Github API* for crawling [24]. Then, we deleted all files that are empty or duplicated, as well as the input files that have a different input format despite having the intended file name suffix (e.g. a Wavefront OBJ file has the same suffix ".obj" as a binary OBJ file that was created by a compiler). This resulted in 7835 unique input files (*cf. #Unique files*). We also separated files that contain unsupported grammar extensions. In particular, for JSON and DOT, we removed 166 input files (*cf. #Grammar Files*) that only contain literals like a number or a string (e.g. which are invalid JSON [11]) and JSON files that contain multiple JSON files appended to each other, as written by some programs.

To determine actual invalid input files (*cf. #Invalid*), we *filter* out the valid input files from the set of unique files by checking that (1) the file does not lead to a lexing/parsing error when parsed by *ANTLR* and (2) the file was successfully opened by all subject programs (of the input format) without crashing (using the test oracle in Section 2.1). In total, 7702 input files (*cf. #Valid Files*) passed

Table 3: Mined Input Files

Input Format	#Crawled Files	#Unique Files	#Valid Files	#Invalid Files	*Cause of Invalidity (#files rejected by)* Grammar	≥ 1 subject	All subjects	Mean Valid Size (KiB)	Mean Invalid Size (KiB)
JSON	8654	7006	6948	222	164	58	52	12.84	0.78
Wave. OBJ	509	480	455	25	0	25	0	401.57	64.15
DOT	381	349	303	48	2	46	4	4.74	2.88

the check of the filtering process and the remaining 295 input files represent our set of real-world invalid files (*cf. #Invalid Files*).

Test Oracle. In our setup, the *test oracle* for *ddmax* is a *crashing oracle*. An input is treated as *invalid* if it crashes the subject program, or the result of the subject is empty, or the subject takes more than 10 seconds to process the input[1]. Opening a *valid* file, however, produces a non-empty output after 10 seconds and does not crash the subject program. To automate tests, we ensure that all subject programs have a full command-line interface (CLI) support or a `Java/Python` API. The test oracle was implemented in 890 LOC of `Java` and 412 LOC of `Python` code.

2.2 Evaluation

RQ1: How prevalent are invalid inputs in practice? Invalid input files are common. About *four percent* of all inputs in our corpus (295 files) were invalid (*cf. Table 3*); they were either rejected by subject program(s) or the input grammar. Specifically, about two percent of the input files (129 files) in our sample were rejected by at least one subject program; however, less than 1% (56 files) were rejected by *all* subject programs in our evaluation setup.

A common cause of invalidity is wrong syntax, missing or non-conforming elements. Many input files were invalid because of single character errors, such as a deleted character, a missing character or an extraneous character. For instance, some JSON inputs were invalid due to deletions of characters such as quotes, parentheses and braces. These errors are difficult to find because they are often hidden in large documents. For example, our set of crawled OBJ files contained many files of about 300KiB 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 and find this single corrupted character. Other sources of invalidity include the addition of elements that do not conform with the input specification. Some JSON files contained comments that begin with the "$" character. Comments are not permitted in JSON, however, this was common practice in some JSON files and a few parsers support comments (e.g. Google Gson).

In our sample of GitHub files, four percent could not be processed either by the input grammar or at least one subject program.

3 LEXICAL REPAIR

Now that we have established that there are actually files that cannot be properly parsed or opened, let us introduce the *ddmax* algorithm for recovering and repairing invalid input. *ddmax* works on a character-by-character basis; we thus call it *lexical ddmax*.

[1]This execution time of 10 seconds was determined as a maximum opening time to successfully process all valid input files in our evaluation corpus.

3.1 Delta Debugging

Our *ddmax* technique can be seen as a variation on *the minimizing delta debugging algorithm,* a technique for automatically reducing failure-inducing inputs by means of systematic tests. The *reduction problem* is modeled as follows: *Configurations* consisting of individual (input) elements which may or may not be present. There are two configurations: a *passing configuration* $c_{✔}$ and a *failing configuration* $c_{✘}$. The passing configuration $c_{✔}$ typically stands for an empty or trivial input ($c_{✔} = \emptyset$), and the failing configuration $c_{✘} \supset c_{✔}$ stands for the failure-inducing input in question. In our example from Section 1, the failing configuration would be

$$c_{✘} = \texttt{\{ "item": "Apple", "price": **3.45 \}} \quad (1)$$

Zeller et al. [45] define the *ddmin* algorithm as follows. *ddmin* produces one set $c'_{✘}$ with $c_{✔} \subset c'_{✘} \subseteq c_{✘}$, where $c'_{✘}$ has a *minimal size overall*. It works by testing sets c' that lie *between* $c_{✔}$ and $c_{✘}$ (i.e., $c_{✔} \subseteq c' \subseteq c_{✘}$). A test involves running the original program on the newly synthesized input c'. The outcome $test(c')$ of the test—either ✔ (passing), ✘ (failing), or **?** (unresolved)—determines algorithm progress: Whenever a subset $c' \subseteq c_{✘}$ fails ($test(c') = ✘$), *ddmin* further narrows down the difference between c' and $c_{✔}$. In our example from Section 1, Figure 2 shows a typical *ddmin* output $c'_{✘}$: The one character in the input suffices to cause the (syntax) error.

When choosing a new candidate c', *ddmin* initially splits the sets to be tested in half; as long as tests always pass or fail, this is as efficient as a binary search. If tests are unresolved (say, because the input is invalid), *ddmin* resorts to cutting quarters, eighths, sixteenths of the input (*ddmin*). Eventually, *ddmin* tests each remaining element (character) for its relevance in producing the failure.

3.2 The *ddmax* Algorithm

Our definition of *ddmax* is shown in Figure 5. *ddmax* uses the same setting as *ddmin*; however, rather than *minimizing* the failure-inducing input $c_{✘}$, it starts with a passing input $c'_{✔} = c_{✔}$; like *ddmin*, it assumes for simplicity that $c_{✔} = \emptyset$ holds. It then *maximizes* $c'_{✔}$, systematically minimizing the difference between $c'_{✔}$ and $c_{✘}$ using the same techniques as *ddmin* (first progressing with large differences, then smaller and smaller differences), until every remaining difference would cause $c'_{✔}$ to fail. This makes *ddmax* act in exact symmetry to *ddmin*, and complements the original definitions of *dd* and *ddmin* [45].

