Binary Reduction of Dependency Graphs

Christian Gram Kalhauge
kalhauge@cs.ucla.edu
Computer Science Department
University of California, Los Angeles (UCLA)
Los Angeles, CA, USA

Jens Palsberg
palsberg@ucla.edu
Computer Science Department
University of California, Los Angeles (UCLA)
Los Angeles, CA, USA

ABSTRACT

Delta debugging is a technique for reducing a failure-inducing input to a small input that reveals the cause of the failure. This has been successful for a wide variety of inputs including C programs, XML data, and thread schedules. However, for input that has many internal dependencies, delta debugging scales poorly. Such input includes C#, Java, and Java bytecode and they have presented a major challenge for input reduction until now. In this paper, we show that the core challenge is a reduction problem for dependency graphs, and we present a general strategy for reducing such graphs. We combine this with a novel algorithm for reduction called Binary Reduction in a tool called J-Reduce for Java bytecode. Our experiments show that our tool is 12x faster and achieves more reduction than delta debugging on average. This enabled us to create and submit short bug reports for three Java bytecode decompilers.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Debugging, dependencies, reduction

ACM Reference Format:
Christian Gram Kalhauge and Jens Palsberg. 2019. Binary Reduction of Dependency Graphs. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338956

1 INTRODUCTION

Delta debugging automates a process that programmers otherwise do by hand. When a program crashes on an input, the programmer tries to understand the cause of the crash by *reducing* the input. Intuitively, the programmer can cut the input in half and see if one of the two halves causes the crash as well. After some repetitions of this step, the input may be small enough for the programmer to spot the cause of the problem. Delta debugging executes a more advanced version of this, automatically. For example, delta debugging can map the original input to a nonconsecutive subsequence. Thus, delta debugging relieves programmers from the tedium of reducing and executing, and lets them focus on improving their programs.

In their seminal paper on delta debugging, Zeller and Hildebrandt [27] showed successful experiments in which the inputs were C programs, Mozilla user actions, and UNIX commands. Other papers have reported on experiments with XML data [19], thread schedules [5], and event sequences [11]. The problem of reducing failure-inducing input to a minimal size is NP-complete [19], and for an input with n *characters*, trying all 2^n substrings may be futile. Instead, the delta debugging algorithm ddmin [27] tries $O(n^2)$ substrings. This led to massive success but when most natural subsets of the input are invalid, most iterations of ddmin fail and are of no help towards reduction. As a step towards scalability, Zeller and Hildebrandt showed how ddmin does better when applied to a list of *lines*. This is better than a character-oriented approach because often a line of code represents a syntactic element such as a statement. Misherghi and Su [19] went further and introduced hierarchical delta debugging (HDD) that works with the syntactic structure of the data. For example, for reduction of a method body, HDD represents the body as a list of *statements* and runs ddmin on the list. This is better than a line-oriented approach because a statement can span multiple lines. Use of the syntactic structure ensures that each input is syntactically valid and increases the chance that each run produces useful information.

In this paper we consider the next level of difficulty, which arises when elements of the syntactic structure have many *internal dependencies*. Such input includes C#, Java, and Java bytecode, where a class may depend on other classes and where compilation and bytecode verification require all dependencies to be present. We can represent such a program as a list of classes and run ddmin on the list, yet most runs will fail because the input is invalid. We solve this by modeling the internal dependencies in the input as a *dependency graph* and then running reduction on a list of transitive closures in the dependency graph. We will show experiments with reduction by both ddmin and a novel algorithm called Binary Reduction.

In the remainder of the paper, Section 2 introduces the challenge in detail, after which Sections 3–6 present our contributions:

- We show that dependency graphs are a convenient data structure for reduction, particularly by ddmin (Section 3).
- We present a new reduction algorithm, called Binary Reduction that runs only $O(n \log n)$ iterations (Section 4).
- We evaluate on 238 Java bytecode programs that induce failures in three decompilers. Binary Reduction on graphs is 12x faster and reduces more than ddmin (Section 5).
- We submitted bug reports for the decompilers (Section 6).

Finally, Section 7 discusses related work, and Section 8 concludes.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338956

2 THE CHALLENGE

We will explain the challenge of reducing input with dependencies via an example. The example concerns the Java bytecode decompiler called CFR (http://www.benf.org/other/cfr/). CFR takes as input a valid Java bytecode program and decompiles it to a Java source program. This is useful for programmers who want to inspect and reason about libraries that have been shipped as bytecode. Ideally, a decompiler produces source code that can be compiled to bytecode such that the input bytecode and output bytecode are behaviorally equivalent. When we look for bugs, we will use a more modest quality measure: a decompiler should produce source code that compiles. If CFR maps a valid bytecode program to a source program that doesn't compile, we say that *CFR fails*.

We define a valid bytecode program as a set of class-files that each individually verifies and depends only on classes in the program itself or in the standard library. A class `A` depends on another class `B` if `A` mentions `B` anywhere in its bytecode. This can happen in many places, such as in an extends-clause, in a type annotation, in a new-expression, or in a type cast.

Our example begins with the discovery of a bug in CFR. We ran CFR on a valid Java bytecode program with 17 classes and then we ran `javac` on the produced source program, which led to this error message from `javac`:

```
... error: illegal start of expression
if (var2_3.hasNext()) ** break;
```

Now we would like to send a bug report to CFR, but it can be hard to locate the bug in 17 classes. In this paper we focus on reducing the bytecode program to one with *a subset* of the classes that still induces CFR to fail with the same bug report. Thus, the reducer *picks* classes without *changing* them.

The task of reducing a set of classes to a smaller set of classes is of the kind for which delta debugging usually excels. We implemented the delta debugging algorithm called `ddmin` by Zeller and Hildebrandt [27] such that it works on a list of classes. However, the result of reducing our Java bytecode program with 17 classes was disappointing: the result was a program with 14 classes.

Figure 1 illustrates our run of `ddmin`. The boxes and ×'s show which classes were input to an iteration of `ddmin`, while the column labeled fail shows whether CFR failed (marked with *yes*), succeeded (marked with *no*), or whether the bytecode program was invalid (marked with ?). In most cases, the input bytecode program is invalid so to highlight the few steps with valid inputs, we use boxes in those steps. Specifically, when CFR reproduces the bug we use □, and in all other cases with valid inputs we use ■.

The many iterations with invalid bytecode programs inputs are of no help towards reduction. Additionally, each invocation of CFR and `javac` can take between a couple of seconds and multiple minutes, which decreases scalability.

Regehr et al. [21] identified this kind of problem in 2012 and called it the *test-case validity problem*. They also identified two kinds of solutions, namely:

(1) detect invalid inputs or
(2) avoid invalid inputs.

In the context of C, Regehr et al. [21] used two tools to *detect* invalid code, which led to an excellent reducer. However, they left *avoiding* invalid code as an open problem.

