Automatically Reducing Tree-Structured Test Inputs

Satia Herfert
Department of Computer Science
TU Darmstadt, Germany
satiaherfert@gmx.de

Jibesh Patra
Department of Computer Science
TU Darmstadt, Germany
jibesh.patra@gmail.com

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany
michael@binaervarianz.de

***Abstract*—Reducing the test input given to a program while preserving some property of interest is important, e.g., to localize faults or to reduce test suites. The well-known delta debugging algorithm and its derivatives automate this task by repeatedly reducing a given input. Unfortunately, these approaches are limited to blindly removing parts of the input and cannot reduce the input by restructuring it. This paper presents the Generalized Tree Reduction (GTR) algorithm, an effective and efficient technique to reduce arbitrary test inputs that can be represented as a tree, such as program code, PDF files, and XML documents. The algorithm combines tree transformations with delta debugging and a greedy backtracking algorithm. To reduce the size of the considered search space, the approach automatically specializes the tree transformations applied by the algorithm based on examples of input trees. We evaluate GTR by reducing Python files that cause interpreter crashes, JavaScript files that cause browser inconsistencies, PDF documents with malicious content, and XML files used to tests an XML validator. The GTR algorithm reduces the trees of these files to 45.3%, 3.6%, 44.2%, and 1.3% of the original size, respectively, outperforming both delta debugging and another state-of-the-art algorithm.**

I. Introduction

Developers often have a test input that triggers behavior of interest, such as inducing a failure in a buggy program or covering particular parts of a program under test. However, the input may be larger than needed to preserve the property of interest. For example, consider a program that crashes the compiler or interpreter when given as an input. The larger this input program is, the more difficult it is to localize the fault, making the debugging process unnecessarily cumbersome [1].

To ease the task of dealing with such overly complex test inputs, several automated techniques have been proposed. Given a test input and an oracle that determines whether a reduced version of the input still preserves the property of interest, these techniques automatically reduce the input. With a reduced test input, the developer is likely to find the root cause of the bug faster and may even turn the reduced test input into a regression test case after the bug has been fixed. Similar, reducing test inputs while preserving some testing goal, such as coverage, can help to reduce a test suite.

Existing techniques for reducing inputs roughly fall into two categories. On the one hand, delta debugging [2] and its derivatives [3] reduce inputs in a language-independent way by repeatedly removing parts of the input until no further reduction is possible. While being simple and elegant, these approaches disregard the language of the input and therefore miss opportunities for input reduction. In particular, these techniques cannot restructure inputs, which often enables further reductions. As an example, consider the following JavaScript code and suppose that it triggers a bug, e.g., by crashing the underlying JavaScript engine.

```
for (var i = 0; i < 10; i++) {
  if (cond1 || cond2) {
    partOfBug();
  }
  if (cond3) {
    otherPartOfBug();
  }
}
```

Further suppose that the two function calls are sufficient to trigger the bug. That is, the following code is sufficient as a test input to enable a developer to reproduce and localize the bug:

```
partOfBug();
otherPartOfBug();
```

Unfortunately, existing language-independent techniques are challenged by this example. The original delta debugging algorithm blindly removes parts of the program, which is likely to lead to a syntactically invalid program or to a local minimum that is larger than the fully reduced example. Hierarchical delta debugging [3], a variant of delta debugging that considers the tree structure of the input, fails to find the reduced input because it can remove only entire subtrees, but it cannot restructure the input.

On the other hand, some techniques [4] exploit domain knowledge about the language of the test input. While being potentially more effective, hard-coding language knowledge into the approach limits it to a single kind of test input.

This paper presents the Generalized Tree Reduction algorithm (GTR), a language-independent technique to reduce arbitrary tree-structured test inputs. The approach is enabled by two key observations. First, we observe that transformations beyond removing entire parts of the input are beneficial in reducing inputs. GTR exploits this observation by incorporating tree transformations into the reduction process. The challenge is how to know which transformations to apply without hard-coding knowledge about a particular language. Addressing this challenge improves the effectiveness of test input reduction. Second, we observe that for most relevant input formats, there are various examples that implicitly encode

This work has been supported by the DFG within ConcSys, and by the BMBF and the HMWK within CRISP.

information about the language. For example, there are various programs in public code repositories, millions of HTML files, and many publicly available XML documents. GTR exploits this situation to improve the efficiency of input reduction by automatically pruning the search space of transformations for a particular language after learning from a corpus of example data.

Our work focuses on inputs that can be represented as a tree. This focus is motivated by the fact that the inputs of many programs have an inherent tree structure, e.g., XML documents, or can be easily converted into a tree, e.g., the abstract syntax tree of source code. The input to GTR is a tree with a desirable property, such as triggering a bug, and an oracle that determines whether a reduced version of the tree still has the desirable property. The algorithm reduces the tree level by level, i.e., it considers all nodes of a level to minimize the whole tree, before continuing with the next level. The output of the algorithm is a reduced tree that has the desirable property according to the oracle.

At the core of our approach are tree transformations that modify a tree into a new tree with fewer nodes. We describe two transformation templates that we find to be particularly effective. The first template removes a node and all its children, drastically shrinking the tree's size. As deleting nodes alone is insufficient for various inputs, the second template replaces a node with one of its children, i.e., it pulls up a subtree to the next level of the tree. While we find these two transformation templates to be effective, the algorithm is easily extensible with additional templates. In principle, these transformation templates are applicable to arbitrary kinds of nodes in the tree. To reduce the size of the search space considered by GTR, i.e., ultimately the time required to reduce an input, we specialize the transformation patterns to a specific input language by learning from a corpus of example data. Since the learning is fully automatic, the approach remains language-independent.

To evaluate GTR, we apply the algorithm to a total of 429 inputs in the form of Python programs, JavaScript programs, PDF documents, and XML documents. The Python programs each trigger a bug in the Python interpreter, while the JavaScript programs cause inconsistencies between browsers. The PDF documents contain malicious content. The XML documents achieve a certain coverage when given to an XML validator, and that coverage should be preserved during the reduction. We find that GTR reduces the inputs to 45.3%, 3.6%, 44.2%, and 1.3% of the original size, respectively. Compared to the best existing approach [3], GTR consistently improves efficiency and also significantly improves the effectiveness of reduction in three of four experiments.

To summarize, we make the following contributions:

- We identify the lack of restructuring as a crucial limitation of existing language-independent input reduction techniques.
- We present a novel tree reduction algorithm that transforms trees based on tree transformation templates. If a set of example inputs is available, the approach automatically specializes the templates to the language of the input.
- We show the presented algorithm to be significantly more effective and efficient than two state-of-the-art techniques.
- We make our implementation available to the public.[1]

II. Background

A. Delta Debugging

Zeller and Hildebrandt proposed delta debugging (DD) [2], a greedy algorithm for isolating failure inducing inputs. In a nutshell, DD splits the input in chunks of decreasing sizes, trying to remove some chunks while maintaining a property of the input. "Chunk" can refer, e.g., to individual characters or lines of a document. Often but not necessarily, the property is that the input induces a bug when fed to a program. DD does not guarantee to find the smallest possible input but instead ensures *1-minimality*. This property guarantees that no single part of the input can be removed without loosing the property of interest. For example, when applying line-based delta debugging to reduce a program that triggers a compiler bug, 1-minimality means that removing any line of the input will cause the input to not trigger the bug anymore.