3.3 A *ddmax* Example

How does *ddmax* work? Let us illustrate it on the example from Section 1. We have $c_{✘}$ defined as in Equation (1), above, and evaluate $ddmax(c_{✘})$ to obtain $c'_{✔}$, the maximal subset of $c_{✘}$ that passes the test (i.e., that can be still be processed by our JSON application at hand). We now invoke $ddmax(c_{✘})$ and get $ddmax_2(\emptyset, 2)$—that is,

Maximizing Delta Debugging Algorithm

Let *test* and $c_{✘}$ be given such that $test(\emptyset) = ✔ \wedge test(c_{✘}) = ✘$ hold.
The goal is to find $c'_{✔} = ddmax(c_{✘})$ such that $c'_{✔} \subset c_{✘}$, $test(c'_{✔}) = ✔$, and $\Delta = c_{✘} - c'_{✔}$ is 1-minimal.
The *maximizing Delta Debugging algorithm* $ddmax(c)$ is

$$ddmax(c_{✘}) = ddmax_2(\emptyset, 2) \quad \text{where}$$

$$ddmax_2(c'_{✔}, n) = \begin{cases} ddmax_2(c_{✘} - \Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c_{✘} - \Delta_i) = ✔ \text{ (“increase to complement”)} \\ ddmax_2\big(c'_{✔} \cup \Delta_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{✔} \cup \Delta_i) = ✔ \text{ (“increase to subset”)} \\ ddmax_2\big(c'_{✔}, \min(|c_{✘}|, 2n)\big) & \text{else if } n < |c_{✘} - c'_{✔}| \text{ (“increase granularity”)} \\ c'_{✔} & \text{otherwise (“done”).} \end{cases}$$

where $\Delta = c_{✘} - c'_{✔} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c_{✘} - c'_{✔}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmax_2$ is $test(c'_{✔}) = ✔ \wedge n \leq |\Delta|$.

Figure 5: Maximizing Lexical Delta Debugging algorithm

$c'_{✔} = \emptyset$ and $n = 2$. The set $c'_{✔}$ will continually hold more and more characters, and n will hold the current granularity.

$ddmax_2$ determines $\Delta = c_{✘} - c'_{✔} = c_{✘} - \emptyset = c_{✘}$, and splits it into two parts $\Delta_1 \cup \Delta_2 = \Delta$:

$$\Delta_1 = \texttt{ "price": **3.45 }}$$
$$\Delta_2 = \texttt{\{ "item": "Apple",}$$

As part of “increase to complement”, $ddmax_2$ first tests $c_{✘} - \Delta_1$ (which is Δ_2) and then $c_{✘} = \Delta_2$ (which is Δ_1). Neither of both is a valid JSON input, hence the tests do not pass. In “increase to subset”, the sets to be tested are $c'_{✔} \cup \Delta_1 = (\emptyset \cup \Delta_1) = \Delta_1$ and $c'_{✔} \cup \Delta_2 = (\emptyset \cup \Delta_2) = \Delta_2$; we already know that these tests do not pass. Hence, we “increase granularity” and double n to $n = 4$.

With $n = 4$, we now split Δ into four parts $\Delta_1 \cup \cdots \cup \Delta_4 = \Delta$:

$$\Delta_1 = \texttt{\{ "item":} \qquad \Delta_2 = \texttt{ "Apple",}$$
$$\Delta_3 = \texttt{ "price":} \qquad \Delta_4 = \texttt{ **3.45 }\}$$

In “increase to complement”, the tests run on the failing set $c_{✘}$ *without* the individual Δ_i—that is:

$$c_{✘} - \Delta_1 = \texttt{ "Apple", "price": **3.45 }}$$
$$c_{✘} - \Delta_2 = \texttt{\{ "item": "price": **3.45 }}$$
$$c_{✘} - \Delta_3 = \texttt{\{ "item": "Apple", **3.45 }}$$
$$c_{✘} - \Delta_4 = \texttt{\{ "item": "Apple", "price":}$$

None of these inputs is syntactically valid JSON, and no test passes; so *ddmax* further increases granularity to $n = 8$. In this round, again none of the Δ_i pass; but one of the complements does:

$$c_{✘} - \Delta_6 = \texttt{\{ "item": "Apple", "price":45 }}$$

$$\text{with} \quad \Delta_6 = \texttt{ **3.}$$

The set $c_{✘} - \Delta_6$ is indeed a syntactically valid JSON input, and $test(c_{✘} - \Delta_6)$ passes (“increase to complement”). At this point, we have recovered $\frac{31}{36} = 86\%$ of the input data already.

Can we add more characters? Following the *ddmax* definition, we reinvoke $ddmax_2$ with $c'_{✔} = c_{✘} - \Delta_6$. Now, the remaining difference between $c'_{✔}$ and $c_{✘}$ is Δ_6 as above. We restart with $n = 2$ and decompose the remaining $\Delta = c_{✘} - c'_{✔} = \Delta_6$ into Δ_{6_1} and Δ_{6_2}:

$$\Delta_{6_1} = \texttt{ **} \qquad \Delta_{6_2} = \texttt{3.}$$

Now, $c_{✘} - \Delta_{6_1}$ passes, yielding the syntactically correct input:

$$c_{✘} - \Delta_{6_1} = \texttt{\{ "item": "Apple", "price":3.45 }}$$

A further iteration will also recover the space character before the number, eventually yielding the repaired input in Figure 3 and the remaining difference Δ in Figure 4.

The example demonstrates two important properties of *ddmax*:

- *ddmax* is *thorough*. Its result $c'_{✔}$ is *1-maximal*—that is, adding any further character from $c_{✘}$ will no longer pass. Formally, this means that $\forall \delta_i \in c_{✘} - c'_{✔} \cdot test\big(c'_{✔} \cup \{\delta_i\}\big) \neq ✔$ holds.[2]
- *ddmax* can be *slow*. The *complexity* of *ddmax* is the same as *ddmin*—in the worst case, the number of tests carried out by $ddmax(c_{✘})$ is $|c_{✘}|^2 + 3|c_{✘}|$; and in the best case—if there is only one failure-inducing change $\Delta_i \in c_{✘}$, and all cases that do *not* include Δ_i pass, then the number of tests t is limited by $t \leq 2\log_2(|c_{✘}|)$.
  In practice, as with *ddmin*, things will be somewhere between the two extremes; but keep in mind that at maximum granularity, *ddmax* runs at least $|c_{✘} - c'_{✔}|$ tests—that is, one test for every character that possibly still could be restored.

With these properties, what we get with *ddmax* is an algorithm that guarantees a maximum of data recovery, albeit at the price of possibly running a large number of tests.