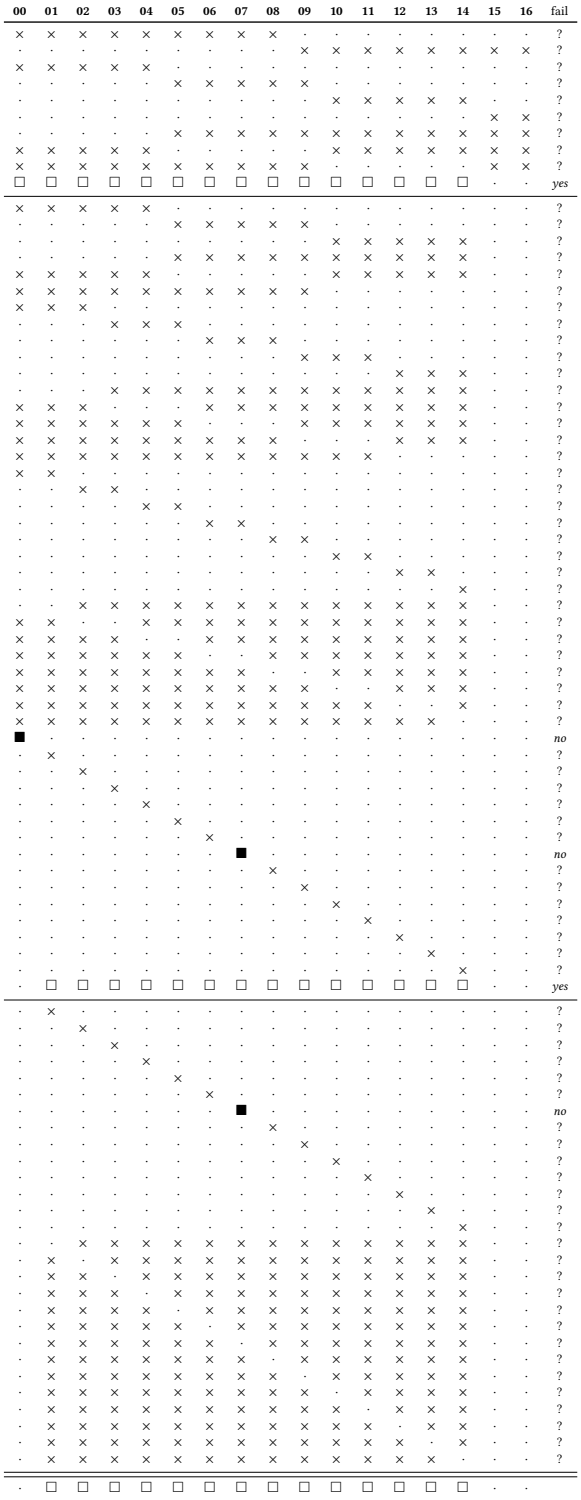

Figure 1: A detailed run of the example using unmodified delta debugging (`ddmin`). The rows are the iterations of `ddmin`. The columns identifies classes (represented using a number) in the input of each iteration: if a class is included it is marked with □, ■, or ×.

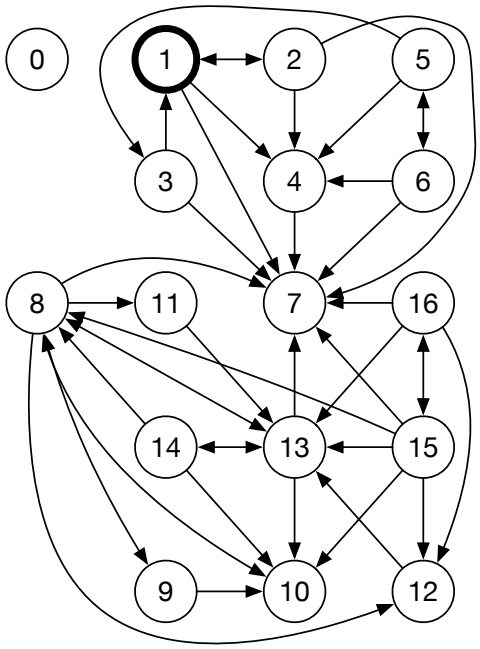

Figure 2: The dependency graph of our example program. The nodes are classes in the program and the edges represent references to other classes. The class marked 1 induces the bug in the decompiler.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	fail
□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	·	·	*yes*
■	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	*no*
·	·	·	·	·	·	·	■	·	·	·	·	·	·	·	·	·	*no*
·	□	□	□	□	□	□	□	□	□	□	□	□	□	□	·	·	*yes*
·	·	·	·	·	·	·	■	·	·	·	·	·	·	·	·	·	*no*
·	□	□	□	□	□	□	□	□	□	□	□	□	□	□	·	·	

Figure 3: A run where all the invalid bytecode program inputs have been filtered out before execution (`verify`).

Inspired by the success of Regehr et al. [21], our first attempt to improve the situation was to *detect* invalid bytecode programs. Specifically, we enhanced `ddmin` to a version called `verify` that checks, in every iteration, that the bytecode program is valid before running CFR and `javac`. Given that the original bytecode program is valid and that each class stays unchanged, a check of whether a bytecode program is valid boils down to checking that all dependencies are present. We do this by going through each class to find its dependencies, after which we assemble the dependencies into a graph.

Figure 2 shows the dependency graph for our example; each node represents a class and each edge represents a dependency. The classes are numbered from 0 to 16 (corresponding to numbers in Figure 1), for simplicity. The edge $1 \rightarrow 4$ means that class 1 depends on class 4. Sometimes classes are tightly coupled, in that case bidirectional edges are possible. Using this graph, `verify` can check for each iteration that all the dependencies are present before running CFR. Figure 3 shows that `verify` invokes CFR and `javac` just five times, yet still produces a program with 14 classes. In Section 5, our experiments show that `verify` is 3x faster than `ddmin` on a list of classes, on average.

We have found that the actual error is induced by class 1, marked in bold in Figure 2. However, for a bytecode program with class 1 to be valid, classes 2, 4, and 7 also have to be present. Thus, the smallest valid input that induces an error in CFR is $\{1, 2, 4, 7\}$, which is 3.5x smaller than the result given by `ddmin` and `verify`. This raises the question: why do `ddmin` and `verify` do poorly and what can we do about it?

The problem has to do with a lack of *monotonicity* that we explain now. In our example, consider the two valid, failure-inducing inputs $\{1, 2, 4, 7\}$ and $\{0, 1, \ldots, 16\}$. Figure 3 shows many sets S where

$$\{1, 2, 4, 7\} \subseteq S \subseteq \{0, 1, \ldots, 16\}$$

and in every case, S is invalid bytecode. Thus, we don't have the property that as inputs get bigger, failure is preserved. Equivalently, we don't have the property that as inputs get smaller, nonfailure is preserved. In other words, when we run CFR followed by `javac` on possibly invalid bytecode, this combined operation fails to be monotonic.

The lack of monotonicity has a big effect on the reduction process. Specifically, the process can move from a failure-inducing input such as $\{0, 1, \ldots, 16\}$ to a smaller input such as $\{0, 1, \ldots, 8\}$ that induces no failure, and still miss the even smaller, failure-inducing input $\{1, 2, 4, 7\}$. For example, if we from $\{0, 1, \ldots, 8\}$ remove some classes that had missing dependencies, the removal may make the input valid again, hence make the failure reappear.

We note that the original paper on `ddmin` assumes that "*failure is monotone*" [27, Section VIII]. However, delta debugging has been successful even when monotonicy fails (including when the input is a C program) so what is different about our case? The answer is that

> *for input with many internal dependencies, monotonicity can fail spectacularly.*

Indeed, Figure 1 shows that almost every subset is invalid bytecode so trying $O(n^2)$ subsets among the (2^n) possible subsets has little chance of success.