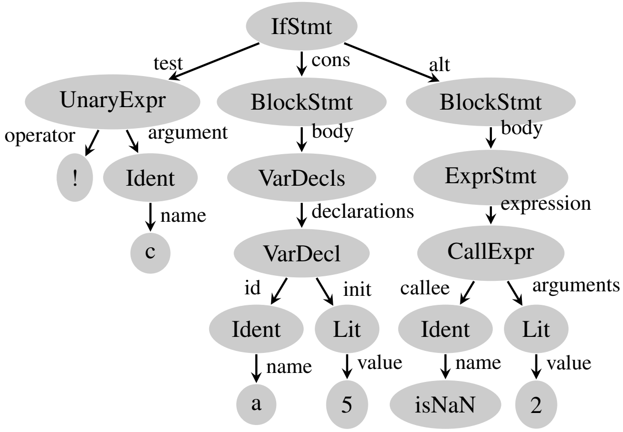
DD has an important disadvantage for structured input because it disregards the structure of the input when splitting it into chunks. As a result, DD may generate various invalid inputs and invoke the oracle unnecessarily. For instance, when applying DD to the example from the introduction, the algorithm may delete a closing bracket without removing its counterpart, generating a syntactically invalid program. Since each candidate input is given to the oracle, such invalid inputs increase the execution time of the algorithm.

B. Hierarchical Delta Debugging

Hierarchical delta debugging (HDD) [3] addresses the limitation that DD disregards the structure of the input. The algorithm considers the input to be a tree, which is a natural way to interpret various inputs, e.g., code represented as an abstract syntax tree (AST). HDD starts from the root of the tree and visits each level. At every level, the algorithm applies the original DD algorithm to all nodes at this level to find the smallest set of nodes necessary. The algorithm terminates after running DD on the last level of the tree.

HDD often purges large parts of the input early, leading to more reduction than DD, while also requiring fewer oracle invocations. In contrast to DD, HDD does not provide 1-minimality. To guarantee this property, HDD* repeatedly uses HDD, until no more changes to the tree are performed [3]. A limitation of HDD is that it only removed nodes (and all their children), but it does not use any other tree transformations. Therefore, for an input where the important part is deeply nested in the input tree, HDD produces far-from-minimal results. An example is the code excerpt provided in

[1] https://github.com/sherfert/GTR



```
1  if(!c) {
2    var a = 5; // root cause of failure
3  } else {
4    isNaN(2);
5  }
```

Fig. 1. An abstract syntax tree and code for our running example.

the introduction. Here, HDD attempts to remove the entire if-branches, which yields a program that does not trigger the bug anymore. The algorithm yields the following code as the reduced input:

```
for (;;) {
  if (cond1 || cond2) {
    partOfBug();
  }
  if (cond3) {
    otherPartOfBug();
  }
}
```

III. Problem Statement

Previous approaches for reducing failure-inducing inputs miss opportunities for reduction because they ignore the structure of the input and because they are limited to removing parts of the input. Motivated by these limitations, we aim for an algorithm that exploits the structure of the inputs, and that finds near-minimal results even when the root cause of a failure is deeply nested inside the input. Our work focuses on inputs that can be represented as a tree.

Definition 1. *A* labeled ordered tree *is a recursive data structure* (l, c)*, where* l *is a textual label and* c *is the (possibly empty) ordered list of outgoing edges. An edge* $e \in c$ *is a tuple* (l, t)*, where* l *is a textual label for the edge and* t *is the child node, which itself is a labeled ordered tree. We use* $\mathcal{T}$ *to refer to the set of all trees.*

Figure 1 shows an example: A small piece of JavaScript code that triggers a bug, e.g., in a JavaScript engine. Suppose that the bug is triggered by the statement at line 2. The example input can be represented as a tree – in this case, the AST.

We will refer to a labeled ordered tree hereafter simply as a tree or node, depending on the context. Trees have several properties. The *size* of a tree is the number of its nodes: $size : \mathcal{T} \to \mathbb{N}$. The tree in Figure 1 has a size of 19. The *context* of a tree is a partial function that returns the label of the parent node and the label of the incoming edge: $context : \mathcal{T} \to (String \times String)$. The context of the root node is undefined. For the example, the context of the $UnaryExpr$ node on the left side of our tree is $(IfStmt, test)$. The *level* of a node in a tree is the edge-distance from the node to the tree's root node. All nodes of a particular level in a tree can be obtained by a function $level : (\mathcal{T} \times \mathbb{N}) \to P(\mathcal{T})$, where $P(\mathcal{T})$ denotes the power set of $\mathcal{T}$. The *depth* of a tree is defined as the maximum distance of a leaf node to the root: $depth : \mathcal{T} \to \mathbb{N}$. The example tree has a depth of 5. Finally, we say that a tree t' is derived from another tree t, written $derived(t', t)$, if one can build t' from t by deleting nodes and edges, or by moving nodes and edges within the tree without changing a single label.

Definition 2. *An* oracle o *is a function that, given a tree, decides whether the tree provides a desired property:* $o : \mathcal{T} \to Bool$*. We use* $\mathcal{O}$ *to denote the set of all oracles.*

A tree t' is *minimal* w.r.t. an oracle o and a source tree t if t' satisfies the oracle and if there is no smaller derived tree that also satisfies the oracle. Formally, t' is minimal if $derived(t', t) \wedge o(t') = true \wedge \big(\nexists t'' \neq t' : derived(t'', t) \wedge o(t'') = true \wedge size(t'') < size(t')\big)$.

Definition 3. *A* tree reduction algorithm *is a function* $A : (\mathcal{T} \times \mathcal{O}) \to \mathcal{T}$ *that, given a tree* t *and an oracle* o *where* $o(t) = true$*, returns another tree* t' *for which* $o(t') = true$ *and* $size(t') \leqslant size(t)$*.*

The algorithm tries to find a smaller tree that still provides a property of interest, as decided by the oracle. If a reduction algorithm cannot further reduce a tree, it will return the same tree.

The goal of this work is to provide a tree reduction algorithm that returns near-minimal trees with respect to the given oracle, while maintaining the number of oracle invocations low. A small number of oracle invocations is important, as they can be costly operations, such as running a compiler, that significantly increase the overall runtime of the algorithm. In general, finding the minimal tree is impractical because the number of trees to check with the oracle grows exponentially with the size of the input tree.