3.4 Evaluation Setup

Workflow. Figure 6 shows the workflow of our evaluation. First, we collect real-world invalid input files from the set of *crawled files*, according to Section 2. Those files are then filtered into a set of valid files and a set of invalid files (Step. 1) and duplicates and files with a wrong format are deleted. Secondly, we select and mutate 50 valid *crawled files* to produce an additional set of corrupted input files (Step. 2). Then, we feed a invalid file to each subject program, and the *ANTLR* parser framework (Step. 3). *ANTLR* executes its default *error recovery strategy* while generating a parse tree for the input. Next, we feed the invalid file to *lexical ddmax* (Step. 4). *Lexical*

[2]Both maximality and complexity properties are proven in a way analogous to the properties of *ddmin* in [45].

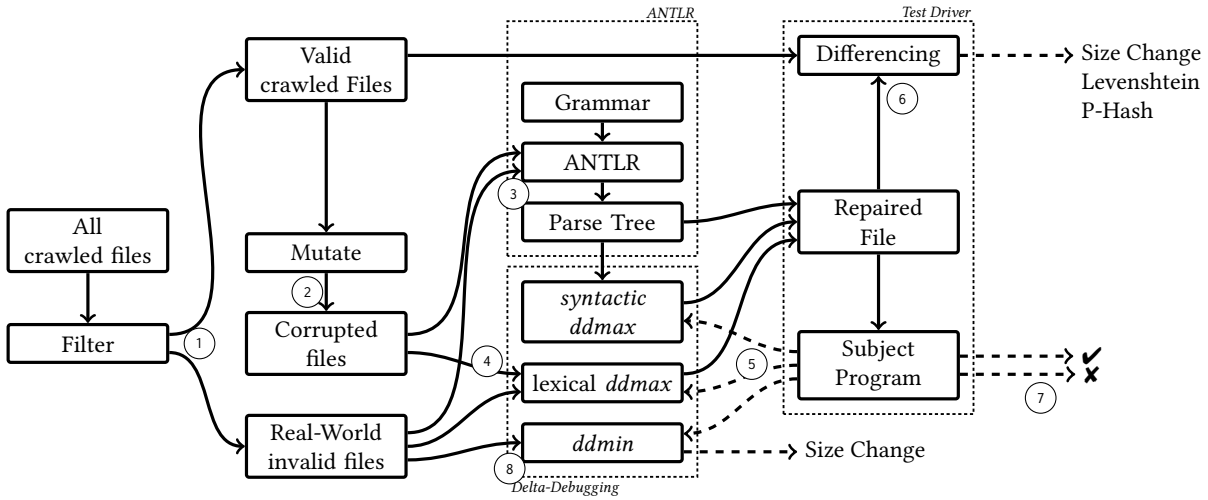

Figure 6: Workflow of the *ddmax* evaluation

ddmax tests the input under repair repeatedly using the feedback from the subject program (Step. 5). Then, we feed the original crawled files and the resulting repaired file from each technique to the *differencing* framework (Step. 6), which computes the *change in file size*, *Levenshtein distance* and *perceptive hash value* for both files. We save the feedback from our subject program (Step. 7). Finally, to ensure the quality of our approach, we also execute *ddmin* on the real-world invalid inputs (Step. 8) and report the content and size of the result.

Lexical ddmax was implemented in 595 LOC of `Java` code. *ANTLR* also implements an inbuilt error recovery strategy which is designed to recover from lexing or parsing errors (e.g. missing/wrong tokens or incomplete parse trees) [25].

Mutations. In addition to the real-world invalid inputs (*cf. Section 2*), we also simulate real-world data corruption by applying *byte-level* mutations on valid input files. These mutations were chosen because they are similar to the corruptions observed in real-world invalid files (*see Section 2 and Section 5*). We perform the following mutations at a random position in each valid input file: *byte insertion*, *byte deletion* and *byte flip*. To simulate *single data corruption*, we randomly choose one of these mutations and apply it once on the valid input file. For *multiple data corruptions*, we perform up to 16 random mutations on each input file. A mutation is only successful (for an input format), if *at least* one of the subject programs (that passes before) fails after the mutation. These criteria is similar to how we collected invalid input files in the wild.

Metrics and Measures. In order to determine the quality of *ddmax* repair, we use the following metrics and tools:

(1) **File Size:** We measure the *file size* of the inputs recovered by *ddmax* and the *difference in file size* between the original *valid input* and the *repaired file*. We use these measurements to account for the amount of data recovered by *ddmax* as well as the amount of data loss incurred.
(2) **Levenshtein Distance:** Additionally, we measure data loss using the Levenshtein distance metric [30], measuring the *edit distance* between *valid input* and *repaired file*.
(3) **Perceptive Image Difference:** In order to measure the *(semantic) information loss* incurred by *ddmax*, we calculate the hash value of our 3D images, i.e. `Wavefront OBJ` format. We compute the image distance of our 3D image files by rendering both the repaired image and the original valid image into several 2D images from three different camera angles and three scales, then measuring the 2D image distance of all nine images. We compare these images using the `Python ImageHash library` [7] in order to obtain a good approximation of the real image difference between those two 3D models as a *perceptive image difference* between both images. In our setup, we use two rendering engines (Blender [17] and Appleseed [36]) to render the images.

Research Protocol. For each input format, we collect real-world invalid input files. Secondly, we perform single and multiple mutations on 50 valid input documents. Then, we execute all files on the different subject programs, in order to determine the proportion of input files which fail for each subject program. We proceed to run *lexical ddmax* on each invalid or mutated input file. In particular, we are interested in determining the following: (1.) ***Baseline:*** the proportion of invalid input files which are accepted by a subject program as *valid inputs* (i.e. non-failure-inducing inputs processed by the program without leading to a crash), in order to measure the *effectiveness* of the built-in *error recovery* feature of the program; and (2.) ***ANTLR:*** the proportion of invalid inputs which are repaired by ANTLR inbuilt *error recovery strategy*; (3) ***Lexical:*** the proportion of invalid inputs which are repaired by *lexical DDMax*.

All experiments were conducted on a Lenovo Thinkpad with four physical cores and 8GB of RAM, specifically an Intel(R) Core i7 2720qm @ 2.20GHz, 8 virtual cores, running 64-bit Arch Linux. All our prototypes are single-threaded.