Notice that in Figure 1, `ddmin` managed to remove the interdependent classes 15 and 16 in a single step. This was mostly due to a lucky ordering of the classes. We can see from this example that for `ddmin` to remove interdependent classes, it must remove them at the same time. An attempt to remove either one in a single step would run into invalid bytecode.

For another example, notice that if we remove class 11 from $\{0, 1, \ldots, 16\}$, we get an invalid bytecode program. By inspecting Figure 2 we can see that in addition to removing class 11, we also have to remove class 8, thus also class 13, and so on. For `ddmin` to have a chance to remove such a long dependency chain in a single step, we would need the classes to be ordered in a particularly fortunate way. However, given that the reduction problem is NP-complete, finding a good listing is a hard problem.

The above analysis has led us to abandon the idea of *detecting* invalid input and instead pursue how to *avoid* it. We will present a new approach that avoids invalid bytecode programs entirely by putting dependencies front and center. The key idea is to do reduction of dependency graphs, as we explain next.

3 REDUCTION OF DEPENDENCY GRAPHS

In this section we will distill the essence of reducing an input with internal dependencies in a way that avoids invalid inputs. Thus, we will run CFR followed by `javac` only on valid bytecode. Hence, all remaining violations of monotonicity, in the sense of Section 2, come from CFR and `javacc`, and those tend to be insignificant.

Let us assume that the validity of an input can be modeled with a dependency graph. If all elements in the input have no missing dependencies, the input is valid. Intuitively, if we group all elements with their dependencies, and with the dependencies of their dependencies, and so on, then picking such a group would be a valid input. For the case of a set of verified classes, no missing dependencies mean a valid bytecode program.

We avoid invalid inputs by changing the reduction problem from working with a list of elements to working with a list of *sets of elements*. We ensure that each such set of elements is a valid input by requiring that it is a self-contained subset without missing dependencies. Those subsets are the *transitively closed* subsets of nodes in the dependency graph of the input. Recall that the transitive closure (or simply *closure*) of a set of nodes is the smallest superset that is transitively closed.

Our reduction strategy is based on the idea that *a set of closures represents the union of those closures*, which makes sense because the union of two closures is itself a closure. This means that no matter what subset of the list of closures a reduction algorithm picks, the union of that subset would be a closure. And since every closure is valid input we have a strategy that *avoids* invalid inputs.

The dependency graph reduction strategy. Here is our strategy for reduction of an input with internal dependencies:

(1) Map the input to its dependency graph.
(2) Compute the closure of each node.
(3) Form a list of the closures.
(4) Run a reduction algorithm on the list of closures.
(5) Output the union of the reduced list of closures.

For our dependency graph in Figure 2, Step 2 maps the 17 nodes to the following 8 different closures: $S_1 = \{7, 8, \ldots, 16\}$, $S_2 = \{7, 8, \ldots, 14\}$, $S_3 = \{1, 2, \ldots, 7\}$, $S_4 = \{1, 2, 3, 4, 7\}$, $S_5 = \{1, 2, 4, 7\}$, $S_6 = \{4, 7\}$, $S_7 = \{7\}$, and $S_8 = \{0\}$. We have fewer closures than nodes because of cycles in the graph.

The above strategy leaves two aspects to be refined. First, in Step 3 we must decide how to order the closures. This turns out to be important, as we will discuss below. Second, in Step 4 we must decide how to reduce the list of closures. We have some freedom here because when we remove a closure from the list, the result is again a list of closures. Thus, a reduction algorithm can remove closures and avoid invalid subsets entirely.

Using ddmin on a list of closures. In this section we use ddmin in Step 4 and refer to this algorithm as `closure`. In each iteration of ddmin, the union of the closures represents a valid bytecode program so all we need to do is to check whether CFR fails.

Figure 4 shows two runs of `closure` on the closures of the nodes in Figure 2. The difference lies in how we ordered the closures up front; any ordering is possible. In both cases, the number of iterations is much smaller than in the run of `ddmin` shown in Figure 1. The reason is that `closure` encounters no invalid subsets so it homes in on a solution in just four and five iterations, respectively. In this example `closure` run as fast as `verify`, in Figure 3, but the first run returns $S_3 = \{1, 2, \ldots, 7\}$, which is much better than the output in Figure 3, which is $\{1, 2, \ldots, 14\}$. The second run returns $S_5 = \{1, 2, 4, 7\}$, which is the best possible subset.

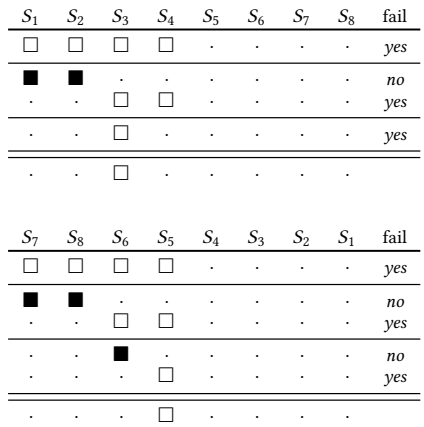

Figure 4: Two runs of `closure` on the example in Figure 2, where $\{1\}$ induces failure. The first run has the closures in an arbitrary order; the second has them sorted after size.

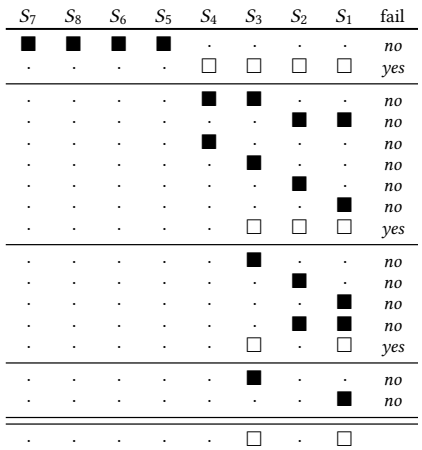

Figure 5: A run of `closure` on the example in Figure 2, where $\{1, 12\}$ induces failure.

For the second run in Figure 4, we sorted the closures by size, from smallest to largest. The reason this works well has to do with a quirk in running ddmin on a list of sets. Specifically, ddmin views S_3, S_4, S_5 as equally good reduced sets, because each one is a *single* closure. Whether ddmin produces one of S_3, S_4, and S_5 depends on the order of the closures, and, like in Section 2, some orders are more lucky than others. In each of the two runs in Figure 4, `closure` produced a single closure and behaved like binary search. We know that ddmin tends to process input from left to right, as we can see in Figure 1, so when we sort the closures by size we have a good chance to get the smallest closure.

The algorithm `closure` leaves room for improvement. For example, suppose the failure is induced by the combination of class 1 and class 12 (rather than only class 1). The smallest reduced set is $S_5 \cup S_2$, which has size 11. However, Figure 5 shows a run of `closure` that produces the larger set $S_3 \cup S_1$, which has size 16. Like before, this happens because ddmin looks for the smallest possible set of sets without considering the sizes of those sets. Here, ordering the closures by size is insufficient to get the best result. In the next section we present an algorithm that matches `closure` in simple cases and is better and faster in general.

4 BINARY REDUCTION

We will extend the classical reduction problem with a notion of *cost* that can model sizes of closures, and we will present an algorithm called Binary Reduction.

