UCLA

UCLA Electronic Theses and Dissertations

Title

Fine-grained Structural Testcase Reduction

Permalink

https://escholarship.org/uc/item/6cw1j1wt

Author

Xiao, Liran

Publication Date

2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library

University of California

UNIVERSITY OF CALIFORNIA
Los Angeles

Fine-grained Structural Testcase Reduction

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Liran Xiao

2021

© Copyright by

Liran Xiao

2021

ABSTRACT OF THE THESIS

Fine-grained Structural Testcase Reduction

by

Liran Xiao
Master of Science in Computer Science
University of California, Los Angeles, 2021
Professor Jens Palsberg, Chair

Modern programs grow more complicated and consist of a large number of components. As programs become more integrated, the root cause of program bugs is hidden in tons of details. For metaprograms taking structural inputs, it could cost many human resources to dig into a sophisticated testcase, while a small portion of code is enough to reveal the internal cause. This thesis focuses on reducing structural inputs on metaprograms and showing some methods for fine-grained structural testcase reduction for better debugging metaprograms.

The thesis of Liran Xiao is approved.

George P. Varghese

Todd D. Millstein

Jens Palsberg, Committee Chair

University of California, Los Angeles

2021

TABLE OF CONTENTS

1 Introduction **1**

2 Background and Prior Work **3**

2.1 Syntax-guided Testcase Reduction 3

2.2 Domain-specific Testcase Reduction 4

2.3 Semantic-guided Testcase Reduction 4

3 Approach and Implementation **6**

3.1 Break Dependency by Reduction Point Stubbing 7

3.1.1 Implementation towards JVM bytecode and decompiler predicate . . 10

3.2 Multi-option Binary Reduction 11

3.2.1 Application on Java Programming Language 13

4 Evaluation **15**

4.1 Experiment Setup 15

4.2 Statistics 16

5 Conclusion **19**

References **20**

LIST OF FIGURES

3.1 Java code example. 7

3.3 Running stubbed binary reduction on Code example 3.1 9

3.4 Dependency graph union process 10

3.5 `A:foo` stubbed result. 11

3.6 Denotations used in Figure 3.7 14

3.7 Restrictions derived from instructions. 14

4.1 Cumulative frequency diagrams of time cost in reduction. The charts show the number of instances that terminate within x seconds. Higher is better. 16

4.2 Cumulative frequency diagrams of size reduced after reduction relative to inital binary size. The charts show the number of instances that reduce x percent of size after reduction. Higher is better. 17

4.3 Cumulative frequency diagrams of size reduced after reduction relative to baseline. The charts show the number of instances that reduce x percent of size after reduction compared to the baseline artifact. Higher is better. 18

LIST OF TABLES

4.1 Aggregated results of size reduced. The percent represents the relative size of the final output compared to the initial binary. 17

4.2 Aggregated results of size reduced relative to baseline. The percent represents the relative size of the final output compared to the baseline artifact. 18

ACKNOWLEDGMENTS

I want to thank Prof. Jens Palsberg for his kindness, patience, and support during my graduate years. Thank you for welcoming me to join the testcase reduction project and leading me into the world of program analysis. Furthermore, you broaden my horizon in both academics and life.

I would also like to thank Dr. Christian Gram Kalhauge. You are always willing to answer my questions, and without your help, I will not be able to build my thesis work. You build the foundation for efficient testcase reduction.

Finally, I want to thank my parents. Though we are not living together, you still support me and give me life suggestions and encouragement. Words can not express my gratitude to them.

CHAPTER 1

Introduction

For debugging and fixing program bugs, programmers provide test inputs therefore programs can reproduce the faults. Many programs are designed to accept structural inputs, like compilers, interpreters, and web render engines. These programs (often called metaprograms) play vital roles in the software development process since their correctness provides a foundation of interpreted results (often another program) produced by them. However, these programs are often complicated and hard to debug. For example, GNU C Compiler (gcc) has more than 14.5 million lines of code [Lar15] and many new codes are pushed to its base to enable better optimizations and support the new architecture. Meanwhile, hundreds of bug reports are created over one week and most of them are left unassigned [bug]. Based on Sun's report [SLZ16], the average lifetime of GCC bugs is 200 days.

