Fuzz Testing and Test Case Reduction

Summary of the Ph.D. Dissertation

by

Renáta Hodován

Supervisor: Dr. Tibor Gyimóthy, Professor

Submitted to the

Ph.D. School of Computer Science

Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged, 2019

Introduction

As technology continues to evolve and change, software systems encompass almost every aspect of our life. They are present in our homes, they are used at work, they entertain kids, they are used by doctors for operations, they manage production lines, they predict earthquakes, and all this is just the tip of the iceberg. While these systems make our lives easier, they also act as a potential threat, since they possess or access sensitive information about us. Therefore, security testing has an outstanding importance both to users and vendors of software systems.

When it comes to security testing, the concept of random test generation or fuzz testing [17] turns up quickly, too. Fuzz testing is a popular and automatic negative testing approach, which is based on the idea of generating large numbers of random or semi-random test cases and feeding them to the software under testing (or SUT for short), hoping to trigger some unexpected behaviour. Fuzzing can be used as a standalone testing method, but it can also serve as an exploratory step looking for potential entry points of an actual exploit. Its popularity is in great part due to its effectiveness in finding real and reproducible bugs while using fully automated frameworks without the need for professional attendance.

A fuzzer framework usually consists of three main parts: a test generator providing the test cases, a SUT executor functionality which executes and monitors the tested software, and an optional test case reducer, which finds a minimal subset of the issue triggering test case to narrow down the reason of the failure. In this thesis, I focus on the test generator and the test case reducer components of such a framework.

As my primary fuzzing target, I choose JavaScript engines. JavaScript is the de facto standard programming language of web browsers, which became the most popular programming language of the world in the last decade [22, 3]. Because of the prevalence of the language, ensuring the correctness of its execution engines – both functionally and security-wise – is of paramount importance. In the first part of this thesis, I introduce a novel fuzzing approach that targets JavaScript engines and has achieved significantly better code coverage than a state-of-the-art JavaScript fuzzing solution, and a prototype implementation that has found more than 100 unique issues in various JavaScript engines to date.

Having an effective random test generator helps us in finding bugs and issue triggering reproduction cases. However, usually only a small portion of these findings are required to reproduce the failure. Throwing away the unnecessary parts helps better understand the problem – which is useful, e.g., for fixing it as soon as possible. Finding that small portion can be really challenging though, even in real life scenarios, not to mention the labor of locating it in a fuzzer generated test case. To lower the amount of work needed for test case minimization, multiple automated solutions have evolved in the last decades.

In the second part of this thesis, I investigate the two most widespread reduction algorithms, pinpoint their weaknesses, and propose improvements to them. These improvements got implemented into two open-source tools that were used to evaluate our proposals in practice. The results show that our solutions produce smaller reproduction cases in much less time compared to the baseline approaches.

The additional third part focuses on the practical aspects of my research. I will introduce Fuzzinator, an open-source fuzzer framework that was used during the experimental evaluations of this dissertation and which assisted me in finding and reporting more than 1000 issues in various projects.

In the thesis, five main results are stated which are listed below:

1. Prototype Graph-based fuzzing of JavaScript engines
2. Decomposition of the Delta Debugging algorithm
3. Analysis of the effects of different parser grammars on Hierarchical Delta Debugging
4. Improvements to the Hierarchical Delta Debugging with tree transformations
5. Coarse Hierarchical Delta Debugging

Fuzz Testing of JavaScript Engines

Fuzzing, or random testing is a popular technique as it promises the creation of a large number of test cases with limited effort. Moreover, as a result of randomness, it is often capable of generating extreme test cases that are easily overlooked by a human test engineer.

In this thesis, we focused on the fuzz testing of JavaScript engines, with special interest on the type API that the engines expose to their users.

1. Prototype Graph-based Fuzzing of JavaScript Engines

First, we defined a model, titled the Prototype Graph, that is able to describe the object-based type system of the JavaScript language.

A Prototype Graph is a collection of *type* and *signature* vertices connected by six different kinds of edges: *property*, *proto*, *call*, *construct*, *parameter*, and *return* edges. *Type* vertices represent JavaScript 'types', i.e., categories of similar objects, while *signature* vertices give information about the possible signatures of callable types, i.e., functions or constructors. *Proto* and *property* edges connect *type* vertices, while the others connect *type* and *signature* vertices in one direction or the other. Member name information is encoded in the label of *property* edges while the order of function arguments are stored in the label of *parameter* edges.