3.5 Evaluation

RQ2: How effective is *lexical ddmax* in repairing invalid input documents within a time budget of one minute per file? *Lexical ddmax* repaired about two-thirds (66%) of all invalid inputs (*cf. Table 4*). It also outperformed both the in-built repair strategy

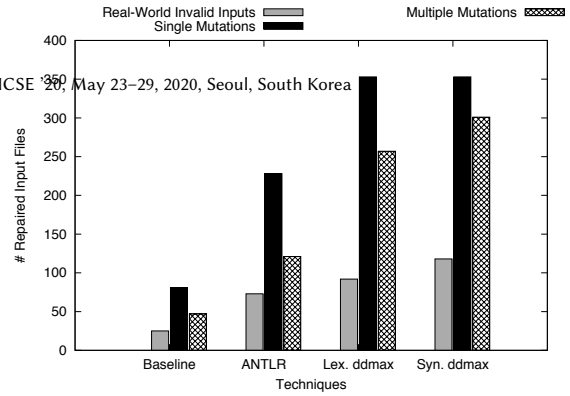

Figure 7: Number of Repaired Files for Each Technique

Table 4: *ddmax* Effectiveness on All Invalid Inputs

Invalid. Type	Format (#subjects)	#Possible Repairs	*# repaired input files* Base.	ANTLR	Lex.	Syn.
Real World	JSON (3)	167	0	40	38	62
	OBJ (3)	33	1	8	24	25
	DOT (2)	64	24	25	30	31
Single Mut.	JSON (3)	150	4	80	115	127
	OBJ (3)	150	34	82	146	144
	DOT (2)	100	43	66	92	82
Multiple Mut.	JSON (3)	150	4	45	79	112
	OBJ (3)	150	3	29	127	126
	DOT (2)	100	40	47	51	63
Total (3)		1064	153	422	702	772

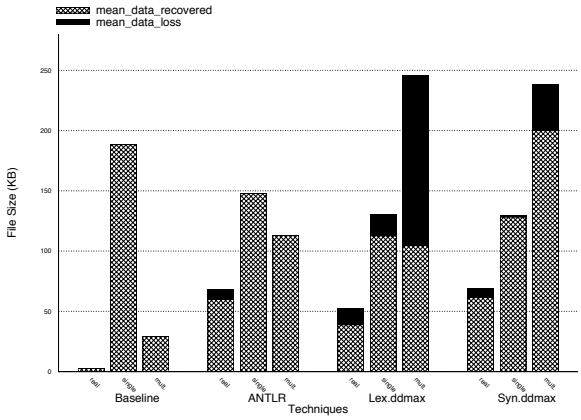

Figure 8: Data Recovered and Data Loss for all Inputs

of the subject programs (*Baseline*) and the ANTLR error recovery strategy (*ANTLR*), both of which repaired 14% and 40% of all invalid input files respectively. Specifically, *lexical ddmax* repaired over four times as many invalid input files as the *Baseline* and 66% more invalid input files than ANTLR (*cf. Figure 7*). The performance of *lexical ddmax* was significantly better for both all mutations.

> *Lexical ddmax repaired about two-thirds of all invalid inputs and significantly outperforms both the basline and ANTLR.*

RQ3: How much data is *recovered* by *lexical ddmax* and how much is the *data loss* incurred by *lexical ddmax*? In terms of recovery rate, *lexical ddmax* performs slightly worse than the other techniques, with a recovery rate of 75% on real-world invalid inputs, 86% on single data corruption, and about 43% on multiple

Table 5: *ddmax* Efficiency on All Invalid Inputs for each technique (A) Baseline, (B) ANTLR, (C) Lexical *ddmax*, (D) Syntactic *ddmax*.

Invalid. Type	Inp. Form	*Runtime (sec.)* A	B	C	D	*#Runs* C	D
Real World	JSON	2	2	1227	153	341525	6029
	OBJ	44	47	2065	1279	6253	3164
	DOT	48	166	3828	3018	2783	1162
Single Mut.	JSON	4	4	1584	1065	45651	129659
	OBJ	491	672	6151	4083	3809	1352
	DOT	58	60	1239	1244	6077	4565
Multiple Mut.	JSON	10	10	5903	2153	1194577	448801
	OBJ	624	728	9938	8132	8577	5043
	DOT	60	60	3365	2241	34876	11956
Mean		153	200	3981	2624	72296	70049

data corruption (*see Figure 8*). For both types of data invalidity, the *baseline* and *ANTLR* maintain an almost perfect data recovery rate (approximately 100%).

> Lexical *ddmax recovered most (75% and 65%) of the data in real-world invalid inputs and mutated input files respectively.*

In theory, *lexical ddmax* is guaranteed to ensure minimal data loss for all repairs. However, due to large file sizes and timeout constraints in our experimental setup, *lexical ddmax* often halts before the maximal valid data is recovered. In our experiment, *lexical ddmax* had timed out for 163 input files during repair. In order to inspect the data recovery rate of each approach in a more balanced setting, we examined the set of input files that were repaired by both *ANTLR* and *lexical ddmax*, before *lexical ddmax* timed out. In total, 109 repairs were accomplished by both lexical *ddmax* and *ANTLR*, before a time out. The data loss of *lexical ddmax* is minimal and comparable to *ANTLR*, this holds for both single and multiple data corruption for the intersecting set before timeout. In fact, on average, *lexical ddmax* recovered 1.724 KiB of data, and *ANTLR* recovered 1.548 KiB.

> *Overall, lexical ddmax incurs minimal data loss during repair: It recovers similar amount of data from invalid input files, in comparison to ANTLR.*

RQ4: How *efficient* is *lexical ddmax* in repairing invalid input documents ? On average, it took less than two minutes (1.3 minutes) to repair a file (*cf. Figure 13*). In comparison, both the *Baseline* and *ANTLR* had an execution time of 3 and 4 seconds per input file respectively. This indicates that *lexical ddmax* is more time-consuming than both the *Baseline* and *ANTLR*. This is expected since *ddmax* requires multiple executions of the subject programs (as indicated in *lexical #Runs in Table 5*).

> *Lexical ddmax is relatively fast in repairing an invalid input file: it takes less than two minutes (78 seconds) on average.*

4 SYNTACTIC REPAIR

We have seen that *ddmax* is general, but also *slow:* If one wants to recover a maximum of data, it runs a single test for every candidate

character that can be recovered. Is it possible to speed things up, possibly by leveraging information on the input format? To this end, we introduce the *syntactic ddmax* algorithm, which improves the performance of *ddmax* using the knowledge of the input grammar.

The key insight is to execute *ddmax* on the *parse tree* of the input, instead of the input characters. Here, we analyze the input at the syntactical level, rather than the lexical level. This improves the runtime and general performance of the *ddmax* algorithm. The main benefit of the approach is that it enables *ddmax* to reason at a more coarse-grained level by testing on the input structure. *Lexical ddmax* may take thousands of test runs, depending on the size of the input, in fact its number of runs is bound to the number of characters in the input. However, *syntactic ddmax* is bound to the number of *terminal nodes* in the parse tree, which is typically smaller than the number of characters in the input. Thus, *syntactic ddmax* can easily exclude corrupted parse tree nodes or subtrees during test runs. Additionally, the knowledge of the input structure ensures that the resulting recovered inputs are syntactically valid. This helps in the case of syntax errors, large corrupted input region(s) and multiple data corruptions on the input (structure).