To preserve the correctness of metaprograms, several methods are proposed. Formal verification builds the mathematical foundation and proves the correctness of metaprograms in proof assistants, like Coq or TLA+. CompCert [Ler09] is the most well-known verification project which builds a proven C-compiler. However, because of language design flaws and the large complexity of proving in proof assistants, this method is not widely employed. While static analysis seems unrealistic due to the complexity of metaprograms, testing them through testcases is more popular. Model checkers [RH01] and fuzzing techniques [LLP19] are used to generating valid structural inputs for metaprograms.

For failure induced by testcases, whether automatically generated or reported by programmers, the developers still need much time to investigate the root cause. The reproduc-

ing testcases can contain a large portion of code that has no relationship with the real bugs. Testcase reduction is a technique to minimize program test inputs while the essential part of the code behind the input causing bugs is kept. Given a predicate, the reduction tool will conduct several runs of the predicate. During the process, the original input will be reduced and the final output will keep the predicate true.

Zeller and Hildebrandt [ZH02] introduced the delta debugging algorithm as a foundation for testcase reduction. The delta debugging algorithm focuses on the textual input where subsets of the original input are likely to be valid input. For example, they simplified an HTML file to find the failure-inducing line. However, for structural inputs, delta debugging could generate many invalid attempts if split by tokens. Therefore, syntax-guided [MS06, SLZ18], domain-specific [RCC12] and semantic-guided [KP19, Kal20] reduction methods are proposed to handle structural inputs.

The state of art semantic-guided reduction technique [Kal20] reduces the input in a coarse-grained way. For Java bytecode, it reduces the loosely connected classes and methods, while leaving the method body untouched. Our approach handles closely connected dependency and fine-grained testcase reduction by identifying dependency introduced by the method body.

The remainder of the thesis is as follows. Chapter 2 presents background information on prior work on testcase reduction and shows the motivation for improving the current state of art reducer. Chapter 3 describes the approach taken to handle fine-grained and closely connected dependency introduced from the method body. Chapter 4 demonstrates the evaluation of the new method on reducing Java bytecode for debugging bugs in popular decompilers. Chapter 5 concludes the thesis and shares some thoughts on the future work.

CHAPTER 2

Background and Prior Work

This thesis builds on an existing testcase reduction tool [Kal20]. In this chapter we describe the prior work and existing tools on how they handle the testcase reduction steps to avoid invalid attempts during delta debugging. DDmin [ZH02] algorithm is the first algorithm to handle testcase reduction. The algorithm splits inputs into tokens or lines and combines subsets of the elements as the final input. Token-based subsets work well on text inputs, like HTML or strings. When applied on C programs or Java bytecode, there is no natural method to split the input into subsets and avoid many invalid test inputs. Understanding the input helps the reduction problem.

2.1 Syntax-guided Testcase Reduction

Hierarchical delta debugging (HDD) [MS06] concerns tree-like inputs. It parses programs into syntax trees and conducts DDmin algorithm on nodes of each level of program trees. After processing all nodes at the same level, the algorithm prunes the unused subtrees. For example, several statements are at the same level of one program tree, and some unnecessary statements can be removed after HDD algorithm processes this level. Though HDD leverages the tree structure of programs, it doesn't understand the program grammar. For instance, removing variable names in the declaration will invalidate the program.

Perses [SLZ18] improves HDD by introducing a general method to handle language-independent grammars. Perses proposes a normal form similar to the extended Backus

Normal Form and defines a subsumption relationship between grammar nodes to lift nodes between different levels. Perses is good at preserving grammar correctness while it fails to handle semantics like variable declarations must happen before variable usage.

2.2 Domain-specific Testcase Reduction

C-Reduce [RCC12] addresses the testcase invalidity problem in a language-specific way. It takes advantage of the LLVM C frontend and calls several kinds of C-specific transformation, such as reducing pointer redirection level and combining variables definitions. C-Reduce also notices that undefined behavior or program violating program semantics caused by reduction is a significant issue, and it employs some semantic-checking interpreters to solve this problem. C-Reduce extends DDmin algorithm by detailed knowledge of the C programming language, and when reducing other programs, this method fails to work. Meanwhile, C-Reduce uses a passive way to handle invalid programs with semantic violations, meaning many failed attempts still exist.

2.3 Semantic-guided Testcase Reduction

Binary reduction [KP19] shows leveraging dependency in inputs can avoid invalid attempts and speed up testcase reduction. It computes the dependency graph of the given program and calculates the closure of each node. Many semantic violations are avoided by introducing dependencies and closures between elements, and the testcase reduction is now monotonic. The monotonicity further helps to reduce the number of attempts since we can employ binary searching to find failure-inducing closures. However, expressing dependency in graphs is hard for fine-grained level items, like methods and fields in Java classes.