IV. The Generalized Tree Reduction Algorithm

This section introduces a novel tree reduction algorithm, called *Generalized Tree Reduction* or GTR. Figure 2 shows the components of the approach and how they interact with each other. Given an input tree (step 1), the algorithm traverses the tree from top to bottom while applying transformations to reduce the tree. For example, a transformation may remove an entire subtree or restructure the nodes of the tree. The transformations are based on tree transformation templates (step 2) that specify a set of candidate transformations (step 3). To specialize a generic template to a particular input format, the approach optionally filters these candidates based on knowledge inferred from a corpus of example inputs (step 4). The algorithm applies the transformations and queries the

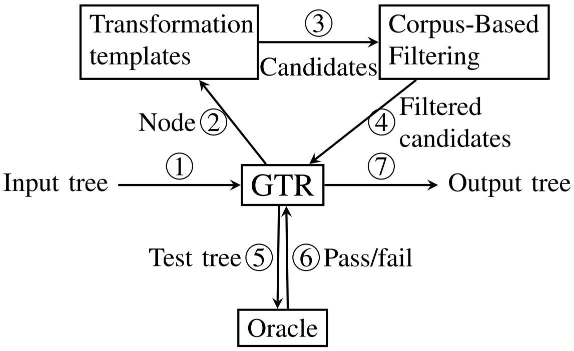


Fig. 2. Overview of the GTR approach.

oracle to check whether a reduced tree preserves the property of interest, e.g., whether it still triggers a particular bug (steps 5 and 6). The algorithm repeatedly reduces the tree until no more tree reductions are found. Finally, GTR returns the reduced tree (step 7).

We call the GTR algorithm "generalized" because it can express different tree reduction algorithms, depending on the provided tree transformation templates. For example, by providing a single template that reduces entire subtrees, GTR is equivalent to the existing HDD algorithm [3] (Section IV-E).

Before delving into the details of GTR, we illustrate its main ideas using the running example in Figure 1. Given the tree representation of the input, the algorithm analyzes the tree level by level, starting at the root node. For example, the algorithm considers a transformation that removes the root node *IfStmt* and all its children, but discards this transformation because the reduced tree (an empty program) does not trigger the bug anymore. As another example, the algorithm considers transformations that replace the root node with one of its children. Replacing the root node with the *BlockStmt* that represents the then-branch yields a smaller tree that still triggers the bug. Therefore, the algorithm applies this transformation and continues to further reduce the remaining tree. Eventually, the algorithm reaches a tree that represents only the statement `var a = 5`, which cannot be reduced without destroying the property of interest.

The remainder of this section explains the GTR algorithm in detail. At first, we present the tree transformations applied by the algorithm (Sections IV-A and IV-B). Then, we describe how GTR combines different transformations into an effective tree reduction algorithm (Section IV-C).

A. Tree Transformation Templates

The core ingredient of GTR are transformations that reduce the size of a tree. We specify such transformations with templates:

Definition 4. *A* transformation template *is a function* $\mathcal{T} \rightarrow P(\mathcal{T}) \cup \{DEL\}$ *that returns a set of candidate trees that are the result of transforming a given input tree. In addition to candidate trees, the template may return the special symbol DEL, which indicates that the tree should be removed rather than modified.*

Algorithm 1 Substitute-by-child template

```
Input: a tree $tree$
Output: a set of candidates nodes
function SBC_TEMPLATE($tree$)
    $candidates \leftarrow \emptyset$
    for $i \in [0, |tree.c|]$ do
        $c \leftarrow tree.c[i].t$
        $candidates \leftarrow candidates \cup \{c\}$
    return $candidates$
```

In this paper, we focus on two transformation templates, which yield a tree reduction algorithm that is more effective than the best existing algorithms.

a) Deletion Template: The first template addresses situations where an entire subtree of the input given to GTR is irrelevant for the property of interest. In our running example (Figure 1), the subtree rooted at the right-most *BlockStmt* node is such a subtree. To enable GTR to remove such subtrees, the deletion template simply suggests for each given tree to delete it by returning the special DEL symbol.

b) Substitute-by-child Template: The second template addresses situations where simply removing an entire subtree is undesirable because the subtree contains nodes relevant for the property of interest. We observe that a common pattern is that the root node of a subtree is irrelevant but one of its children is important for the property of interest. In the running example, the tree rooted at the *IfStmt* matches this pattern, because the if statement is irrelevant, but the nested variable declaration is crucial. To address this pattern, the substitute-by-child template (Algorithm 1) returns each child of a given tree's root node as a candidate for replacing the given tree. The template iterates over all children of the given tree and adds each of them to the set of candidates. Applying this transformation template to the *IfStmt* of the running example yields a set of three candidates, namely the three subtrees rooted at nodes *UnaryExpr*, *BlockStmt*, and *BlockStmt*.

Beyond these two templates, additional templates can be easily integrated into GTR, enabling the approach to express different tree reduction algorithms.

B. Corpus-Based Filtering

The templates defined above are completely language independent. When applying these templates to a tree that ought to conform to a specific input format, many of the candidates may be rejected by the oracle simply because they violate the input format. For example, when the deletion template suggests to remove the *UnaryExpr* from the tree in Figure 1, the resulting tree corresponds to syntactically invalid JavaScript code because every if statement requires a condition. Suggesting such invalid candidates does not influence the effectiveness of our approach because the oracle rejects all invalid candidates. However, a high number of invalid candidates negatively influences the efficiency of the approach since invoking the oracle often imposes a significant runtime cost.

To address the challenge of invalid candidates, we enhance the approach with a language-specific filtering of candidates trees that rejects invalid trees before invoking the oracle. To preserve the language-independence of GTR, the filtering is based on knowledge that gets automatically inferred from a corpus of example inputs in the specific input format. For example, the approach learns from a set of JavaScript programs that if-statements require a condition, and therefore, will filter any candidates that violate this requirement.

a) Deletion Template: To specialize the deletion template to a particular language, we need to know which edges are mandatory for particular node types. The approach analyzes the code corpus to find a set of mandatory edge labels for each node label. An edge is considered mandatory, if it appears on all nodes with the label across the whole corpus. Based on the mandatory edges, we modify the deletion template so that a node only can be deleted if it is not a mandatory child of its parent node.