Given a built graph, our goal from fuzzing perspective is to generate call expressions that invoke functions on API objects with type-correct arguments. That can be achieved by a random walk on the Prototype Graph: "First walk forward on *proto*, *property*, *construct*, *call*, and *return* edges to a *signature* vertex, then walk backward on *parameter* and *proto* edges, and so on..." The formal definition of the Prototype Graph as well as the formalism of the random walk on it is available at Chapter 4 of the dissertation.

As an example, we created a Prototype Graph from a portion of the ECMAScript 5.1 standard. The created graph is presented in Figure 1, where the large and the small nodes represent *type* and *signature* vertices respectively. The single black node on the left represents the type of the global object, however, that is for identification and presentation purposes only. Thick lines with labels represent *property* edges, thin lines with hollow arrows represent *proto* edges, while dashed lines with double-bracketed labels represent *construct*, *call*, *parameter*, and *return* edges.

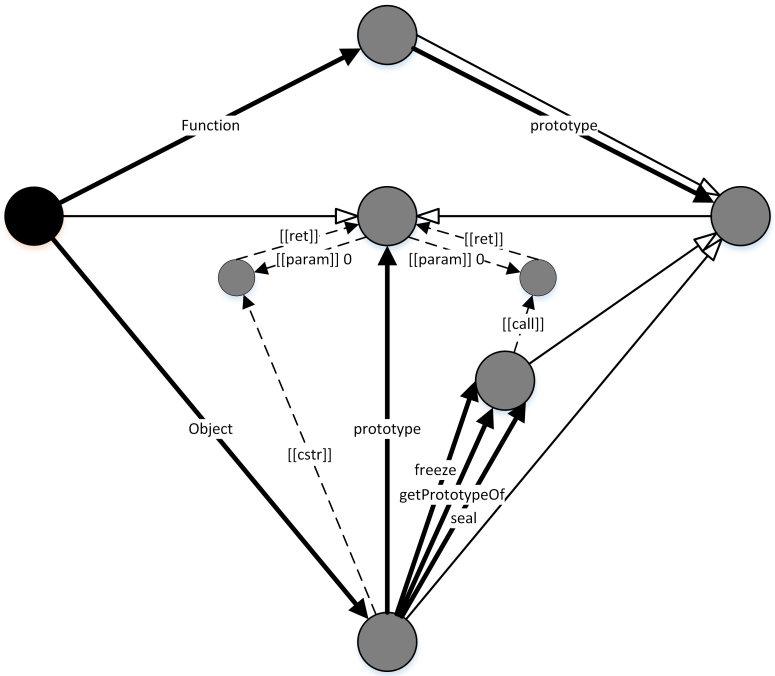

Figure 1: Example prototype graph manually constructed based on a portion of the ECMAScript 5.1 standard [2, Sections 15.2,15.3].

The following two expressions exemplify the test generation ability of our approach with using the manually created graph above as input:

- `this.Object.getPrototypeOf(this.Function.prototype)`,
- `new (this.Object)(this)`.

To avoid the need of building such graphs manually, we outlined and implemented two automated techniques: the engine *discovery* and the *learn* from existing test cases. Both of them rely on the introspection capabilities of the JavaScript language: not only can we determine the actual type of every expression at runtime but we can enumerate all properties and walk the property chains of all objects, and retrieve the number of the formal parameters of every function.

As evaluation target, we have chosen JavaScriptCore (or jsc for short), the command line JavaScript execution tool from the WebKit [1] project (which is the rendering engine working under the hood of the widespread Apple Safari web browser and was also the ancestor of the Blink engine of the Chromium browser). Figure 2 shows the Prototype Graph of jsc resulted by the automatic extraction.

As baseline fuzzer, we used the jsfunfuzz [21] open-source tool, which is used and developed by Mozilla and which has found hundreds of issues in SpiderMonkey, the JavaScript engine of Firefox [20]. With all fuzzers, we generated 50.000