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

Figure 9: Failing JSON input with missing colon

Specifically, the syntactic *ddmax* algorithm takes as input a *parse tree* for the corrupted input file (*cf. Figure 11*) and obtains a pre-order list of terminals in the parse tree. For instance, consider the corrupted JSON input in Figure 9. Repairing this input using the *lexical ddmax* algorithm results in the JSON input in Figure 10, which would take over 100 test runs. Even for this small example, *syntactic ddmax* enhances the performance of *ddmax* with the input grammar, reducing the number of test runs of *ddmax* to nine and improving performance by ten fold.

```
{ "item": "Apple" }
```

Figure 10: Repaired JSON input by *ddmax*

To repair the input (*cf. Figure 9*), syntactic *ddmax* first parses the input into a parse tree[3], shown in Figure 11. Next, we run the *ddmax* algorithm on the parse tree, removing terminal nodes (instead of single characters) in each iteration of *ddmax*[4]. We define $c_{✘}$ as our failing configuration, which contains the terminal nodes of the parse tree from Figure 11.

Let us run the *ddmax* algorithm on our example terminal nodes. We invoke $ddmax(c_{✘})$ which results in $ddmax_2(\emptyset, 2)$, so inside $ddmax_2$, we have $c'_{✔} = \emptyset$ and $n = 2$. At first, our Δ is split into

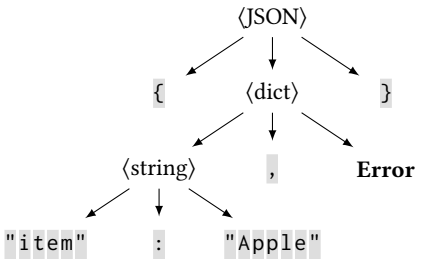

Figure 11: Parse tree of Figure 9

two parts[5]:

$$\Delta_1 = \texttt{\{ "item" : "Apple"}$$
$$\Delta_2 = \texttt{, Error \}}$$

Running $test(c_{✘} - \Delta_1)$ and $test(c_{✘} - \Delta_2)$ both fail (= ✘). We are at the first run, so with $c'_{✘} = \emptyset$, $c'_{✔} \cup \Delta_1 = c_{✘} - \Delta_2$ and $c'_{✔} \cup \Delta_2 = c_{✘} - \Delta_1$ which also both fail in the "increase to subset" step. Next, we set $n = 4$ and restart $ddmax_2(c'_{✔}, n)$.

With $n = 4$, in the "increase to complement" and "increase to subset" steps, we get

$$\Delta_1 = \texttt{\{ "item"} \qquad \Delta_2 = \texttt{: "Apple"}$$
$$\Delta_3 = \texttt{, Error} \qquad \Delta_4 = \texttt{\}}$$

In the "increase to complement" step, we find that $test(c_{✘} - \Delta_3) =$ ✔, so we repeat our algorithm with $c'_{✔} = c_{✘} - \Delta_3$ and $n = 2$, getting

$$\Delta_1 = \texttt{,} \qquad \Delta_2 = \texttt{Error}$$

Since neither $test(c_{✘} - \Delta_i)$ nor $test(c'_{✔} \cup \Delta_i)$ passes for any i and $n = |c_{✘} - c'_{✔}| = 2$, we are done and end up with the remaining input seen in Figure 10. For this example, the *syntactic ddmax* run needed only 9 test runs of the subject program to repair the faulty input.

Let us now take a look at the complexity of our algorithm. As mentioned in Section 3, *ddmax* has a worst-case complexity of $t = |c_{✘}|^2 + 3|c_{✘}|$ test runs and a best-case complexity of $t \leq 2\log_2(|c_{✘}|)$. Intuitively, the complexity of *syntactic ddmax* is similar to the complexity of *ddmax*, except that it is bounded by the number of terminal nodes instead of the number of characters. In the worst case, an input's parse tree has as many terminal nodes as characters. However, real-world input formats have keywords, data types and character classes to aggregrate group of characters into terminals (e.g. string and integers). This reduces the number of terminal nodes and the required number of test runs for *syntactic ddmax*. It therefore speeds up *ddmax* by decreasing the number of elements to maximize with *ddmax*. Consider the example in Figure 9, there are 33 single characters to search with *lexical ddmax*, which are parsed into 7 terminal nodes for *syntactic ddmax*. In general, we can assume that with an average terminal node length of n characters,

[3] *ANTLR* is capable of generating a parse tree for corrupted input files, it summarizes syntactically wrong symbols or trees into error nodes (similar to Figure 11).

[4] Removing only the error node in the parse tree does not necessarily result in a non-failure-inducing input.

[5] Note that checking for only syntactically valid subsets of the programs (e.g. using the grammar only) is not sufficient to repair the input. We leverage the application, since the semantics and intended use of the input file are encoded in the logic of the applcation.

we have a worst-case complexity of $\left(\frac{|c_{\mathbf{x}}|}{n}\right)^2 + 3\frac{|c_{\mathbf{x}}|}{n}$ test runs and a best-case complexity of $t \leq 2\log_2(\frac{|c_{\mathbf{x}}|}{n})$ test runs.

4.1 Evaluation Setup

Implementation. *Syntactic ddmax* was implemented in 1084 LOC of `Java` code, this implementation is built on top of the *ANTLR* 4.5 parser generator framework [37] for each input grammar. Overall, the implementation of *syntactic ddmax* differs from that of *lexical ddmax* in Section 3.4, because of its use of the input grammar and parse tree. Specifically, *Syntactic ddmax* uses the ANTLR parse tree (from Step. 3 in Figure 6) to repair invalid inputs. In our evaluation, we feed the invalid real-world files into our *syntactic ddmax*, we proceed to run *syntactic ddmax* on each invalid input file and evaluate the change in file size (i.e. the data loss on byte-level). *Syntactic ddmax* tests the input under repair repeatedly using the feedback from the subject program (Step. 5). In addition to the research protocol in Section 3.4, we feed all invalid input files to *syntactic ddmax* and measure the proportion of invalid files which are repaired by our *syntactic DDMax* using the input grammar, this measure is termed ***Syntactic***.