For the running example, consider again the candidate that suggests to remove from the *IfStmt* the *UnaryExpr* subtree. The corpus analysis finds that the set of mandatory edges for an *IfStmt* is $\{test, cons\}$. Based on this inferred knowledge, the algorithm will not attempt to delete the *UnaryExpr* anymore, but discards this candidate before needlessly passing the tree to the oracle.

b) Substitute-by-child Template: To specialize the substitute-by-child template, we gather information on the parent node labels and incoming edge labels of nodes. Specifically, for each node label we collect a set of pairs (p, e) where p is the label of the parent, and e is the label of the incoming edge. This set of pairs is equivalent to all distinct contexts of nodes with that label and we call it the *allowed contexts*. We then replace line 5 of the substitute-by-child template (Algorithm 1) with the following steps:

```
5: if $context(tree) \in allowedContexts(tree.c[i].l)$ then
6:     $candidates \leftarrow candidates \cup \{c\}$
```

The specialized variant of the template checks if the child that we replace the node with can also appear in the same context as the node. For example, the approach infers that there is one valid context for a *VarDecl* node, namely *(VarDecls, declarations)*. Since *(BlockStat, body)* is not a valid context, the algorithm will immediately discard a candidate that tries to substitute *VarDecls* with *VarDecl*.

Inferring from a corpus of examples how to specialize language-independent transformation templates to an input format is optional and automatic. It is automatic because for most input formats used in practice, there are sufficiently many examples to learn from. An alternative approach could be to use a formal grammar of the input language to filter syntactically invalid trees. We rejected this idea because (i) a grammar may not be available, e.g., for proprietary formats, (ii) the checks performed by the specialized transformation templates are more lightweight than parsing the entire input tree with a grammar. Our evaluation measures the effectiveness and efficiency of GTR with and without the corpus-based filtering of candidate trees (Section V-D).

Algorithm 2 Generalized tree reduction

```
Input: tree $t$, oracle $o$, set $\mathcal{L}$ of templates
Output: reduced tree
 1: for $i \in [0, depth(t)]$ do
 2:     for $l \in \mathcal{L}$ do
 3:         $t \leftarrow$ APPLYTEMPLATE$(t, i, o, l)$

 4: function APPLYTEMPLATE$(t, i, o, l)$
 5:     $levelNodes \leftarrow level(t, i)$                ▷ All nodes of level $i$
 6:     if $l$ returns at most one transformation then
 7:         $newNodes \leftarrow$ apply DD to replace $levelNodes$ using $l$
 8:         return tree where $newNodes$ replace $levelNodes$
 9:     else
10:         return reduceLevelNodes($t$, $levelNodes$, $o$, $l$)   ▷ Alg. 3
```

C. GTR Algorithm

Based on the transformation templates described above, the GTR algorithm reduces a given tree by applying transformations at each level of the tree. Algorithm 2 summarizes the main steps. Starting at the root node, the algorithm considers each level of the tree and applies all available transformation templates to each level using a helper function $applyTemplate$ (lines 1 to 3).

a) Delta Debugging-based Search: When applying a template to the nodes at a particular level, the algorithm distinguishes between templates that return at most one candidate transformation, such as the deletion template, and other templates. In the first case, the algorithm needs to decide for which nodes to apply the suggested transformations. This problem can be reduced to delta debugging (DD). The chunks needed as input for DD are the nodes of the level. DD then tries to combine as many replacements as possible while querying the oracle to check if a replacement preserves the property of interest. For each node n for which the transformation template returns a node n', DD will try to replace n with n'. For each node where the symbol DEL is returned, DD will try to delete n. After deciding on the replacements, the result is a new list of nodes for the current level. The helper function $applyTemplate$ replaces the nodes on the level with the new nodes and returns the resulting tree to the main loop of the algorithm (lines 7 and 8).

b) Backtracking-based Search: For templates that may return more than one candidate, the algorithm must decide not only whether to apply a candidate replacement but also which of the suggested candidate replacements to apply. This problem cannot be easily mapped to DD because DD assumes to have exactly one option per chunk (typically, whether to delete it or not). Instead, we present a backtracking-based algorithm that searches for a replacement of nodes on a particular level that reduces the overall tree. Similar to DD, the algorithm is a greedy search.

Algorithm 3 summarizes the main steps of the backtracking-based search for replacements of nodes on a particular level. The algorithm is called at line 10 of the main GTR algorithm. The central idea is to try different *configurations* that specify which replacements to use for each node. The algorithm starts with a configuration that replaces each node with itself (lines 1

Algorithm 3 Backtracking-based reduction of level nodes

```
Input: tree $t$, list of $nodes$ on the same level, oracle $o$, template $t$
Output: reduced tree
      ▷ Maps each node to its current replacement:
 1: $conf \leftarrow$ empty map
 2: for $n \in nodes$ do
 3:     $conf.put(n, n)$
 4: repeat
 5:     $improvementFound \leftarrow false$
 6:     for $n \in nodes$ do
 7:         $currentRep \leftarrow conf.get(n)$
 8:         for $n' \in l(n)$ where $size(n') < size(currentRep)$ do
 9:             $t' \leftarrow t$ with each $n$ replaced by $n'$
10:             $conf.put(n, n')$
11:             if $oracle(t')$ then
12:                 $improvementFound \leftarrow true$
13:                 $currentRep \leftarrow n'$
14:             else
15:                 $conf.put(n, currentRep)$            ▷ Backtrack
16: until $\neg improvementFound$
17: return $t'$
```

to 3). Then, the algorithm iterates through all nodes (line 6) and tries all configurations where the replacement candidate is smaller than the currently chosen replacement. That is, the algorithm avoids invoking the oracle for replacements that are less effective than an already found replacement. If the oracle confirms that replacing a node preserves the property of interest, an improvement was found w.r.t. the current replacement (lines 12 and 13). Otherwise, the algorithm must revert the replacement and backtracks to the previous configuration (line 15).

The algorithm repeats the search for a replacement of any of the nodes on the current level until no further improvement is found. The reason for repeatedly considering the list of nodes is that using an effective replacement at a later node may enable using previously impossible replacements at previous nodes, which have already been tested in the current iteration. For example, consider the following input, where the `crash(b)` call ensures the property of interest:

```
a = b = 0;
if (a)
  crash(b);
```

During the first iteration of the main loop (lines 4 to 16), the algorithm cannot reduce the assignments in the first line but reduces the input by substituting the if-statement with the `crash(b)` call. Now, during the second iteration, the algorithm again considers the assignment statement and successfully reduces it to `b = 0`, which yields the following reduced input:

```
b = 0;
crash(b);
```

The search for a reduction of the nodes in the current level guarantees to find a local optimum, i.e., a configuration where using any other replacement that yields a smaller subtree would not satisfy the oracle. As the search is greedy, it may miss a configuration that yields a smaller overall tree satisfying the oracle. Searching for a global optimum would require to explore all possible configurations, which is exponential in the number of candidates suggested.

Algorithm 4 GTR*

```
Input: tree $t$, oracle $o$, set $\mathcal{L}$ of templates
Output: 1-transformation-minimal tree
current ← t
repeat
    $previous \leftarrow current$
    $current \leftarrow GTR(previous, o, \mathcal{L})$
until $size(previous) = size(current)$
return $current$
```

c) Example: We illustrate GTR on the running example. Recall that only line 2 of Figure 1 is relevant for reproducing the bug. The algorithm starts on level 0, which contains only the *IfStmt*, and invokes *applyTemplate* with the deletion template. Deletion returns at most one transformation. Therefore, the algorithm applies DD to the node on this level and tries to delete it with all its children. However, this deletion would make bug disappear and is discarded by the oracle.