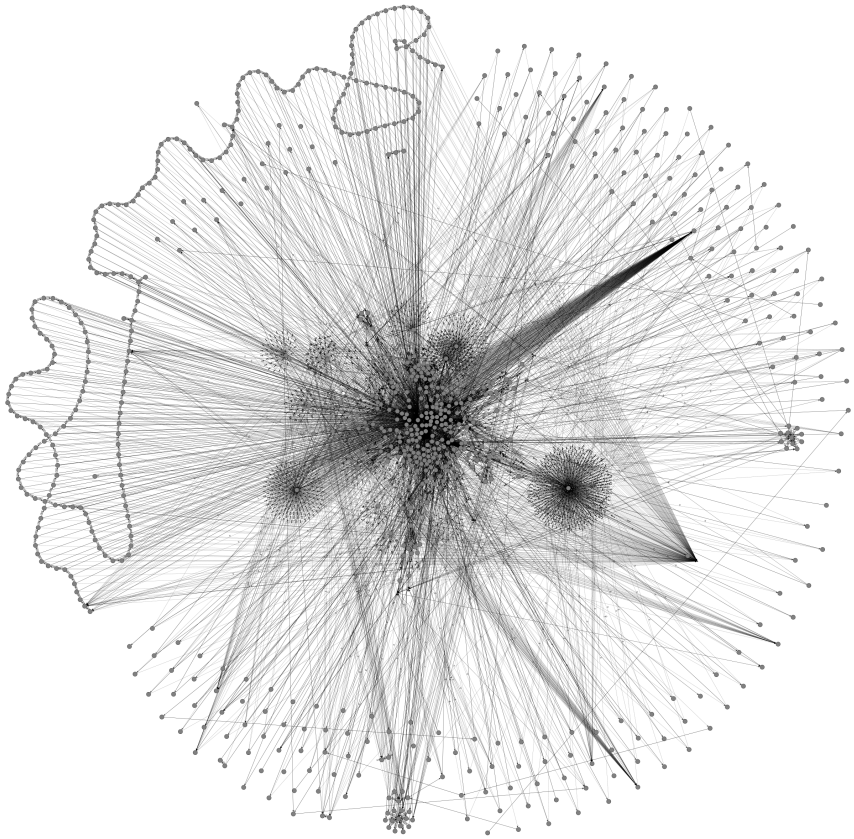

Figure 2: Prototype graph built for *jsc* with the signature learning technique.

JavaScript expressions, executed them with jsc, and compared the results both coverage-wise and from the perspective of the amount of found bugs.

Table 1 shows the line coverage results of all three fuzzing approaches both with module-level granularity and in total. The results show that the basic engine discovery-based approach does not perform as well as jsfunfuzz in terms of total code coverage (23.31% compared to 37.25%), but as soon as we extend our graph with signature information extracted from tests, it improves significantly and gives higher results (44.13%).

We should also highlight results on three important modules, namely on *runtime*, *yarr*, and *jsc.cpp*. The first contains C++ implementations of core JavaScript language functionality (like built-in functions), while the second contains the regular expression engine of the project. The third module (actually, a single file), is the main command line application, which is a classic JavaScript engine embedder in the sense that it binds some extra, non-standard routines into the JavaScript

Table 1: Code coverage results on *jsc* after 50,000 generated expressions (after 50,320 expressions for jsfunfuzz).

Module	**Total Lines**	**Covered Lines**					
		pgdiscover		**pglearn**		**jsfunfuzz**	
API	1698	9	0.53%	9	0.53%	9	0.53%
DerivedSources	4546	148	3.26%	167	3.67%	312	6.86%
assembler	2997	1046	34.90%	2037	67.97%	2054	68.54%
bindings	165	0	0.00%	0	0.00%	0	0.00%
builtins	96	63	65.63%	63	65.63%	62	64.58%
bytecode	8578	1650	19.24%	4196	48.92%	3320	38.70%
bytecompiler	4656	2344	50.34%	2372	50.95%	2887	62.01%
debugger	713	3	0.42%	3	0.42%	3	0.42%
dfg	29959	27	0.09%	11019	36.78%	9403	31.39%
disassembler	1033	3	0.29%	3	0.29%	3	0.29%
heap	4221	2517	59.63%	2671	63.28%	2373	56.22%
inspector	3594	0	0.00%	0	0.00%	0	0.00%
interpreter	1336	594	44.46%	664	49.70%	648	48.50%
jit	8919	814	9.13%	4852	54.40%	4345	48.72%
jsc.cpp	926	507	54.75%	519	56.05%	240	25.92%
llint	840	344	40.95%	424	50.48%	451	53.69%
parser	6586	3618	54.93%	3801	57.71%	4400	66.81%
profiler	788	4	0.51%	4	0.51%	4	0.51%
runtime	27112	12115	44.69%	15101	55.70%	10648	39.27%
tools	534	13	2.43%	13	2.43%	13	2.43%
yarr	3538	486	13.74%	1879	53.11%	856	24.19%
TOTAL	112835	26305	23.31%	49797	44.13%	42031	37.25%

environment. That is, these are the modules that expose API of native code to the JavaScript space and thus are in our focus. As the table shows, even the simpler engine discovery-based technique can outperform jsfunfuzz in two out of the three modules, and the signature-extended variant gives the best results in all three cases.