4.2 Evaluation

RQ5: How effective is *syntactic ddmax* in repairing invalid input documents within a time budget of one minute per file? *Syntactic ddmax* repaired about three-quarters (73%) of all invalid inputs in our evaluation (*cf. Table 4*). Overall, it is about 10% more effective than *lexical ddmax* (*cf. Figure 7*). 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* (*cf. Table 4*). This confirms our hypothesis (*in RQ2*) that *ddmax* can benefit from the knowledge of the input grammar.

Syntactic ddmax repaired about three-quarters of all invalid inputs and it is more effective than lexical ddmax, for all invalid inputs.

RQ6: How much data is *recovered* by *syntactic ddmax* and how much is the *data loss* incurred by *syntactic ddmax* ? On average, *syntactic ddmax* (89%) has a higher data recovery rate in comparison to *lexical ddmax* (58%) for all invalid inputs. For single data corruption, the data recovery rate of *syntactic ddmax* is similar to that of *ANTLR* and the *baseline*, when using mean file size as a metric. On multiple data corruption, *syntactic ddmax* recovered about 84% of the data in the input files (*cf. Figure 8*). For all invalid inputs, the *baseline* and *ANTLR* maintain an almost perfect data recovery rate (approximately 100%). Evidently, the data loss incurred by both *ANTLR* and the *baseline* is negligible.

Syntactic *ddmax has a high data recovery rate, recovering most (89%) of the data in invalid input files.*

The data loss incurred by *ddmax* is very low, in terms of the edit distance between the recovered file and the valid file. Across all mutations, it is less than 50% worse off than *ANTLR*, as captured by the *Levenshtein distance* (*cf. Figure 12*). In particular, the mean edit distance of the repaired file and the originally valid input file is less than four for the *baseline* and about 24 for *ANTLR*. As expected, the Levenshtein distance is lower (21–28) for single data corruptions for *lexical* and *syntactical ddmax* respectively, and higher for multiple corruptions (76–77). On inspection, we found that the high loss of *ddmax* is due to early timeouts for large input files, indeed, *ddmax* finds a valid subset, but times out before the maximal subset is reached. For Wavefront OBJ files, the perceptive image difference shows us similar scaling result as the Levenshtein distance. While it shows small results for Baseline and ANTLR (0.1 and 29.7, respectively), the results for *lexical* and *syntactical ddmax* are higher (76.4 and 51.9), thus the difference between the unmodified image and the repaired image is larger.

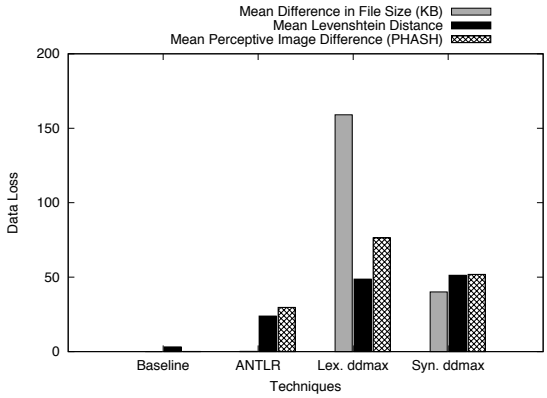

Figure 12: Data Loss Incurred for All Mutations

We conduct our evaluation of minimal data loss similarly to the setting in RQ3 (*cf. Section 3.4*). As expected, *syntactic ddmax* recovered slightly less data, exactly 1.720 KiB on average. This is because *syntactic ddmax* removes terminal nodes, a terminal node may contain more characters than the number of mutated characters in the node. In summary, with a high enough timeout *lexical ddmax* is guaranteed to achieve minimal data loss, this guarantee does not hold for *syntactic ddmax*, since it operates at the parse tree level rather than the byte level.

Overall, syntactic ddmax incurs comparatively similar data loss during repair as lexical ddmax.

RQ7: How *efficient* is *syntactic ddmax* in repairing invalid input documents? *Syntactic ddmax* improves over the runtime performance of *lexical ddmax* (*cf. RQ4 Section 3.4*). It improves over *lexical ddmax* by 34%, its execution time is about two-third of the running time of *lexical ddmax*. Specifically, *syntactic ddmax* is quicker, it took approximately one minute to repair a single file, but requires a grammar and a parse tree[6].

Syntactic ddmax is faster in repairing an invalid input file: it takes less than one minute to repair a file on average.

[6]Depending on the grammar and on the input file size, generating a parse tree should take less than a second.

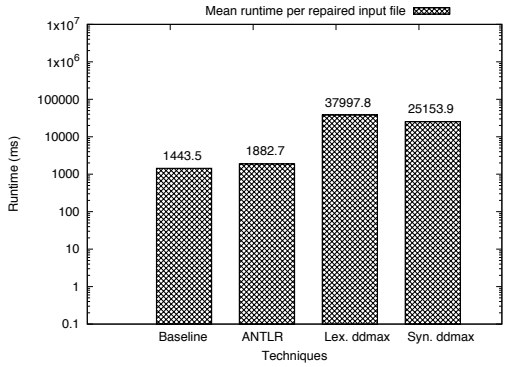

Figure 13: Mean Runtime per Input File for Each Technique

Table 6: Diagnostic Quality on Real-World Invalid Inputs for Ⓐ *ddmin* and Ⓑ *ddmax* diagnoses, and Ⓒ *ddmax* repair.

Format	Diagnosis (B)		Repair (B)	*Intersection (%)*	
(#inputs)	Ⓐ	Ⓑ	Ⓒ	Ⓐ ∩ Ⓑ	Ⓐ ∩ Ⓒ
JSON (21)	2.909	19.095	103.476	13.88	23.18
OBJ (18)	2.722	1.000	189.000	18.03	11.46
DOT (27)	376.654	1.115	675.346	5.76	54.64
Mean	155.754	6.804	360.747	11.69	32.85

5 DIAGNOSTIC QUALITY

Even though *ddmax* is primarily meant for repairing data, its maximized input can also be useful for diagnostics and debugging. In particular, *ddmax* diagnosis is the difference Δ between the failing and maximal passing input. This is a minimal failure cause, which is necessary to debug the input. Most notably, the Δ from *ddmax* includes *all* input characters that are failure-inducing, whereas *ddmin* include only a minimal subset.