4.1 The Input Reduction Problem

For all sets we can refer to the elements using indicies: e.g. if A is a set, then $A_1, \ldots, A_{|A|}$ refer to the elements of A. If Σ is a set, then 2^Σ is the powerset of Σ; we use $D, S, \mathcal{U}$ to range over the elements of 2^Σ. We say that $P : 2^\Sigma \to \mathit{Bool}$, a predicate on subsets of Σ, is *monotonic* if $S_1 \subseteq S_2$ implies that $P(S_1) \Rightarrow P(S_2)$.

We recast reduction as a decision problem:

Definition 1 (Input Reduction Problem). *Given $(\Sigma, P, C, \mathcal{U}, k)$, where Σ is a set of symbols, $P : 2^\Sigma \to \mathit{Bool}$ is a polynomial-time monotonic predicate, $C : 2^\Sigma \to \mathbb{N}$ is a polynomial-time cost function, and $\mathcal{U} \in 2^\Sigma$ is a failure inducing input ($P(\mathcal{U}) = \mathit{True}$), $k \in \mathbb{N}$ is a natural number, decide $\exists S \subseteq \mathcal{U} : P(S) \wedge (C(S) < k)$.*

Intuitively, P represents buggy software and $\mathcal{U}$ represents failure-inducing input. We follow the convention of Misherghi and Su [19] that P returns False both in the case of invalid input and in the case of no failure. The novelty in the above definition is the cost function C. In order to define the problem as a decision problem, we use the standard technique of asking whether the cost of S exceeds a threshold k. Many instantiations are possible, including the following four.

Original Problem. In the seminal delta debugging paper [27], Σ is the index set of the input list, $\mathcal{U} = 2^\Sigma$, and $C(S) = |S|$. The problem is to find the smallest subset of the index set that induces a failure.

Syntax Trees. In the paper on hierarchical delta debugging [19], Σ is a set of subtrees of a tree, $\mathcal{U} = 2^\Sigma$, and $C(S) = |S|$. Thus, the problem is to find the fewest subtrees that induce a failure. Contrary to the original paper, if we define the cost function to be the sum of the *sizes* of the subtrees, a reducer that solves the problem will aim to choose the smallest subtrees.

Set of Sets. If we want to minimize the *union* of a set of sets, we can pick $\Sigma = 2^E$, where E is a set, $\mathcal{U} = 2^\Sigma$, and $C(S) = |\cup S|$. Additionally, we can lift any predicate $Q : 2^E \to \mathit{Bool}$ on subsets of E to a predicate $P : 2^\Sigma \to \mathit{Bool}$ by defining $P(S) = Q(\cup S)$.

Dependency Graphs. The previous section explains how to do reduction of a dependency graph by mapping it to the *Set of Sets* problem above. The idea is to compute the closures of the nodes in the dependency graph and then to find the smallest union of the closures that satisfies the predicate.

Algorithm 1: Binary Reduction

Input: $(\Sigma, P, C, \mathcal{U}, k)$
Define: $A \preceq_{S'} B := C(S' \cup \{A\}) \leq C(S' \cup \{B\})$
Data: $S \leftarrow \emptyset$ and $D \leftarrow \mathrm{sort}_{\preceq_S}(\mathcal{U})$
while $r > 0$ **where** $r \leftarrow \min r$ *st.* $P\left(S \cup \left\{D_j : j \leq r\right\}\right)$ **do**
  $S \leftarrow S \cup \{D_r\}$
  $D \leftarrow \mathrm{sort}_{\preceq_S}\left(\left\{D_j : j < r\right\}\right)$
end
return $C(S) < k$

Misherghi and Su [19] proved that the hierarchical delta debugging problem is NP-complete. In a similar manner, we prove that the Input Reduction Problem is NP-complete.

Theorem 1. *The Input Reduction Problem is NP-complete.*

Proof. The Input Deduction Problem is in NP because, given a witness $S \subseteq \mathcal{U}$, we can check in polynomial time that $P(S) \wedge (C(S) < k)$ since P and C runs in polynomial time.

We show that the Input Deduction Problem is NP-hard by reducing from the Hitting Set Problem, which is NP-complete [15]. The Hitting Set Problem is: given (Σ, Z, k), where $Z \subseteq 2^\Sigma$ is a set of sets, decide $\exists S \subseteq \cup_i Z_i : (\forall i : S \cap Z_i \neq \emptyset) \wedge (|S| < k)$. The reduction works as follows. Define $\mathcal{U} = \cup_i Z_i$ and $P(S) = (\forall i : S \cap Z_i \neq \emptyset)$ and $C(S) = |S|$. Notice that P is monotonic. Notice also that if $(\Sigma, P, C, \mathcal{U}, k)$ has a solution S, then $(S \subseteq \cup_i Z_i = \mathcal{U})$, $(\forall i : S \cap Z_i \neq \emptyset)$, and $(C(S) = |S| < k)$. This means that S is also a solution to (Σ, Z, k). □

Our problem is NP-complete and, unless P = NP, the best we can do in polynomial time is an approximation. In the next section, we will present a polynomial-time approximation algorithm for solving the input reduction problem.

4.2 The Binary Reduction Algorithm

Recall that in Figure 5, `closure` makes a bad choice and rejects the left half of the input. The closures on the left are insufficient to induce a failure. Instead, `closure` finds a much worse solution among the bigger closures on the right. The reason is that `closure` takes no advantage of getting a sorted input list.

We introduce Algorithm 1, an algorithm called Binary Reduction. The algorithm is inspired by the second run of `closure` in fig. 4, where ddmin operates like a binary search and quickly finds a single closure. Binary Reduction extends this idea to work in cases where multiple closures are required. We use $(\mathrm{sort}_{\preceq_S}(X))$ to denote the sorting of X according to the total order $\preceq_S$ and $(\min r \text{ st. } p)$ to denote finding the smallest r such that p is satisfied.

The idea is to maintain two sets S and a D. Here, S is the set of elements that we know are in the final set, and D is a sorted set of elements still to be searched. We initialize S to be empty, indicating we know nothing, and we initialize D to be the input, as we want to search the entire space.

The algorithm searches for the minimal prefix of a sorted listing of D that together with S satisfies P. Since we know that $S \cup \left\{D_j : j \leq r\right\}$ is the smallest prefix that satisfies P, we also know that removing D_r from the sets would make P false, therefore D_r

S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	fail
·	·	·	·	·	·	·	·	*no*
□	□	□	□	·	·	·	·	*yes*
■	·	·	·	·	·	·	·	*no*
■	■	·	·	·	·	·	·	*no*
□	□	□	·	·	·	·	·	*yes*
·	·	⊠	·	·	·	·	·	*yes*
·	·	⊠	·	·	·	·	·	

S_7	S_8	S_6	S_5	S_4	S_3	S_2	S_1	fail
·	·	·	·	·	·	·	·	*no*
□	□	□	□	·	·	·	·	*yes*
■	·	·	·	·	·	·	·	*no*
■	■	·	·	·	·	·	·	*no*
■	■	■	·	·	·	·	·	*no*
·	·	·	⊠	·	·	·	·	*yes*
·	·	·	⊠	·	·	·	·	

Figure 6: Two runs of Binary Reduction (`binary`) on the example in Figure 2, where $\{1\}$ induces failure. The first run has $C(X) = |X|$; the second has $C(X) = |\cup X|$.