As the ultimate goal of fuzzing is not only to reach good code coverage but also to cause system malfunction, we compared the three techniques on the basis of caused crashes as well. Table 2 shows the total number of observed failures, and since several tests triggered the same problem, the number of unique failures as well. Interestingly, both graph-based fuzzing techniques found the same failures. This also means that even the engine discovery technique with lower total coverage ratio could find more errors than jsfunfuzz.

Table 2: Number of failures caused in *jsc*.

	pgdiscover	**pglearn**	**jsfunfuzz**
total failures	1326	1445	4
unique failures	6	6	2

Main Results and Own Contribution

1. Prototype Graph-based Fuzzing of JavaScript Engines

The author, jointly with her co-author, defined and formalised a graph-based model, named the Prototype Graph, to represent the type system of arbitrary JavaScript engines. The model is able to describe the object-based type system of the JavaScript language including the expected signature of the methods and constructors provided by the underlying JavaScript engine. The author invented how the introduced model can be automatically extracted from any JavaScript engine and how it can be extended by analysing existing test cases.

With the help of this model, the author designed and formalized an algorithm to generate random JavaScript expressions from it and stress-test the underlying JavaScript engine. She implemented a prototype tool which is able to build engine-specific models, to extend them with information extracted from existing test cases, and to use the model for fuzz testing real life JavaScript engines. Based on the prototype tool, she evaluated and compared it to another state-of-the-art JavaScript fuzzer and reported the found issues.

Automated Test Case Reduction

It is considered a common knowledge that there is no such thing as a bug-free program. Bugs – and bug-inducing test cases – can be uncovered by users, by developers, or by various automated testing frameworks. However, usually only a small portion of these findings are required to reproduce the failure. Throwing away the unnecessary parts helps better understand the problem – which is useful, e.g., for fixing it as soon as possible. Finding that small portion can be really challenging though, even in real life scenarios, not to mention the labor of locating it in a fuzzer-generated test case. This fact motivates researchers to find effective automated methods to perform these reductions as fast as possible, while getting output as small as possible. Among these approaches, the most well-known solutions are the syntax-unaware approach of Zeller and Hildebrandt named Delta Debugging [24, 4, 25], and its syntax-aware counterpart called Hierarchical Delta Debugging [18].

2. Decomposition of the Delta Debugging Algorithm

Definition 1 shows Zeller and Hildebrandt's latest formulation of the minimizing Delta Debugging [25].

Definition 1 (Zeller and Hildebrandt's)
Let *test* and $c_{✗}$ be given such that $test(\emptyset) = ✓ \wedge test(c_{✗}) = ✗$ hold. The goal is to find $c'_{✗} = ddmin(c_{✗})$ such that $c'_{✗} \subseteq c_{✗}$, $test(c'_{✗}) = ✗$, and $c'_{✗}$ is 1-minimal. The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{✗}) = ddmin_2(c_{✗}, 2) \text{ where}$$

$$ddmin_2(c'_{✗}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = ✗ \text{ (“reduce to subset”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = ✗ \text{ (“reduce to complement”)} \\ ddmin_2(c'_{✗}, \min(|c'_{✗}|, 2n)) & \text{else if } n < |c'_{✗}| \text{ (“increase granularity”)} \\ c'_{✗} & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_{✗} - \Delta_i$, $c'_{✗} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{✗}|/n$ holds. The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{✗}) = ✗ \wedge n \leq |c'_{✗}|$.

The first thing to notice is that although the implementations tend to use sequential loops to realize the “reduce to subset” and “reduce to complement” cases of $ddmin_2$, the potential for parallelization is there in the original formulation, since $\exists i \in \{1, \ldots, n\}$ does not specify how to find that existing i. Since n can grow big for real inputs and *test* is often expected to be an expensive operation, we

propose to make use of the parallelization potential and rewrite $ddmin_2$ to use parallel loops.

Furthermore, we can observe that although $ddmin_2$ seems to be given with a piecewise definition, the pieces are actually not independent but are to be considered one after the other, as mandated by the *else if* phrases. However, we can also observe that this sequentiality is *not* necessary. There may be several Δ_i and ∇_i test cases that induce the original failure, we may choose any of them (i.e., we do not have to prefer subsets over complements) and we will still reach a 1-minimal solution at the end.

However, we cannot benefit from this observation as long as our implementation is sequential. Therefore we propose to combine the two reduce cases, and test all subsets and complements in one step when parallelization is available. This way the algorithm does not have to wait until all subset tests finish but can start testing the complements as soon as computation cores become available. Listing 1 shows the pseudo-code of the algorithm variant with all the aforementioned improvements.