5.1 Evaluation Setup

To comparatively evaluate the diagnostic utility of *ddmax*, we copare *ddmax* diagnoses to the established state of the art input diagnosis approach *ddmin*. We implemented a *ddmin* algorithm following the delta debugging algorithm in [45] in 450 LOC of Java code. Our *ddmin* implementation uses both the subject program and ANTLR as oracles to minimize an invalid input, in order to ensure that *ddmin* diagnosis is syntactically valid. This ensures that *ddmin* does not report a valid subset that may trigger a failure due to syntactic invalidity (e.g. in *cf. Figure 2 in Section 1*), since ANTLR can parse the *ddmin* diagnosis, but the subject program crashes. Then, we feed the real-world invalid files into our *ddmin* implementation (as seen in Figure 6 Step. 8) and compare the diagnosis generated by *ddmin* to that of *ddmax*.

We are interested in evaluating the soundness and completeness of both *ddmin*, using *ddmax* diagnoses as the "ground truth". To be fair to both approaches, we consider the intersection of the diagnoses for both *ddmin* and *ddmax* that finished execution before a time-out, this a set of 66 input files in total (*cf. Table 6*).

5.2 Evaluation

RQ8: How *effective* is *ddmax* in diagnosing the root cause of invalid inputs, especially in comparison to *ddmin*? 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 (33%) of the maximal passing input, which is considered noise in the diagnosis (*cf.* Ⓐ ∩ Ⓒ *in Table 6*). Additionally, *ddmin* diagnosis only contains a small portion (12%) of the minimal failure cause required to diagnose the input invalidity (*cf.* Ⓐ ∩ Ⓑ *in Table 6*). This result shows that *ddmax* diagnosis is more effective for input debugging in comparison to the state of the art, *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.*

6 LIMITATIONS

Both *ddmax* variants are limited in the following ways:

Repair to subsets only. Both *ddmax* variants will return a strict lexical or syntactical *subset* of the original failure-inducing input. The assumption is that only data should be restored that already is there (rather than synthesizing new data, for instance). If the input format has several context-sensitive dependencies, such as checksums, hashes, encryption, or references, a strict lexical or syntactical subset might be small to the point of being useless.

Data repair, not information repair. Both *ddmax* variants are set to recover as much *data* as possible, but not necessarily *information.* Even though the repaired input may be lexically or syntactically close to the (presumed) original input, it can have very different semantics. Users therefore are advised to thoroughly *check the repaired input* for inconsistencies; the goal of this work is to *enable* users to load the input into their program such that they can engage in this task.

7 THREATS TO VALIDITY

Our evaluation is limited to the following threats to validity:

External validity refers to the generalizability of our approach and results. We have evaluated our approach on a small set of applications and input grammars. There is a threat that *ddmax* does not generalize to other applications and grammars. However, we have mitigated this threat by evaluating *ddmax* using mature subject programs with varying input sizes. Our subjects have 478 KLOC and 14 years maturity, on average, making us confident that our approach will work on a large variety of applications and invalid inputs.

Internal validity is threatened by incorrectness of our implementation, specifically whether we have correctly adapted *ddmin* to *ddmax* for input repair. We mitigate this threat by testing our implementation on smaller inputs and simpler grammars, in order to ensure our implementation works as expected.

Construct validity is threatened by our test oracle, and consequently the error-handling implementation of the subject.

For instance, an application which silently handles exceptions would not provide *ddmax* with useful feedback during test runs. We checked that the rendered input produced by the subject is non-empty, after a 10 second timeout, which was sufficient to identify failure-inducing inputs.

8 RELATED WORK

There is a large body of work concerning the interplay of program, inputs, and faults. We discuss the most important related works.

Document Recovery has the goal to fix broken input documents. Docovery [29] uses symbolic execution to manipulate corrupted input documents in a manner that forces the program to follow an alternative error-free path. In contrast to *ddmax*, this is a white-box approach that analyzes the program paths executed by the failure-inducing inputs. S-DAGS [40] is a semi-automatic technique that enforces formal (semantic) consistency constraints on inputs documents in a collaborative document editing scenario. Both of these approaches require program analysis.

Input Rectification aims at transforming invalid inputs into inputs that behave acceptably. Input *rectifiers* [31, 39] address this problem by learning a set of constraints from typical inputs, then transforming a malicious input into a benign input that satisfies the learned constraints. In contrast, *ddmax* does not learn constraints but rather employs the feedback from the program's execution (and a grammar) to determine an acceptable subset of the input. In comparison to security-critical rectification, its goal is maximal data recovery.

Input Debugging. Numerous researchers have examined the problem of simplifying failure-inducing inputs [10, 34, 41, 45]. In particular, [34] (HDD) and [41] are closely related to *ddmax*. Both approaches use the input structure to simplify inputs, albeit without an input grammar. Unlike *ddmax*, these approaches do not recover maximal valid data from the failure-inducing input, but rather minimize the input like *ddmin*.

Data Diversity [2] transforms a failure-inducing input into a valid input that generates an equivalent result, in order to improve *software reliability*. This is achieved by finding the regions of the program's input space that causes a fault, and re-expressing a failing input to avoid the faulty input regions. In contrast, *ddmax* does not require program analysis; it only needs a means to assess whether the program can process the input or not.

Data Structure Repair iteratively fixes corrupted data structures by enforcing they conform to consistency constraints [12–14, 23]. These constraints can be extracted, specified and enforced with predicates [16], model-based constraint systems [13], goal-directed reasoning [14], dynamic symbolic execution [23] or invariants [12]. On the one hand, the goal of data structure repair is to ensure a program executes safely and acceptably, despite data structure corruption. On the other hand, the goal of *ddmax* is to repair the input in order to avoid the corruption of the program's internal data structure.

Syntactic Error Recovery. Parsers and compiler implement numerous syntax error recovery schemes [6, 21]. Most approaches involve a plethora of operations including insertion, deletion and replacement of symbols [3, 4, 9], extending forward or backwards from a parser error [8, 33], or more general methods of recovery and diagnosis [1, 28]. Unlike *ddmax*, the aim of these schemes is to ensure the compiler does not halt while parsing; the input still would not be automatically fixed.

9 CONCLUSION AND FUTURE WORK

With *ddmax*, we have presented the first *generic* technique for automatically repairing failure-inducing inputs—that is, recovering a maximal subset of the input that can still be processed by the program at hand. *ddmax* is a variant of delta debugging that maximizes the passing input, both at a lexical and a syntactical level; it requires nothing more than the ability to automatically run the program with a given input. In our evaluation, we find that *ddmax* fully repairs 79% of invalid input files. Both variants of *ddmax* can be easily implemented and deployed in a large variety of contexts. as they do not require any kind of program analysis.

Our work opens the door for a number of exciting research opportunities. Our future work will focus on the following issues:

Synthesizing input structures. Going for a strict lexical or syntactical subset of the failure-inducing input is a conservative strategy; yet, there can be cases where *adding* a small amount of lexical or syntactical elements can help to recover even more information. We are investigating appropriate grammar-based production strategies as well as hybrid strategies that leverage both syntactical and lexical progression.