S_7	S_8	S_6	S_5	S_4	S_3	S_2	S_1	fail
·	·	·	·	·	·	·	·	*no*
■	■	■	■	·	·	·	·	*no*
■	■	■	■	■	■	·	·	*no*
□	□	□	□	□	□	□	·	*yes*
·	·	·	·	·	·	⊠	·	*no*
■	■	■	·	·	·	⊠	·	*no*
□	□	□	□	□	·	⊠	·	*yes*
□	□	□	□	·	·	⊠	·	*yes*
·	·	·	⊠	·	·	⊠	·	*yes*
·	·	·	⊠	·	·	⊠	·	

Figure 7: A run of Binary Reduction (`binary`) on the example in Figure 2, with $C(X) = |\cup X|$ and failure induced by $\{1, 12\}$.

must be part of the final set. We can therefore add D_r to S and reduce the search space to the smaller prefix without D_r, $\left\{D_j : j < r\right\}$. We continue to reduce the search space until the smallest prefix is empty ($r = 0$). Since we only added elements to the solution that were required and since P is monotonic, the solution is *one-minimal* [27]: if any element is removed from S, then P is no longer satisfied.

The core of the algorithm is the search for the smallest prefix of D that satisfies P. In general, this takes $O(n)$ time, where n is the size of the search space. However, we have a monotonic P so

$$P(\emptyset) \Rightarrow P(\{D_1\}) \Rightarrow P(\{D_1, D_2\}) \Rightarrow \ldots \Rightarrow P(\{D_1, D_2, \ldots, D_n\})$$

and thus we can use binary search.

The final touch is to keep the set D sorted, using the cost function C. Our idea is to use the cost function to sort the search space such that low-cost elements are chosen early. This is a greedy algorithm that makes the best pick possible in each iteration. As with other greedy algorithms, this may fail to produce the best global solution, yet our experiments show that the results are good in practice. Notice that every iteration sorts D according to the cost of the union of the currently selected set S and the individual inputs. This is an advantage because sometimes the cost of the union of two input sets does not equal the sum of the cost of each of the sets.

We will use Binary Reduction in Step 4 of the strategy in Section 3; we refer to this algorithm as `binary`. The first diagram in Figure 6 shows a run of `binary` on the example in Figure 2, with a natural cost function $C(X) = |X|$. We use ⊠ to mark the D_r that is added to the solution in the line $S \leftarrow S \cup \{D_r\}$. Like in the first run of delta debugging over the closures of the graph in Figure 4, we get S_3, which is suboptimal. However, in contrast to `closure`, we can easily modify `binary` to use a more interesting cost function, like the number of elements in the union of the sets $C(X) = |\cup X|$. Figure 6 shows that run. Like in the second run of `closure` in Figure 4, we get S_5, which is the best solution.

Figure 7 shows a run of `binary` on the example in Figure 2, but this time the failure is induced by $\{1, 12\}$. In contrast to the run of `closure` in Figure 5, we get the best solution $S_5 \cup S_2$. Notice that the run took only 3 binary searches and 9 invocations of P.

Even thought the choice of $C(X) = |\cup X|$ solves our problem, we could imagine more interesting cost functions like the total size of classes. Binary Reduction greedily choose a local minimum regardless of cost function, but we expect that it performs best if the cost function is monotone in the size of X.

Complexity analysis. The complexity of Binary Reduction depends on the complexity of the cost function C ($\mathbf{\$C}$), the predicate P ($\mathbf{\$P}$), the size $n = |\mathcal{U}|$ of the input, and the final size s of the reduction. We do at most s binary searches, with $O(\log n)$ invocations of P and worst-case n calculations of $C(S)$ as part of sorting (assuming caching) which takes $O(n \log n)$ time. So in total we have

$$O(s\,(\log n \cdot \mathbf{\$P}(n) + n \cdot \mathbf{\$C}(s) + n \log n))\,.$$

Inspecting the time complexity of the algorithm we can see that we will make at most $O(s \log n)$ invocations of P. Since s is bound by n, the complexity of the algorithm is $O(n \log n)$ iterations.

5 EXPERIMENTAL RESULTS

This section presents an empirical evaluation of using dependency graphs and Binary Reduction for reduction. We have implemented those techniques in a tool for Java bytecode programs called J-Reduce. J-Reduce is a general tool for reducing inputs while preserving errors. We will use three decompilers as part of the evaluation, yet any tool that takes Java bytecode as input could have been used. The evaluation supports the two main claims of the paper:

(1) Reduction based on a list of closures is faster and better than reduction based on a list of classes. When we run ddmin on a list of classes, we time out 75% of the runs after an hour. In contrast, when we run ddmin on a list of closures, we time out only 9% of the runs after an hour. Including the timeouts, the list-of-closures approach gives 7x speedup and 1.07x smaller results, on average.

(2) Binary Reduction is faster and better than ddmin. Only 1% of the runs of Binary Reduction on a list of closures time out after an hour. Including the timeouts, Binary Reduction gives 1.7x speedup and 1.15x smaller results, on average, compared to running ddmin on the same input. Overall, we get 12x speedup and 1.24x smaller results compared to running ddmin on a list of classes.

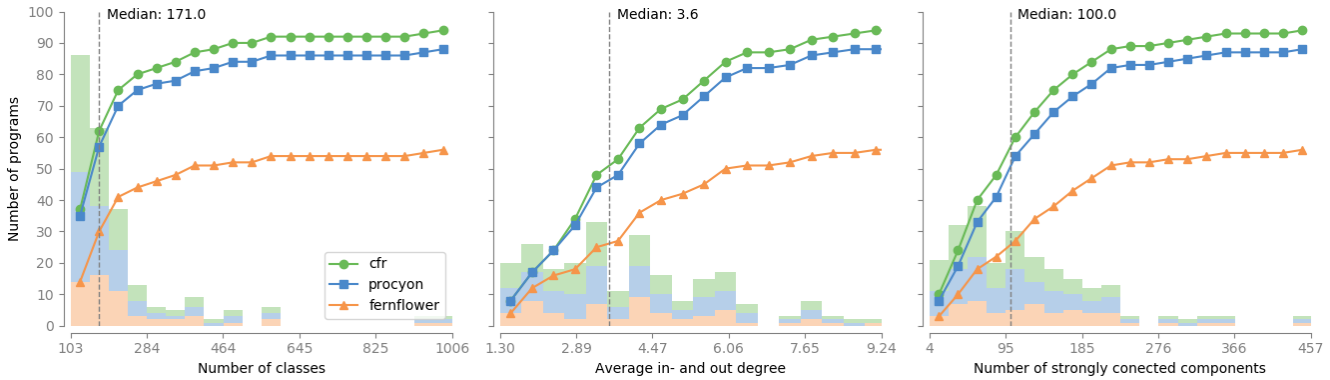

Figure 8: These three histograms show the distribution of the failure inducing inputs over three metrics; number of classes, the average in- and out degree of the underlying dependency graph, and the number of strongly connected components.

5.1 Experimental Setup

5.1.1 Implementation. J-Reduce has a single frontend that extracts a dependency graph from binary Java class files. Specifically, J-Reduce scans through each class-file to search for references to other classes and assembles them into a dependency graph. The common frontend and backend enable easy comparison of the algorithms. J-Reduce implements four different reduction algorithms:

- `ddmin`: Classical delta debugging on a list of classes.
- `verify`: Uses `ddmin` plus detection of invalid bytecode.
- `closure`: Uses ddmin on a list of closures, sorted after size.
- `binary`: Uses Binary Reduction on a list of closures.