We can also observe that the "reduce to subset" case is not even necessary for 1-minimality. It is a greedy attempt by the algorithm to achieve a significant reduction of the test case by removing all but one subsets in one step rather than removing them one by one in the "reduce to complement" case. However, there are several input formats where attempting to keep just the "middle" of a test case almost always gives a syntactically invalid input and thus cannot induce the original failure. For such input formats, the "reduce to complement" case may occur significantly more often, while the "reduce to subset" case perhaps not at all. Thus, we argue that it is worth experimenting with the reordering of the reduce cases, and also with the complete omission of the "reduce to subset" case, as it may be simply the waste of computation resources.

During the investigation of the minimizing Delta Debugging algorithm, we created a prototype tool [8] that implemented our proposed improvements. We presented an experiment conducted on 4 artificial test cases and on 2 wide-spread browser engines with 3 real test cases each. All test cases were minimized with several algorithm variants and with 12 different levels of parallelization (ranging from single-core execution – i.e., no parallelization – to 64-fold parallelization). The results of the 1098 successfully executed test case minimizations prove that all improvements to the Delta Debugging algorithm presented in the dissertation achieved performance improvements, with best variants reducing the running time on real test cases significantly, by cca. 75–80%.

Listing 1: Parallel variant of *ddmin*$_2$ with combined reduce cases

```
1  procedure ddmin_2^κ(c'_✗, n)
2  begin
3    while true do begin
4      (* reduce to subset or complement *)
5      found = 0
6      parallel forall i in 1..2n do
7        if 1 ≤ i ≤ n then
8          if test(Δ_i^(c'_✗,n)) = ✗ then begin
9            found = i
10           parallel break
11         end
12       else if n + 1 ≤ i ≤ 2n then
13         if test(∇_{i−n}^(c'_✗,n)) = ✗ then begin
14           found = i
15           parallel break
16         end
17     if 1 ≤ found ≤ n then begin
18       c'_✗ = Δ_found^(c'_✗,n)
19       n = 2
20       continue
21     end else if n + 1 ≤ found ≤ 2n then begin
22       c'_✗ = ∇_{found−n}^(c'_✗,n)
23       n = max(n − 1, 2)
24       continue
25     end
26     (* increase granularity *)
27     if n < |c'_✗| then begin
28       n = min(|c'_✗|, 2n)
29       continue
30     end
31     (* done *)
32     break
33   end
34   return c'_✗
35 end
```

3. Analysis of the Effects of Different Parser Grammars on Hierarchical Delta Debugging

Another popular reduction approach is Hierarchical Delta Debugging [18, 19] (or HDD) that applies the aforementioned DD algorithm to the levels of parse trees.

As even the original authors of HDD noted, standard context-free grammars use recursion to represent lists, which yields heavily unbalanced trees. This property does not only increase the number of test executions in HDD but also has an effect on the size of the reduced test case.

We argue that instead of standard context-free grammars, *extended* context-free grammars (ECFGs) should be used for parsing inputs and building the input trees of HDD. Extended context-free grammars allow the right-hand side of rules to be regular expressions over terminals and non-terminals, i.e., alternation operators, groupings, and quantifiers (?, *, +) may appear. While extended context-free grammars describe exactly the context-free languages, just like standard context-free grammars do, the quantifiers allow the omission of recursive rules for list-like structures, and this results in much better balanced parse trees.

To evaluate the effect of ECFG-based tree building on HDD, we have implemented the above outlined ideas. Experimental evaluation of the prototype tool [9] supports the grammar-related ideas outlined in this part: its reduced outputs are significantly smaller (by cca. 25–40%) on the investigated test cases than those produced by the original HDD implementation using standard context-free grammars.

4. Improvements to the Hierarchical Delta Debugging with Tree Transformations

Even with the use of extended context-free grammars, there remain cases where HDD performs suboptimally. One of the possible reasons is that parse trees can contain linear components: paths where each node has at most one child. Such linear components cause unnecessary test attempts: if HDD already visited the topmost node of a linear component at a given level and DD decided to keep it, then it will not decide otherwise for the rest of the nodes of the subgraph either as it progresses to the next levels. If the minimal replacement strings of the nodes are identical, then the opposite is true as well: marking any node in the subgraph as removed (i.e., replacing it with its smallest allowable syntactic replacement [19]) will yield the same output. Thus, collapsing such lines into a single node can squeeze the height of the tree and reduce the number of test steps.

The optimization, which can be applied to the root of the tree before HDD is actually invoked, is formalized as algorithm *squeezeTree* in Listing 2.