Learned grammars. Right now, our syntactical variant of *ddmax* requires an input grammar to start with. We are investigating whether such a grammar can also be *inferred* from the program at hand [20, 22], thus freeing users or developers from having to provide a grammar.

From input repair to code repair. A minimal difference Δ between a maximized passing and a full failure-inducing input also brings great opportunities for fault localization and repair. For instance, what is the code executed by the failure-inducing input, but not by the maximized passing input? What are the differences in variable values? Such differences in execution can help developers to further narrow down failure causes as well as synthesizing code repairs.

End-user debugging. Our input repair technique needs no specific knowledge on program code or programming, and could thus also be applied by end users. We are investigating appropriate strategies to communicate the results of our repair and information about conflicting document parts to end users, such that they can *fix the problem* without having to *fix the program.*

Our implementations of *ddmax* as well as all experimental data is available as a replication package at

https://tinyurl.com/DebuggingInputs

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. Special thanks go to Sascha Just for discussions on paper and ideas.

REFERENCES

[1] Alfred V Aho and Thomas G Peterson. 1972. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.* 1, 4 (1972), 305–312.
[2] Paul Eric Ammann and John C Knight. 1988. Data diversity: An approach to software fault tolerance. *Ieee transactions on computers* 4 (1988), 418–425.
[3] S. O. Anderson and Roland Carl Backhouse. 1981. Locally least-cost error recovery in Earley's algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 3, 3 (1981), 318–347.
[4] S. O. Anderson, Roland Carl Backhouse, E. H. Bugge, and CP Stirling. 1983. An assessment of locally least-cost error recovery. *Comput. J.* 26, 1 (1983), 15–24.
[5] assimp team. 2018. The Open-Asset-Importer-Lib. http://www.assimp.org
[6] Roland C Backhouse. 1979. *Syntax of programming languages: theory and practice*. Prentice-Hall, Inc.
[7] Johannes Buchner. 2017. ImageHash 4.0. https://pypi.org/project/ImageHash/
[8] Michael Burke and Gerald A. Fisher Jr. 1982. *A practical method for syntactic error diagnosis and recovery*. Vol. 17. ACM.
[9] Carl Cerecke. 2003. Locally least-cost error repair in LR parsers. (2003).
[10] James Clause and Alessandro Orso. 2009. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 249–260.
[11] Douglas Crockford. 2017. ECMA-404 The JSON Data Interchange Standard. https://www.json.org/
[12] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. 2006. Inference and Enforcement of Data Structure Consistency Specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1146238.1146266
[13] Brian Demsky and Martin Rinard. 2003. Automatic Detection and Repair of Errors in Data Structures. *SIGPLAN Not.* 38, 11 (Oct. 2003), 78–95. https://doi.org/10.1145/949343.949314
[14] B. Demsky and M. Rinard. 2005. Data structure repair using goal-directed reasoning. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. 176–185. https://doi.org/10.1109/ICSE.2005.1553560
[15] Stephen Dolan. 2018. Command-line JSON processor. https://stedolan.github.io/jq/
[16] Bassem Elkarablieh and Sarfraz Khurshid. 2008. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th international conference on Software engineering*. ACM, 855–858.
[17] Blender Foundation. 2018. blender.org - Home of the Blender project - Free and Open 3D Creation Software. https://www.blender.org
[18] Gephi. 2018. The Open Graph Viz Platform. https://gephi.org/
[19] GitHub. 2018. Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4
[20] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. 2018. Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing. *arXiv preprint arXiv:1810.08289* (2018).
[21] K Hammond and Victor J. Rayward-Smith. 1984. A survey on syntactic error recovery and repair. *Computer Languages* 9, 1 (1984), 51–67.
[22] Matthias Höschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 31–34.
[23] Ishtiaque Hussain and Christoph Csallner. 2010. Dynamic symbolic data structure repair. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, Vol. 2. IEEE, 215–218.
[24] GitHub Inc. 2018. REST API v3. https://developer.github.com/v3/
[25] ANTLR 4.7.1 API JavaDocs. 2018. Class DefaultErrorStrategy. https://www.antlr.org/api/Java/org/antlr/v4/runtime/DefaultErrorStrategy.html
[26] json simple. 2018. A simple Java toolkit for JSON. https://github.com/fangyidong/json-simple
[27] Eleftherios Koutsofios, Stephen North, et al. 1991. *Drawing graphs with dot*. Technical Report. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ.
[28] Tomasz Krawczyk. 1980. Error correction by mutational grammars. *Inform. Process. Lett.* 11, 1 (1980), 9–15.
[29] Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. 2014. Docovery: Toward Generic Automatic Document Recovery. In *International Conference on Automated Software Engineering (ASE 2014)*. 563–574.
[30] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.
[31] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. 2012. Automatic Input Rectification. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 80–90. http://dl.acm.org/citation.cfm?id=2337223.2337233
[32] 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.
[33] Jon Mauney and Charles N. Fischer. 1982. A forward move algorithm for LL and LR parsers. *ACM SIGPLAN Notices* 17, 6 (1982), 79–87.
[34] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th international conference on Software engineering*. ACM, 142–151.
[35] University of Utah. 2003. Wavefront OBJ Specification. https://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf
[36] The appleseedHQ Organization. 2018. appleseed - A modern, open-source production renderer. https://appleseedhq.net
[37] Terence Parr. 2018. ANTLR. http://www.antlr.org
[38] AT&T Labs Research. 2018. Graphviz - Graph Visualization Software. https://www.graphviz.org/
[39] Martin C. Rinard. 2007. Living in the Comfort Zone. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 611–622. https://doi.org/10.1145/1297027.1297072
[40] Jan Scheffczyk, Uwe M Borghoff, Peter Rödig, and Lothar Schmitz. 2004. S-DAGs: Towards efficient document repair generation. In *Proc. of the 2nd Int. Conf. on Computing, Communications and Control Technologies*, Vol. 2. 308–313.
[41] Chad D. Sterling and Ronald A. Olsson. 2007. Automated bug isolation via program chipping. *Software: Practice and Experience* 37, 10 (2007), 1061–1086.
[42] Ralf Sternberg. 2018. minimal-json - A fast and small JSON parser and writer for Java. https://github.com/ralfstx/minimal-json
[43] Microsoft Support. 2018. How to recover a lost Word document. https://support.microsoft.com/en-us/help/316951
[44] 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.
[45] 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