We implemented J-Reduce in 7,929 lines of Haskell code that passed FSE's artifact evaluation [13] and is open source[1].

5.1.2 Choice of Predicate. For testing of the decompilers, we use the property that a decompiler should produce source code that compiles with `javac`; otherwise it has a bug. We use the predicate that `javac` produces the same bug as the original bug.

To get a monotonic predicate we took special care to keep all inputs except the reduced class files exactly the same. For example, the internal ordering in the file system may play a role in the output of the decompilers and in `javac`. Specifically, `javac` produces only a subset of the bugs in the source code, depending on which files it reads first. So, we kept a sorted lists of files and only wrote to the file system and jars in that order.

5.1.3 Choice of Decompilers. We choose three decompilers as the basis of our predicates: CFR [3, version 0.132], Fernflower [22, commit 8be977e76], and the decompiler from the Procyon project [23, version 0.5.30]. We set up each decompiler according to the instructions on its webpage. We ran Fernflower with the `-dgs=1` flag to enable handling of generics. We ran CFR with `--caseinsensitivefs true`. We ran Procyon with no special arguments.

5.1.4 Benchmarks. Our benchmarks are 100 large Java programs that we obtained from the NJR project [20]. We selected programs that each has at least 100 classes and for which we have source code. We focus on bytecode files that we have produced from source code ourselves to ensure that we start each reduction with a valid bytecode program. Some of the projects are dependent on large-scale libraries, but our reduction leaves those libraries unchanged and we exclude them from the dependency graph and our measurements.

We found that CFR fails on 94% of the programs, Fernflower fails on 56%, and Procyon fails on 88%. Thus, in total we have 238 failure-inducing inputs.

Figure 8 shows how the distribution of the inputs over three metrics: number of classes, average in-degree and out-degree in the underlying dependency graph (excluding self-loops), and number of strongly connected components. The inputs contain a median of 171 classes, and between 103 and 1006 classes. The benchmarks are diverse both in terms of the in-degree and out-degree, with a median of 3.6, and in terms of the number of strongly connected components, with a median of 100.

5.1.5 Platform. We performed the experiments on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 Gb RAM. We executed the experiments using OpenJDK (1.8.0_172-02). We ran the experiments in parallel in batches of 8. We ran each reduction for no more than an hour (3600s).

Table 1: Aggregated results of all the runs. The first column indicates the percentage of runs that we had to timeout. The second column is the average (GM) final relative size after reduction. The third columns are the average (GM) running times in seconds. Smaller is better.

	timeout	final size	time [s]
`binary`	0.8%	25.7%	203
`closure`	8.8%	29.8%	336
`verify`	19.8%	42.6%	750
`ddmin`	74.8%	31.9%	2339

5.2 Results

For each reduction algorithm (`binary`, `closure`, `verify`, and `ddmin`) we measured the total time in seconds, the number of invocations

[1] https://github.com/ucla-pls/jreduce

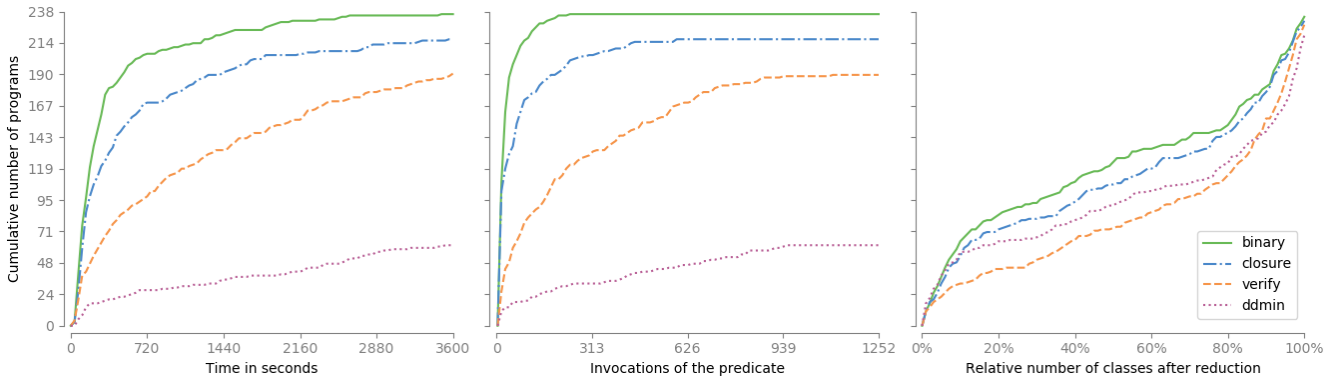

Figure 9: For reduction that preserves the full bug, the first two charts show the number of cases that terminate within x seconds and x iterations, and the third chart shows the final number of classes relative to the original size. Higher is better.

of the predicate made by the algorithm, and the fraction of classes left in the output after reduction. The results are shown in Figure 9. If a tool was timed out after an hour (3,600 seconds), we report the smallest set of classes that had been found to preserve the bug. This reflects that a user can use the best result available at time out.

The times include the generation of the graph (median 0.5s, max 7.1s), the initial run that tests if the predicate is true, and for the tools that used closures, the calculation of the closures (median 5.0ms, max 96.0ms).

Table 1 shows the aggregated results of all the runs: the percentage of the runs that we time out, the geometric mean of the relative final size, and the mean time used (including timeouts). The geometric mean allows us to talk about how many times a tool is better than another, based on how much is left after reduction.

We have also plotted all the results in cumulative charts. Figure 9 shows the results of the four configurations in three charts. The first chart shows how many programs that each reducer has finished after some seconds. The second chart shows how many programs that each reducer has either finished or timed out on after some invocations of the predicate. The third and final chart represents the relative size after reduction for an hour.

First, let us evaluate how ddmin (`ddmin`) performs against ddmin plus detection of invalid bytecode (`verify`). Unsurprisingly, `verify` is much faster than `ddmin`, because it does not have to run the predicate for the cases where not all the dependencies are present. Also, `verify` timeouts on 19.8% of the programs where `ddmin` timeouts on 74.8%. The resulting size of `verify`, however, is worse, with 42.6% average final size against `ddmin`'s average of 31.9%. There are two factors that affect this. One, adding the verifier makes the predicate more non-monotonic: missing dependencies in classes not visited by `javac` emerge as a problem. Two, our dependency graph may be an over-approximation of the actual dependencies used by the decompilers and compiler. This means that our verifier can reject a program that might decompile and make `javac` produce an error.

Second, let us evaluate runs of ddmin on a list of closures (named `closure`). This is affected by the overapproximation of the dependency graph, but the number of items is now both smaller (median 100 vs median 171), and also the predicate is now monotonic (disregarding non-determinism). This has a dramatic effect on speed and leads to a smaller output. `closure` only timeouts in 8.8% of all the inputs, and produces on average 29.8% final size. `closure` preforms better on most of the inputs, but `ddmin` outperforms `closure` in a few cases. We think this happens because the dependency graph is an overapproximation of the actual dependencies used by the compiler. This means that `ddmin` in some cases can remove an extra class, because it is in reality not needed by the compiler.