Another interesting thing we can notice while analyzing the HDD-reduced trees, is that even though some of the nodes are marked as removed, they show up in the output nevertheless. The reason behind it is that the smallest allowable syntactic replacement computed for some tokens from the grammar is identical to the occurrence of the token in the input. The consequence is that no matter

Listing 2: The "Squeeze Tree" Algorithm

```
1  procedure squeezeTree(node)
2  begin
3    if not isToken(node) then begin
4      forall i in 1..children(node) do
5        child(node, i) ← squeezeTree(child(node, i))
6      if children(node) = 1 and Φ(node) = Φ(child(node, 1)) then
7        return child(node, 1)
8    end
9    return node
10 end
```

Listing 3: The "Hide Unremovable Tokens" Algorithm

```
1  procedure hideUnremovableTokens(node)
2  begin
3    if isToken(node) then
4      if text(node) = Φ(node) then
5        markAsRemoved(node)
6    else
7      forall i in 1..children(node) do
8        hideUnremovableTokens(child(node, i))
9  end
```

how DD decides, whether to keep them (in which case the actual token text will be used) or not (in which case the replacement will contribute to the output), it will yield the same test case. But DD will try to keep and remove them anyway, leading to superfluous test attempts.

However, if we already marked such "unremovable" tokens as removed in a preprocessing step, before applying HDD to the tree, then they will be hidden from *ddmin*. The optimization is formalized as the algorithm *hideUnremovableTokens* in Listing 3.

These ideas were added to our aforementioned prototype tool. According to our evaluation, these optimizations together caused more than 5-fold speedup of the minimization process in the best case.

5. Coarse Hierarchical Delta Debugging

From test case reduction perspective, it's obvious, that the biggest gain is from the complete removal of subtrees, while other parts of the parse tree, which are not

Listing 4: The Coarse HDD Algorithm

```
1  procedure coarseHDD(input_tree)
2      level ← 0
3      nodes ← tagNodes(input_tree, level)
4      while nodes ≠ ∅ do
5          nodes ← filterEmptyPhiNodes(nodes)
6          if nodes ≠ ∅ then
7              minconfig ← ddmin(nodes)
8              prune(input_tree, level, minconfig)
9          end if
10         level ← level + 1
11         nodes ← tagNodes(input_tree, level)
12     end while
13 end procedure
```

allowed by syntax to disappear, deserve less attention. Therefore, we have created a variant of HDD called Coarse Hierarchical Delta Debugging, which visits all levels of the tree one by one like the original HDD but filters out those nodes from the input configuration of DD that have a non-empty replacement fragment. In the case when a level has no node with empty replacement string, the algorithm performs no testing steps related to that level and leaves the tree completely unchanged. The pseudocode of the algorithm is given in Listing 4: all auxiliary routines are the same as explained in the original publication of HDD [18], except for *filterEmptyPhiNodes*, which performs the above described filtering.

Since the Coarse HDD algorithm only considers the nodes with empty replacement strings, the effectiveness of this approach is highly dependent on the grammar that creates the parse tree. Trees that are created by grammars expressing repetitions with recursive expressions instead of quantifiers contain much less completely removable subtrees, lowering the performance of Coarse HDD. One way of solving the problem would be to manually rewrite the grammar to use quantified constructs extensively. However, this would require manual labor for every grammar that is used for reduction. The other way is to apply automated transformation to the input trees to convert them into a form as if they were parsed by non-recursive grammars.

This algorithm looks for left and right-recursive tree constructs and flattens these "list elements". Such a transformation yields trees as if lists were not parsed by recursive rules but by quantified expressions. The idea of this transformation is formalized as the Flatten Tree Recursion algorithm in Listing 5, where *emptyPhiNode* creates a new tree node with its arguments as its children and defines the

Listing 5: The Flatten Tree Recursion Algorithm

```
1  procedure flattenTreeRecursion(node)
2      forall child in children(node) do
3          flattenTreeRecursion(child)
4      end forall
5      num ← |children(node)|
6      if num > 1 then
7          if name(children(node)[1]) = name(node) then
8              left ← children(node)[1]
9              right ← children(node)[2..num]
10             children(node) ← children(left) + emptyPhiNode(right)
11         elif name(children(node)[num]) = name(node) then
12             left ← children(node)[1..num−1]
13             right ← children(node)[num]
14             children(node) ← emptyPhiNode(left) + children(right)
15         end if
16     end if
17 end procedure
```

empty string as the smallest allowable syntactic replacement for the newly created node.