Next, *applyTemplate* is invoked with the substitute-by-child template. Since this template may return multiple candidates, the algorithm invokes the backtracking-based *reduceLevelNodes* function, i.e., Algorithm 3. There is only one node to consider in line 6 of Algorithm 3, and in line 8 three different candidates are tested. The first is the *UnaryExpr* on the left side. This candidate has a size of 4. But, since the important code piece is removed, line 15 reverts this change. The next candidate is the *BlockStatement* in the middle. It has a size of 7. The oracle returns true for this transformation, so $currentRep$ is updated in line 13. The third candidate is the *BlockStatement* on the right. Since it also has a size of 7, which is not smaller than the size of $currentRep$, the candidate is not tested. Now that an improvement was found, the main loop (lines 4 to 16) is repeated. As there is only one node, nothing new will be tested. After having finished both templates on level 0, GTR will advance to level 1 and continue in the same manner.

D. GTR* Algorithm

The existing DD and HDD* algorithms guarantee 1-minimality and 1-tree-minimality, respectively. In essence, this property states that, given a reduced input, there is no single reduction step that can further reduce the input. We define a similar minimality property for GTR:

Definition 5. *A tree t is called* 1-transformation-minimal *w.r.t. an oracle o and a set of templates $\mathcal{L}$ if $o(t) = true \wedge \forall n$ in t and $\forall l \in \mathcal{L}$, there is no candidate n' in $l(n)$ that, when replacing n with n' yields a tree t' with $o(t') = true \wedge size(t') < size(t)$.*

In other words, for 1-transformation-minimal trees, all trees obtained by single replacements of one node of the tree cause the oracle to return $false$. The main difference to the existing 1-minimality and 1-tree-minimality properties is to consider arbitrary tree transformations.

The GTR algorithm does not guarantee to find a 1-transformation-minimal tree. The reason is that by optimizing a tree

TABLE I
INPUT FILES USED FOR THE EVALUATION.

Format	Inputs	Bytes			Lines			Nodes		
		Min	Med	Max	Min	Med	Max	Min	Med	Max
Python	7	212	483	1,574	19	22	73	82	232	591
JavaScript	41	32	515	21,806	1	28	458	19	223	5,529
PDF	371	2,901	22,966	1,167,807	–	–	–	155	258	4,324
XML	10	7,225	29,065	51,180	170	500	888	319	1,042	1,823

on one level, a transformation on a higher level, which had been rejected by the oracle before, can become possible. To guarantee 1-transformation-minimality, we present a variant of GTR, the GTR* algorithm (Algorithm 4). GTR* repeats GTR until the tree does not change its size anymore, which indicates that no transformation can be applied thereafter.

E. Generalization of HDD and HDD*

GTR and GTR* generalize the existing HDD and HDD* algorithms, respectively. To obtain HDD, we configure GTR to include only the deletion template, without specializing the template to a particular language. The resulting algorithm applies DD on every level of the input tree by deleting a subset of the nodes on this level. This behavior is exactly what HDD does, i.e., the reduced tree is the same as returned by HDD. This generalization also applies to HDD*, where we simply run GTR* with the variant of GTR that is equivalent to HDD.

V. EVALUATION

We evaluate GTR by applying it to four input formats and usage scenarios, including reducing fault-inducing inputs for debugging, reducing malicious inputs for easier security analysis, and reducing test inputs for more efficient testing. The evaluation compares GTR and GTR* to the existing DD, HDD, and HDD* algorithms. We focus on three research questions:

- RQ1: How effective is the approach in reducing trees?
- RQ2: How efficient is the approach?
- RQ3: What are the effects of specializing transformation templates to an input format?

A. Experimental Setup

1) Input Formats and Oracles: We consider four sets of inputs that comprise a total of 429 input files that can be represented as a tree. Table I summarizes the inputs and shows their size in terms of bytes, lines, and number of tree nodes. For the binary PDF format we do not report lines.

a) Failure-inducing Python Code: We use GTR to reduce Python files that cause the Python interpreter to crash. To obtain such files, we search the Python bug tracker for segmentation faults and stack overflows reported along with code to reproduce it. Because these files have been reported by users or developers, they are likely to have been manually reduced, presenting a non-trivial challenge to any input reduction algorithm. We use a Python parser [5] to represent code as trees. The oracle to check whether a reduced Python file preserves the property of interest is to execute the file and to check the status code returned by the Python interpreter. Only checking the status code bears the risk of misclassification, e.g., if the program is altered to return that status code without triggering the bug. Given the low number of inputs, we could exclude this possibility manually for the given inputs. As a corpus to specialize the transformation templates, we gather 900 files from popular (measured by number of stars) GitHub projects.

b) Inconsistency-exposing JavaScript Code: We also use GTR to reduce JavaScript files that cause inconsistencies between browsers. The files are generated by TreeFuzz [6], an existing fuzz testing technique. We configure TreeFuzz to generate 3,000 files and keep all files that trigger a browser inconsistency, which results in 41 files. Since these files are automatically generated, they generally contain parts that are not required to trigger a browser inconsistency, providing a good data set to complement the manually written Python files. We use Esprima [7] to transform code to trees. As the oracle, we compare the runtime behavior of a JavaScript file in Firefox 25 and Chrome 48, as described in [6]. This oracle compares read and written values as well as error types and messages. The JavaScript corpus for specializing transformation templates comprises around 140,000 files [8].

c) Malicious PDF Documents: As a usage scenario beyond reducing inputs for debugging, we use GTR to reduce malicious PDF files while preserving their maliciousness, which facilitates further security analysis. We download malicious PDF files from the Contagio malware dump [9]. PDF documents are binary data but have an internal tree structure. Using the pdfminer [10] and iTextPDF [11] libraries, we convert between PDFs and trees. We filter the PDFs to keep only those that are classified as malicious according to PDF Scrutinizer [12] and that are compatible with our tree conversion. Out of the over 8,000 remaining files we chose a random subset of 371 files. As the oracle, we check whether PDF Scrutinizer classifies a file as malicious. This oracle may accept a malformed PDF and classify it as malicious. This behavior is desired, since PDF viewers try to display even malformed PDFs and could thus still execute harmful code. In contrast to the above formats, PDF trees have weaker constraints over their nodes, and the malicious content, contained in embedded objects, is typically not spread over the tree. The PDF corpus for specializing templates are 16,000 files from the Contagio malware dump, including both malicious and benign documents.