Third, let us evaluate runs of Binary Reduction on a list of closures (`binary`). `binary` performs better than `closure`, with a timoute rate of only 0.8%, and is on average 1.7x faster. The final size of the reduction is also better, with 25.7% final size on average. The better reduction can be attributed to two factors: fewer timeouts and Binary Reduction's ability to pick the smallest closure. When controlling for the fewer timeouts, by not counting the benchmarks where either of the algorithms timed out, Binary Reduction is able to produce 1.11x smaller results than delta debugging. On a few cases `ddmin` outperforms `binary`, this is partly due to the overapproximation, but also that some of the benchmarks could yield different results when run twice. `ddmin` sometimes runs the same reduction candidate twice, which means that it has a higher chance of getting lucky and accepting the reduction.

In conclusion, using Binary Reduction on the dependency graph of Java bytecode programs are 12x faster and 1.24x smaller results on average than delta debugging directly on the list of classes.

5.3 Threats to Validity

5.3.1 External Validity. The primary threats to external validity is the choice of domain. We chose the domain of decompilers, because bugs were plentiful and easy to find. We do, however, believe that the results extend to all domains with inputs with internal dependencies, and especially to domains that expect valid bytecode programs as inputs.

5.3.2 Internal Validity. We chose 100 fairly large benchmarks at random from the NJR repository; we deem them to be representative of real life programs. We chose programs with over 100 classes to go beyond what people may be willing to reduce by hand. The timeout

time was set at one hour, which might skew the study; however, we think that few users would run a reduction program for more than an hour. Our definition of a bug in a decompiler (*produces code that does not compile*) is not the strongest definition. We could expect that we could find even more bugs if we had used a stronger requirement. This does however not affect our study as we find plenty bugs.

In our experiment we reduced using a cost function that tries to minimize the number of classes, however the total size of the classes would also be an interesting metric, as two small classes might be better than one big class. Our technique is sufficiently general to reduce using any cost function, though we do expect ddmin to perform even worse in this case and it would be an unfair comparison as ddmin can only reduce based on counts. While we believe that ddmin is an adequate baseline, we could calculate the reduction approximation ratio of both algorithms if we had an ideal reduction, which we could find by doing an exhaustive search. We leave this to future work, perhaps for a smaller benchmark suite. Finally, the decompilers that we have chosen are not completely deterministic, which means that the predicates, even over the closures, are not completely monotonic. We see this as a strength of the study since in real life programs are often not deterministic, and predicates are often not completely monotonic.

5.4 Data Availability

We have made the raw data used for the analysis available [14]. It includes two files: a "benchmarks.csv" file, which contains the data for histogram in Figure 8, and a "deliverable.csv" file, which contains the data for the cumulative diagrams in Figure 9 and averages in Table 1.

6 REPORTING BUGS

Section 5 listed results from reducing input in a general manner that preserves the output from `javac` in its entirety. The median reduced bytecode program has 84 classes, which is too many to include in a succinct bug report. This observation led us to consider how domain-specific knowledge about `javac` can lead to additional reduction. We found that the output from `javac` may list multiple problems so a straightforward idea is to preserve *less* than the entire output. As a radical step towards more aggressive reduction, we ran an experiment in which we preserve only `javac`'s exit code. Thus, we preserve that `javac` returns an error, but not which one(s). Indeed, the final list of problems may have no overlap with the initial list.

Our experiment with running Binary Reduction on 238 programs took 34 minutes in wall clock time, or 13 hours in processing time, for an average of 3 minutes per program. The median reduced bytecode program had 2 classes, excluding libraries. Indeed, in 133 cases out of 238 cases, the reduced program had 1 or 2 classes: 57 cases for CFR, 46 for Procyon, and 30 for Fernflower. The output from `javacc` included many distinct error messages (disregarding line number and class): 67 distinct error messages for CFR, 83 for Procyon, and 27 for Fernflower.

We used the results of the experiment to report 2 bugs to CFR, 1 bug to Procyon, and 2 to Fernflower. We choose the benchmarks of size no more than 2 classes, which induced errors that looked like a fixable bugs and were significantly different from each other. At the time of writing these lines, the developers of CFR have confirmed and fixed one of the bugs, the developers of Procyon have triaged the bug, but not yet fixed it, and the developers Fernflower have triaged the bugs but not had time to fix them.

We stopped short of filing additional bug reports because we are aware that two failure-inducing inputs may be about the same bug. We want to avoid reporting the same bug twice and we leave it to future work to find an effective way to categorize bug reports.

7 RELATED WORK

The literature on program reduction and delta debugging is rich and diverse. We will cover some of the most closely related papers from that literature and we will focus on three aspects. The first aspect is how previous work has dealt with the test-case validity problem, the second aspect is how our approach compares to various approaches to input reduction, and the final aspect looks at program reduction as slicing.

The Test-Case Validity Problem. Our paper is the first to *avoid* invalid input. We will discuss how some prominent papers have dealt with invalid input.

Zeller and Hildebrandt [27] introduced delta debugging. They wrote [27, Section VIII] that *"Delta Debugging assumes that failure is monotone"*. However, their paper showed how to apply Delta Debugging to a variety of input for which failure *isn't* monotone, namely C programs, Mozilla user actions, and UNIX commands. For each kind we can remove a few characters from a failure-inducing input and thereby change it into an invalid input. In some cases, we can remove additional characters and get another failure-inducing input. Delta debugging works well for those kinds of inputs because most *natural* subsets are valid. In contrast, for Java bytecode, most natural subsets are invalid. Our experiments show that for Java bytecode, delta debugging of a list of classes times out often and gives a disappointing factor of reduction.

Delta debugging has also been implemented in the Delta tool [18]. This tool uses a line-based algorithm that suppresses newlines below a particular depth in the syntax tree. This decreases the risk of removing half of a subtree and thereby producing invalid inputs.

Misherghi and Su [19], in their paper on hierarchical delta debugging, avoided invalid subsets by structuring the input as a syntax tree and by removing entire subtrees at a time. Their insight is that the elements of a subtree can be a natural subset of the input, such as a statement in a statement list. They found that they can remove a single statement from a statement list and preserve that the syntax tree is valid. Compared the classical delta debugging algorithm, the hierarchical approach gave a decrease in the number tests needed for C-program input by a factor of 11.5 on average. However, while each Java bytecode class is a natural subset of a bytecode program, most subsets of the classes are invalid. For Java bytecode, the top level of hierarchical delta debugging is delta debugging of a list of classes, which we have shown works poorly.

Regehr et al. [21] identified the problem with invalid input and pursued an approach, for C, that detects invalid input. The core of their approach is akin to the algorithm we called `verify`. They went further and built in detailed knowledge of C that enabled their tool to reduce C-program 25 times more than language-independent

tools. In contrast, our tool avoids invalid code and uses a general reducer. Our experiments show that for Java bytecode, detection of invalid input is slow and this gives a small factor of reduction.

Sun et al. [24] showed, with their tool Perses, how to avoid invalid inputs in a language-independent manner. They did this by transforming an input grammar into a convenient form that can guide reduction. They showed that this approach is competitive with less general approaches. However, the approach relies on that once the grammar has reached the convenient form, two specific transformations preserve validity. While indeed those transformations do preserve validity for many kinds of input, they often produce invalid subsets in the case where the input is a list of Java bytecode classes. The problem is that a grammar has no model of the many internal dependencies. Thus, while the generality of the approach is attractive, the approach is ineffective for inputs such as Java bytecode programs.