To see what we really gain – and what we lose – we have evaluated an implementation of the algorithms on various test cases. The two key algorithms presented in this thesis, Coarse HDD* and Flatten Tree Recursion have achieved their goals together: they have reduced test cases with 58% fewer steps, on average, with a maximum gain of 79%. For our largest test case, this has meant several hours faster test case reduction. The price of the speed-up, in the worst case, was an increase in the output by 0.36% of the input size.

Main Results and Own Contribution

2. Decomposition of the Delta Debugging Algorithm

The author has analysed the widespread Delta Debugging algorithm, with special focus on its *ddmin* variant. She recognised, that the two components of *ddmin*, i.e., the subset and complement-based reduction parts, are not necessary sequential: they can be reordered and one of them is even avoidable while still ensuring the original 1-minimality of the result. Additionally, she has also shown that the ordering of the test executions is irrelevant from the perspective of 1-minimality. This observation has led the author to define a parallel version of the *ddmin* algorithm.

The author, jointly with her co-author, created a clean room implementation of *ddmin* which incorporates the aforementioned findings. Using this prototype tool, the author performed an extensive evaluation to empirically analyse the effectiveness of the approach on artificial and real-life examples.

3. Analysis of the Effects of Different Parser Grammars on Hierarchical Delta Debugging

The author investigated the syntax-aware version of Delta Debugging, named Hierarchical Delta Debugging, which applies *ddmin* to the levels of parse trees. She recognised, that using trees built by standard context-free grammars is not optimal since they can be heavily unbalanced due to recursive rules. She proposed to use extended context-free grammars to build the input trees and designed an improved algorithm for the minimal replacement string calculation. The author and her co-author created a prototype implementation and used it to compare the result of Hierarchical Delta Debugging on standard and extended context-free grammar-based trees. Using this tool, the author performed the experimental evaluation of the proposed idea.

4. Improvements to the Hierarchical Delta Debugging with Tree Transformations

The author has recognised that some constructs in the parse tree trigger superfluous test executions. She identified two such constructs and suggested to apply two preprocessing algorithms to the trees before the actual reduction: "tree squeezing" and the "hiding of unremovable tokens". The author, jointly with her co-authors, implemented these algorithms to the aforementioned prototype tool and evaluated their effects.

5. Coarse Hierarchical Delta Debugging

The author and her co-authors have observed that not all nodes of parse trees contribute equally to the process of reduction. They recognised that the most gain comes from the removal of subtrees that can be actually deleted from the output (i.e., syntax does not require some replacement string). Based on this observation, the author created the coarse version of Hierarchical Delta Debugging, which only considers such nodes of parse trees during reduction and she designed the algorithm of tree flattening. She implemented this variant into the aforementioned prototype tool and evaluated its effect on real life reduction tasks.

Summary

Our contributions can be grouped into five major thesis points which cover two areas of software testing, namely fuzz testing and automated test case reduction.

The main result of the first area and first thesis point is the definition of a novel representation model to describe the type system of the JavaScript language that was shown to be effective for fuzz testing.

To the area of automated test case reduction belong four main contributions, organized into four thesis points. First, we have investigated the widespread, syntax-unaware, minimizing Delta Debugging algorithm and showed that its test executions can be parallelized. Furthermore, we have noticed that its components can be reordered and one of them is even avoidable. We have proved that these changes do not harm the guarantee of 1-minimality. Next, we turned to a the state-of-the-art syntax-aware reduction approach, Hierarchical Delta Debugging, and evaluated the effect of various input trees to the algorithm. As a result, we have shown that extended context-free grammars perform much better both from the perspective of performance and that of the output size. In the third thesis point, we have introduced two tree transformation algorithms, "tree squeezing" and the "hide of unremovable tokens", that caused significant performance improvement when applied as a preprocessing steps. Finally, we have introduced the Coarse variant of Hierarchical Delta Debugging that intended to trade the guarantee of 1-minimality for performance improvement. Beside the modified version of the HDD algorithm, we proposed another tree-transformation, named "tree flattening", which helped exploit the strength of Coarse HDD in a more efficient way. As a result, Coarse HDD fulfilled our expectations: using it together with the "tree-flattening" step achieved significant performance improvement while the growth of the output test was negligible.

We have developed a fuzzer framework, named Fuzzinator [13, 5], that can integrate all the aforementioned ideas and others, too. Fuzzinator was used for the practical evaluation presented in the dissertation and it is actively used for years to test real applications. Beside the aforementioned JavaScript fuzzer, we have integrated several other generators into Fuzzinator like our own Grammarinator [16, 7] and Generinator:RATS [6] tools or the variants of the popular American Fuzzy Lop (AFL) [23]. Altogether, we have discovered, reduced and reported more than 1000 issues in various projects during the years of writing this dissertation.