d) Test Suite of XML Files: As another usage scenario, we use GTR for test suite reduction, i.e., reducing test cases while preserving the code coverage. We download a corpus of more than 140,000 XML files that adhere to the same XML document type definition (DTD) [13]. From the corpus, we select a random subset of 10 XML files and parse them using the xmllint [14] XML validation tool. Subsequently, we measure the coverage in xmllint using gcov [15]. As the oracle, the coverage in xmllint using a reduced XML file must be at least the original coverage. For XML there was no necessity to specialize the templates because both valid and invalid XML files are accepted by xmllint. Therefore, we omit the template

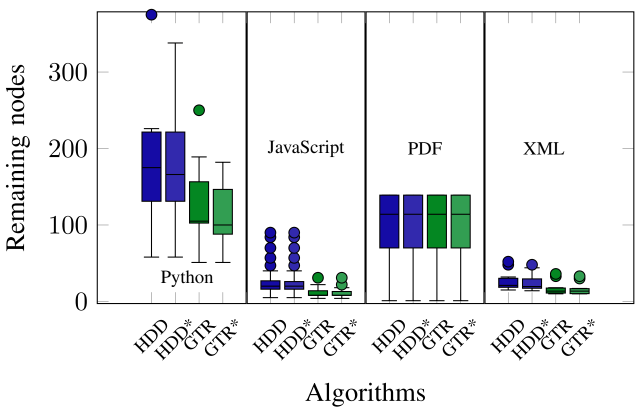


Fig. 3. Effectiveness of reduction measured in terms of the number of the remaining tree nodes. The boxes indicate the median and the first and third quartiles. The whiskers include up to 1.5 inter-quartile-ranges above and below the box.

specialization step.

2) State of the Art Approaches: We compare our approach to our own implementations of the existing DD, HDD, and HDD* algorithms. The DD implementation works on the line-level, i.e., each line of the input is a chunk considered by DD. The HDD implementation uses the same tree representation of the inputs as the GTR implementation.

B. *Effectiveness*

To evaluate the effectiveness in reducing test inputs, we apply GTR and GTR* to the inputs in Table I. To measure effectiveness, we compute the remaining size relative to the original inputs, measured both in terms of the file size and the tree size. Since DD does not represent inputs as trees, we measure only the file size for DD-reduced inputs.

Table II summarizes the results. The table shows for each approach the remaining file sizes and nodes, along with the percentage of the original size. Each value is the median over all inputs we consider. The best approach for a particular measure and input format is highlighted. Overall, GTR* consistently yields the smallest remaining trees (closely followed by GTR), with 43.1%, 3.6%, 44.2%, and 1.3% of the original size for Python, JavaScript, PDFs, and XML files, respectively.

To better understand the variations in effectiveness across different inputs of a format, Figure 3 shows the distribution of remaining tree sizes. The figure shows that, even though the effectiveness varies across inputs, GTR and GTR* outperform the other approaches for most inputs.

We further discuss our results for the different formats:

- *Python*. For the Python data set, GTR* produces the smallest trees and GTR the second smallest. The relative reduction is not as high as for other formats. The reason is that these inputs have been reduced manually before reporting them to the Python developers, which leaves little room for any subsequently applied tree reduction algorithm.
- *JavaScript*. For the JavaScript data set, we observe larger reductions by all algorithms, sometimes removing more than 99% of the file. The main reason is that these files are generated by a fuzz tester and have not been processed by a human. GTR and GTR* consistently outperform all other algorithms.
- *PDF*. For the PDF data set, all tree-based algorithms are equally efficient, leaving only 44.2% of the nodes in a tree, on average. The reduced file size is about 75%, i.e., larger than the reduced tree size. The reason is that a few nodes in the tree representation of a PDF are large objects, such as images or embedded code, and that these large objects often contain the malicious content. Surprisingly, DD actually achieves marginally better file size reductions compared to the other algorithms. However, after examining these files manually, we noticed they were not valid PDF files anymore, even though PDF Scrutinizer still flags them as malicious. These syntactically invalid files would likely not help a security analyst that much. In contrast, removing more than half of a PDF's nodes is a vast improvement for a security analyst who manually inspects the file's content.
- *XML*. For the XML test suite, GTR and GTR* achieve the best reductions, sometimes removing up to 98% of the trees. DD is only able to reduce the XML files minimally. Our hypothesis is that when DD removes random lines, a malformed XML file result, which trigger only the error handling code of xmllint.

In summary, GTR and GTR* are more or as effective as the best existing input reduction approach with respect to the remaining tree size. Using GTR, the median percentage of nodes after reduction for four different input formats is 45.3%, 3.6%, 44.2%, and 1.3%, respectively.

C. *Efficiency*

We evaluate the efficiency of our approach by measuring the number of oracle invocations required to reduce a tree. Using this metric instead of, e.g., wall clock time, is motivated by two reasons. First, invoking the oracle typically is the most important operation during automated input reduction, because it often involves running a complex piece of software, such as a compiler or interpreter, on non-trivial inputs, such as large programs. Second, wall clock time is highly dependent on the implementation of the tree reduction and the oracle. To check that the number of oracle invocation is a meaningful measure, we compare the time spent in the oracle with the time in other parts of the algorithm for the JavaScript data set. For each algorithm, the oracle invocation time dominates and comprises more than 98% of the total execution time, on average.

Table II shows the median number of oracle invocations required by the different approaches to a reduce a single file. Figure 4 illustrates the distributions of this number for each input format. For three of the four formats, GTR and GTR* require fewer oracle invocations, i.e., are more efficient, than their counterparts HDD and HDD*. The GTR* and HDD* algorithms both need more invocations than their *-less counterparts, which is unsurprising because they run the algorithm, including oracle invocations, multiple times.

TABLE II

EFFECTIVENESS AND EFFICIENCY OF REDUCTION BY GTR AND GTR* COMPARED TO BASELINE APPROACHES. FOR EACH MEASURE, THE BEST APPROACH IS HIGHLIGHTED. WE REPORT THE MEDIAN VALUES OVER ALL FILES OF A DATA SET.