The resent tool Chisel [12] uses reinforcement learning to do fast debloating of C programs. The approach is 3.7–7.1x faster than competing approaches. The approach *detects* invalid input, as illustrated by the following quote from the paper: "Chisel simply rejects nonsensical programs without invoking the test script by using a simple dependency analysis, such as programs that do not contain the main function, variable declarations, variable initializations, or return statements.". We speculate that one can combine their idea of reinforcement learning with our idea of using a dependency graph to avoid invalid input. We leave this to future work.

In Cleve and Zeller's work on STRIPE [6], they tried to use different clustering techniques to increase the speed of delta debugging on an execution trace. Specifically, they wrote "[O]ur future work will concentrate on introducing domain knowledge into delta debugging. In the domain of code changes, we have seen significant improvements by grouping changes according to files, functions, or static program slices, and rejecting infeasible configurations[.]" We believe that we have solved this problem by giving the user a simple interface, dependencies, with which they can encode many different kinds of domain specific dependency information.

Approaches to Input Reduction. BiSect [7, 8] is a tool for use with git that does a binary search to find the commit that introduced a bug. This is akin to the binary search that we use to implement the min function in Binary Reduction. The BiSect technique does no reduction of the input.

The papers by Artho [2], by Li et al. [17], and by Yu et al. [26] all have the goal to isolate failure-inducing changes in a revision history. They use clever representations of revision histories and use variations of delta debugging to achieve the goal. In all three papers, dependencies among changes and validity of history slices play major roles. Artho [2] notes that, in the context of interdependent changes, the approach "cannot deal with certain changes affecting multiple files". Li et al. [17] *detects* invalid history slices, while Yu et al. [26] uses classical delta debugging with no optimization for invalid input. We speculate that those approaches can be enhanced with ways to avoid invalid history slices, in a way that is akin how we avoid invalid input. We leave this to future work.

Delta debugging has a wide range of applications. In particular, researchers have shown how to use delta debugging to help normalize, generalize, and improve test cases [9, 10, 16]. For test cases with many internal dependencies, our approach can be used to avoid giving invalid inputs to the reducer.

Program Slicing and De-bloating. Delta debugging can be used to slice a program [25], that is, reduce the program while preserving its behavior. When the slices are intended to be used as runnable program, such reduction is called de-bloating.

Delta debugging is a simple approach to slicing as it requires little knowledge about the program: we can reduce the program while preserving the observable properties like those given test cases. Binkley et al. [4] uses delta debugging to reduce a set of files using a technique they call observational slicing (ORBS). Our technique is sufficiently general that it can augment ORBS by allowing the user to define dependency edges between lines in different files.

J-Reduce functions perfectly as a program slicer. Since we are using a static analysis to detect the edges in the dependency graph we likely under-approximate edges that are the result of reflection. Under-approximating the dependency graph is acceptable, as it will only result in more strongly connected components in our algorithm. In the worst case we have exactly one strongly connected component for each input node, which means that we are just doing regular reduction. This might take longer, but will always output correct byte code.

Our technique is akin to Agrawal and Horgan [1], which uses a over-approximating static analysis to collect a dependency graph between statements in a program. It then uses a dynamic analysis to traverse the program and reduce the graph to see which nodes are actually executed. In contrast our technique starts with a possible under-approximating static analysis and does not need to run the program. This means that it can be used on other properties like finding bugs in decompilers. Since our technique tolerates under-approximation, we might use a dynamic analysis to generate the dependency graph. We leave this for future work.

8 CONCLUSION

We have presented a new approach to reducing failure-inducing input with many internal dependencies. Our approach uses a dependency graph to avoid invalid inputs, and it uses a new algorithm called Binary Reduction, that we showed works better than ddmin. We have implemented an open-source tool J-Reduce that reduces Java class-files. We evaluated our tool on decompilers, yet our tool works for any program that takes class files as input. Examples include static and dynamic analyses, code coverage tools, and code visualizers. Our tool is 12x faster and achieves more reduction than delta debugging. This enabled us to create and submit short bug reports for three Java bytecode decompilers.

Our approach may work well for other kinds of inputs with many internal dependencies. For example, our technique can be used for languages with module systems, such as C# or Python. We can also consider use of a dependency graph in which the nodes are methods and fields. We leave these points to future work.

ACKNOWLEDGMENTS

We thank John Bender, Shuyang Liu, Akshay Utture, and the anonymous reviewers for helpful suggestions. DARPA award number RF228-G1:5 and ONR award N00014-18-1-2037 supported us.

REFERENCES

[1] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *ACM SIGPlan Notices*, Vol. 25. ACM, 246–256.

[2] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.

[3] Lee Benfield. [n. d.]. CFR – another Java decompiler. ([n. d.]). http://www.benf.org/other/cfr/ (accessed Aug 24, 2018).

[4] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 109–120.

[5] Jong-Deok Choi and Andreas Zeller. 2002. Isolating Failure-inducing Thread Schedules. In *ISSTA*.

[6] Holger Cleve and Andreas Zeller. 2000. Finding failure causes through automated testing. *arXiv preprint cs/0012009* (2000).

[7] Wiktor Czajkowski. 2018. Sneaky Bugs and How to Find Them (with git bisect). *Netguru* (January 2018). https://www.netguru.co/codestories/sneaky-bugs-and-how-to-find-them.

[8] Developers. [n. d.]. Bisect. ([n. d.]). https://git-scm.com/docs/git-bisect (accessed Aug 24, 2018).

[9] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause reduction: delta debugging, even without bugs. *Software Testing, Verification and Reliability* 26, 1 (2016), 40–68.

[10] Alex Groce, Josie Holmes, and Kevin Kellar. 2017. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1–11.

[11] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. 2015. 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, 333–344.

[12] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 380–394.

[13] Christian Gram Kalhauge and Jens Palsberg. 2019. Artifact from "Binary Reduction of Dependency Graphs". https://doi.org/10.5281/zenodo.3262201

[14] Christian Gram Kalhauge and Jens Palsberg. 2019. Results from "Binary Reduction of Dependency Graphs". https://doi.org/10.5281/zenodo.2574326

[15] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher (Eds.). Plenum Press, 85–103.

[16] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 417–420.

[17] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2016. Precise semantic history slicing through dynamic delta refinement. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 495–506.

[18] S McPeak, DS Wilkerson, and S Goldsmith. 2015. Berkeley Delta. *URL http://delta.tigris.org* (2015).

[19] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *ICSE'06, International Conference on Software Engineering*.

[20] Jens Palsberg and Cristina Lopes. 2018. NJR: A Normalized Java Resource. In *SOAP'18, Proceedings of ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*.

[21] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *PLDI*.

[22] Roman Shevchenko and other contributors. [n. d.]. Fernflower. ([n. d.]). https://github.com/fesh0r/fernflower (accessed Aug 24, 2018).

[23] Mike Strobel. [n. d.]. Procyon Java Decompiler. ([n. d.]). https://bitbucket.org/mstrobel/procyon/overview (accessed Aug 24, 2018).

[24] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *ICSE'18, International Conference on Software Engineering*.

[25] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

[26] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. 2012. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *Journal of Systems and Software* 85, 10 (2012), 2305–2317.

[27] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *TSE* (2002).