Lastly, Table 3 summarizes which publications cover which results of the thesis.

	[10]	[12]	[11]	[15]	[14]
1.	•				
2.		•			
3.			•		
4.				•	
5.					•

Table 3: Relation between the main results of the thesis and the corresponding publications.

Acknowledgements

Getting here has been a long journey, but by no means a lonely one. Many people helped me along the way and I will always be grateful to them. Chronologically first, I would like to say thank you to my high school teachers, Zsuzsanna Bali and Imre Szőke, who persuaded me that girls could also become computer scientists. I am thankful to Tibor Gyimóthy who invited me to join the Department of Software Engineering many years ago and later became my Ph.D. supervisor. I am grateful to him and to Ákos Kiss for their continuous support, especially after I left the safe harbour by choosing a topic that has not had a long research history at our department. I am also thankful to Gábor Lóki and Zoltán Herczeg for putting my research into practice. Last but not least, I would like to thank my family that they provided a pleasant background to my studies throughout the years and also encouraged me to go on with my research.

Bibliography

[1] Apple Inc. WebKit. A fast, open source web browser engine. `https://webkit.org/`. [Accessed: 2019-03-01].

[2] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1 edition, 2011.

[3] GitHub Inc. Top languages over time. `https://octoverse.github.com/projects#languages`. [Accessed: 2019-03-01].

[4] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 135–145. ACM, 2000.

[5] Renáta Hodován and Ákos Kiss. Fuzzinator. Random Testing Framework. `https://github.com/renatahodovan/fuzzinator`. [Accessed: 2019-03-01].

[6] Renáta Hodován and Ákos Kiss. Generinator: Random Attributes, Tags & Style. `https://github.com/renatahodovan/generinator-rats`. [Accessed: 2019-03-01].

[7] Renáta Hodován and Ákos Kiss. Grammarinator. ANTLR v4 grammar-based test generator. `https://github.com/renatahodovan/grammarinator`. [Accessed: 2019-03-01].

[8] Renáta Hodován and Ákos Kiss. Picire: Parallel Delta Debugging Framework. `https://github.com/renatahodovan/picire`. [Accessed: 2019-03-01].

[9] Renáta Hodován and Ákos Kiss. Picireny: Hierarchical Delta Debugging Framework. `https://github.com/renatahodovan/picireny`. [Accessed: 2019-03-01].

[10] Renáta Hodován and Ákos Kiss. Fuzzing JavaScript Engine APIs. In *Integrated Formal Methods – 12th International Conference, iFM 2016, Reykjavík, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science (LNCS)*, pages 425–438. Springer, 2016.

[11] Renáta Hodován and Ákos Kiss. Modernizing Hierarchical Delta Debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, pages 31–37. ACM, 2016.

[12] Renáta Hodován and Ákos Kiss. Practical Improvements to the Minimizing Delta Debugging Algorithm. In *Proceedings of the 11th International*

Joint Conference on Software Technologies, ICSOFT 2016, pages 241–248. SciTePress, 2016.

[13] Renáta Hodován and Ákos Kiss. Fuzzinator: An Open-Source Modular Random Testing Framework. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, ICST 2018, pages 416–421, April 2018.

[14] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Coarse Hierarchical Delta Debugging. In *In Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*, ICSME 2017, pages 194–203. IEEE Computer Society, 2017.

[15] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging. In *Proceedings of the 12th IEEE/ACM International Workshop on Automation of Software Testing*, AST 2017, pages 23–29. IEEE Computer Society, 2017.

[16] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: A grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, pages 45–48. ACM, 2018.

[17] Barton Miller. Foreword for Fuzz Testing Book. `http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html`, 2008. [Accessed: 2019-03-01].

[18] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 142–151. ACM, 2006.

[19] Ghassan Shakib Misherghi. Hierarchical delta debugging. Master's thesis, University of California, Davis, 2007.

[20] Mozilla Foundation. The new Firefox. Fast for good. `https://www.mozilla.org/en-US/firefox/new/`. [Accessed: 2019-03-01].

[21] Mozilla Security. Javascript engine fuzzers. `https://github.com/MozillaSecurity/funfuzz/`. [Accessed: 2019-03-01].

[22] Stack Exchange Inc. Most popular technologies, programming, scripting, and markup languages. `https://insights.stackoverflow.com/survey/2018#technology`. [Accessed: 2019-03-01].

[23] Michał Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. [Accessed: 2019-03-01].

[24] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '99)*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer-Verlag, 1999.

[25] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.