	DD	HDD	HDD*	GTR	GTR*
Failure-inducing Python code:					
Remaining file size	359 (74.3%)	437 (90.5%)	423 (87.6%)	301 (62.3%)	261 (54.0%)
Remaining nodes	-	175 (75.4%)	166 (71.6%)	105 (45.3%)	100 (43.1%)
Oracle invocations	125	809	1,089	205	492
Inconsistency-exposing JavaScript code:					
Remaining file size	84 (16.3%)	49 (9.5%)	49 (9.5%)	28 (5.4%)	28 (5.4%)
Remaining nodes	-	20 (9.0%)	20 (9.0%)	8 (3.6%)	8 (3.6%)
Oracle invocations	77	21	22	16	17
Malicious PDF documents:					
Remaining file size	17,225 (75.0%)	17,304 (75.3%)	17,304 (75.3%)	17,304 (75.3%)	17,304 (75.3%)
Remaining nodes	-	114 (44.2%)	114 (44.2%)	114 (44.2%)	114 (44.2%)
Oracle invocations	358	509	665	239	389
Test suite of XML files:					
Remaining file size	28,897 (99.4%)	1,259 (4.3%)	1,246 (4.3%)	1,271 (4.4%)	940 (3.2%)
Remaining nodes	-	21 (2.0%)	20 (1.9%)	14 (1.3%)	14 (1.3%)
Oracle invocations	1,746	92	107	100	114

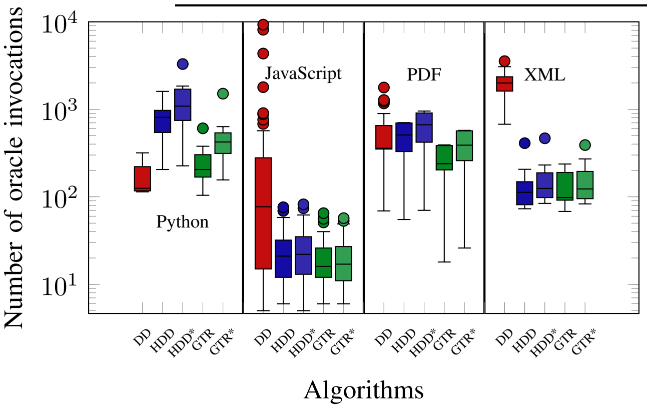


Fig. 4. Number of oracle invocations. Note the logarithmic scale. The boxes indicate the median and the first and third quartiles. The whiskers include up to 1.5 inter-quartile-ranges above and below the box.

We next discuss the results for the different formats:

- *Python*. GTR needs 64% more invocations than DD but HDD needs 295% more invocations than GTR.
- *JavaScript*. GTR is the most efficient approach. HDD needs 31% more invocations, and DD 381% more invocations.
- *PDF*. GTR is the most efficient approach. HDD needs 113% more invocations, and DD 150% more invocations.
- *XML*. HDD is the most efficient approach. GTR needs 9% more invocations, and DD 1898% more invocations.

The large difference in the results for DD can be explained by the size of the input files. Since the Python files are relatively small and cannot be reduced as much as the relatively large JavaScript files, DD reduces them quickly. In contrast, the structure-unaware search of DD takes significantly more oracle invocations for larger files.

To summarize, GTR is either more efficient or only slightly less efficient than the best existing approach.

D. Benefits of Corpus-Based Filtering

Our approach specializes language-independent transformation templates to a specific input language by learning filtering rules from a corpus of examples of inputs. To evaluate how the corpus-based filtering influences the effectiveness and efficiency of GTR, we compare the approach with a variant of GTR that does not filter any candidate transformations. For both variants, we perform the same experiments as described in Sections V-B and V-C, except for the XML format, where we do not use any corpus.

We find that the GTR variant without filtering of candidates achieves the same effectiveness for JavaScript and PDF and slightly higher reductions (5%) for Python. The reason is that the corpus does not mirror all facets of the target languages, which may cause the filtering to overly constrain the transformations. For example, if the corpus would not contain any if-statement without an else-branch, then GTR would not consider removing the else-branch. Fortunately, the results show that such overly constrained filtering is very unlikely.

The GTR variant without filtering needs significantly more oracle invocations. For the Python data set, the variant needs 423 invocations (median), whereas the full GTR approach needs only 205 invocations. For the JavaScript data set, the results are 30 and 16 invocations without and with filtering, and for the PDF data set 611 and 239 invocations, respectively.

Finally, we measure how long extracting language-specific information from the corpus takes. In total, extracting this knowledge takes around 21 minutes, 19 seconds, and 15 seconds for the JavaScript, Python and PDF data sets, respectively.

In summary, comparing GTR with and without specialization transformations shows that both variants are roughly equally effective and that the specialization significantly improves the efficiency of the algorithm.

VI. Related Work

A. Minimizing Test Inputs

Delta debugging (DD) [2] sets the foundations to automate the process of minimizing test inputs. In contrast to GTR, it cannot handle large structured inputs effectively. Recently, parallelization of DD has been explored [16], which is orthogonal to our contributions. The hierarchical variant HDD [3] applies DD on hierarchical documents. However, HDD fails to restructure trees in a way that allows obtaining significantly smaller results, and also is less efficient than GTR. An improved HDD variant proposed later uses a different kind of grammars [17]. This targets the conversion of code documents to trees and is complementary to our findings.

C-Reduce [4] is a variant of DD that applies domain-specific transformations to reduce C code. These transformations include changing identifiers and constants, removing pairs of parentheses or curly braces, or inlining functions. One big advantage of C-Reduce is to never produce any input with nondeterministic or undefined behavior. There are two big differences in comparison to our approach. First, the transformations are source-to-source and not tree-to-tree. Second, C-Reduce loses generality by applying domain-specific changes to the document. However, many of the transformation included by C-Reduce can be expressed in a more general way and are included in our approach. For example, "removing an operator and one of its operands (e.g., changing a+b into a or b)" is equivalent to replacing the operator node with one of its children in the tree.

When the input to a program is not as complex as a code document itself, more efficient techniques can be employed. One possibility is to minimize the path constraints of the input that led to a particular failure [19]. However, the set of path constraints grows exponentially for more complex inputs, rendering this approach unfeasible for code inputs.

Another interesting approach taints parts of each input to identify the parts relevant to a failure [20]. The default setting taints each byte independently, making it possible to also account for complex inputs. At the same time, this disregards the structure of the document, similar to DD, and thus becomes overly expensive. Another setting tracks inputs on a per-entity basis, which is insufficient for test input reduction.

Test input reduction is particularly important for automatically generated test inputs. Fuzzing [21] is a popular technique to create such inputs. It has been successfully applied to various kinds of input formats, including program code to test compilers and runtime engines [22, 23, 24, 25, 26], program code to test refactoring engines [27], document formats to test word processors [28], and structured text formats for security testing [29, 30]. All these formats can be represented as a tree and may benefit from reduction via GTR.

B. Fault Localization

Various techniques aim at localizing a fault in the buggy program itself, instead of the input. Zeller applied DD also to the program with this goal [31]. There are various other approaches for fault localization [32, 33, 34]. Program slicing reduces a program while maintaining its behavior with respect to a particular variable [35]. Ideally, the smaller slice contains the bug and eases its localization. Dynamic slicing [36] focuses on the subset of the program that give a variable its value *with the current input*. Just slicing the variables that appear in the line causing the bug (if known) does not guarantee to obtain a program that produces the same buggy behavior, though. The combination of DD with dynamic forward and backward slicing has also been explored previously [37]. Another approach is to record traces to find shorter program executions with the observed buggy behavior [38, 39, 40], which ultimately also reveals likely locations for the bug.

Fault localization and test input reduction have different goals. In a first step, a tester confronted with a failure needs a small (and fast running) input to reproduce the failure. In a second step, the bug must be located in the program and fixed. Finally, the small input can be turned into a regression test. Thus, both techniques complement each other.

C. Minimizing Test Suites

While randomly generating tests, high code coverage can be achieved. The tests in randomly generated suites are often rather big and can also benefit from a reduction with DD [41, 42]. To maintain the good coverage of the test suite, the oracle can be modified to account for that, instead of testing for particular failures [43, 44].

D. Inference of Language Constraints

Our corpus-based filtering relates to work on inferring grammars [18, 45, 46] and probabilistic models of structured program inputs [6]. As an alternative to inferring language constraints from a corpus, GTR could reuse inferred grammars and models to prune candidate trees.

VII. Conclusion

We present GTR, a novel algorithm to reduce tree-structured test inputs in a generalized and language-independent way. Our algorithm applies tree transformations hierarchically to reduce a given test input. The algorithm combines Delta Debugging and a greedy backtracking-based search to choose which transformations to apply. To specialize generic tree transformation templates to a particular input format, GTR automatically infers language-specific filters from a corpus of examples. We compare our approach with three existing algorithms, DD, HDD, and HDD*, on 429 test inputs. In three of four experiments, GTR outperforms other algorithms in reduction effectiveness. At the same time, GTR is either only slightly less or even more efficient than the best existing approach. We envision GTR to be applied to various problems that benefit from reduced inputs, e.g., to reduce bug-triggering inputs provided by users or fuzz testing techniques, to reduce test suites for more efficient test execution, or to reduce potentially malicious code or documents before a manual security analysis.

References

[1] I. Vessey, “Expertise in debugging computer programs: A process analysis,” *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[2] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[3] G. Misherghi and Z. Su, “HDD: Hierarchical delta debugging,” in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 142–151.

[4] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-case reduction for C compiler bugs,” in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 335–346.

[5] Python Software Foundation. (2017) Python AST library. [Online]. Available: https://docs.python.org/2/library/ast.html

[6] J. Patra and M. Pradel, “Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data,” TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664, Nov 2016.

[7] A. Hidayat. (2017) Esprima. [Online]. Available: http://esprima.org/

[8] Multiple authors. (2016) Learning from ”Big Code”. [Online]. Available: http://learnbigcode.github.io/datasets/

[9] M. Parkour. (2011) Contagio malware dump. [Online]. Available: http://contagiodump.blogspot.de/2010/08/malicious-documents-archive-for.html

[10] Y. Shinyama. (2014) PDFMiner. [Online]. Available: https://euske.github.io/pdfminer/

[11] iText Group NV. (2017) iText PDF. [Online]. Available: http://itextpdf.com/

[12] F. Schmitt, J. Gassen, and E. Gerhards-Padilla, “PDF Scrutinizer: Detecting JavaScript-based attacks in PDF documents,” in *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*. IEEE, 2012, pp. 104–111.

[13] Hindawi. (2017) Hindawi XML Corpus. [Online]. Available: https://www.hindawi.com/corpus/

[14] xmllint. (2017) The XML C parser and toolkit of Gnome. [Online]. Available: http://xmlsoft.org/xmllint.html

[15] gcov. (2017) gcov – a test coverage program. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.3.4/gcc/Gcov.html

[16] R. Hodován and A. Kiss, “Practical improvements to the minimizing delta debugging algorithm,” in *Proceedings of the 11th International Joint Conference on Software Technologies*, 2016, pp. 241–248.

[17] R. Hodován and Á. Kiss, “Modernizing hierarchical delta debugging,” in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM, 2016, pp. 31–37.

[18] Z. Lin and X. Zhang, “Deriving input syntactic structure from execution,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 83–93.

[19] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, “Finding bugs in dynamic web applications,” in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 261–272.

[20] J. Clause and A. Orso, “Penumbra: automatically identifying failure-relevant inputs using dynamic tainting,” in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 249–260.

[21] P. Oehlert, “Violating assumptions with fuzzing,” *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[22] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing.” in *PLDI*, 2008, pp. 206–215.

[23] V. Ganesh, T. Leek, and M. C. Rinard, “Taint-based directed whitebox fuzzing,” in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 474–484.

[24] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments.” in *USENIX Security Symposium*, 2012, pp. 445–458.

[25] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 216–226. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594334

[26] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, “Many-core compiler fuzzing,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 65–76. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737986

[27] B. Daniel, D. Dig, K. Garcia, and D. Marinov, “Automated testing of refactoring engines,” in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 185–194.

[28] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” in *Network and Distributed System Security Symposium (NDSS)*, 2008.

[29] P. Saxena, S. Hanna, P. Poosankam, and D. Song, “Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications.” in *NDSS*, 2010.

[30] S. Lekies, B. Stock, and M. Johns, “25 million flows later: Large-scale detection of DOM-based XSS,” in *ACM Conference on Computer and Communications Security*, 2013, pp. 1193–1204.

[31] A. Zeller, “Isolating cause-effect chains from computer programs,” in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. ACM, 2002, pp. 1–10.

[32] M. Renieres and S. P. Reiss, “Fault localization with nearest neighbor queries,” in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 2003, pp. 30–39.

[33] T. Janssen, R. Abreu, and A. J. van Gemund, “Zoltar: A toolset for automatic fault localization,” in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 662–664.

[34] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, “Fault localization using execution slices and dataflow tests.” in *ISSRE*, vol. 95, 1995, pp. 143–151.

[35] M. Weiser, “Program slicing,” in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[36] H. Agrawal and J. R. Horgan, “Dynamic program slicing,” in *ACM SIGPLAN Notices*, vol. 25, no. 6. ACM, 1990, pp. 246–256.

[37] N. Gupta, H. He, X. Zhang, and R. Gupta, “Locating faulty code using failure-inducing chops,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 263–272.

[38] M. Burger and A. Zeller, “Minimizing reproduction of software failures,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 221–231.

[39] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel, “On the use of delta debugging to reduce recordings and facilitate debugging of web applications,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 333–344.

[40] M. Jose and R. Majumdar, “Cause clue clauses: error localization using maximum satisfiability,” *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 437–446, 2011.

[41] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, “Efficient unit test case minimization,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 417–420.

[42] Y. Lei and J. H. Andrews, “Minimization of randomized unit test cases,” in *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE, 2005, pp. 10–pp.

[43] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, “Cause reduction for quick testing,” in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 243–252.

[44] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, “Evaluating non-adequate test-case reduction,” in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 16–26.

[45] M. Höschele and A. Zeller, “Mining input grammars from dynamic taints,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 720–725.

[46] O. Bastani, R. Sharma, A. Aiken, and P. Liang, “Synthesizing program input grammars,” in *PLDI*, 2017.