Logical input reduction [Kal20] introduces propositional logic to describe the dependencies between input elements. Both syntactic and semantic dependencies are collected during

static program analysis. These dependencies are fed into the general binary reduction algorithm to identify failure-inducing logical closures. In evaluation, logical input reduction works better than any other testcase reducer.

Though logical input reduction achieves great success, it only handles the reduction in a coarse-grained manner. Logical input reduction considers method bodies unbreakable, leaving function parameters and instructions inside methods untouched. Removing instructions, especially method callings, can further reduce dependencies by removing logical constraints between logical variables. Directly applying logical reduction on method bodies doesn't help much since instructions inside method bodies are closely connected, and very few independent logical closures can be identified.

This thesis proposes a solution to remove logical dependencies for further testcase reduction. We also show a method to do multi-option binary reduction by leveraging monotonic options. The multi-option binary reduction facilitates the reduction of method parameters and related dependencies.

CHAPTER 3

Approach and Implementation

In this chapter, we describe the approach for fine-grained structural testcase reduction. Reduction points are the potential candidates we can try to remove from the final output. We show how to break logical dependencies by stubbing reduction points and improving the previous logical input reduction. We start from the dependency graph and stub vertices to remove dependency edges incoming from and towards other nodes. For Java bytecode and decompiler testcases, we define method calls as our reduction points and NOP substitution as our stubbing function. If some method calling invokes the actual bugs, we leverage binary reduction on the potential reduction points to minimize the output test input size.

We also show employing multi-option binary reduction on function arguments can further remove dependencies to make the final testcase smaller. By considering Java inheritance, we find that a monotone sequence of potential substitutions of function parameters can be found and used to do multi-option binary reduction. This process removes the function parameter type dependencies if only parent types are essential for triggering the potential bugs.

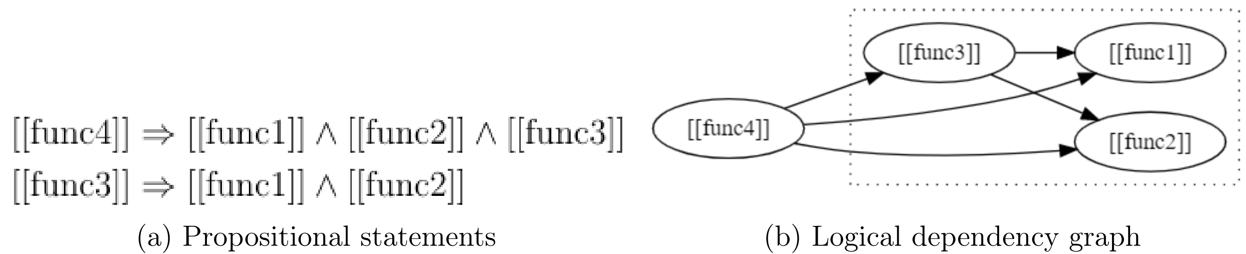
We first propose the algorithms for stubbing reduction points and multi-option binary reduction in the following sections. In each section, we discuss the implementation towards JVM bytecode and decompilers predicates.

3.1 Break Dependency by Reduction Point Stubbing

```java
public class Example {
    public void example() {
        int a = func1();
        int b = func2();
        int c = func3(a, b); // call to func3(a, b) incurs the bug
        func4(a, b, c);
    }
}
```

Figure 3.1: Java code example.

Taking Figure 3.1 as an example. From the view of dependencies, the three calls form a single closure. With logical input reduction, we collect the following propositional statements and dependency graph (use [[f]] to express the existence of function call f):



(a) Propositional statements (b) Logical dependency graph

From these statements, we find that logical input reduction can not reduce the example the minimum case (leaving `func3` and eliminating the other). Existence of `func3` requires the existence of `func1` and `func2`, where the function calls forms a closure.

Our approach takes a different view. Instead of preserving or removing the entire closure, we define reduction points as all function calls and define stub function $S(r, e)$. The stub function S takes the reduction point and environment around the reduction point as input and returns the replacement for removing this reduction point. The environment stands for the related information about the reduction point. For function calls, arguments are essential information in the environment of the reduction point.

For our program in Figure 3.1, the reduction points are all function calls, and the environment of each reduction point is argument information. We can define a simple stub function $S(r, e)$: ignore all environment and return zero as a replacement for these function calls returning integer values.

Though stub functions are designed to be semantic preserving, not all reduction points can be stubbed as some reduction points are the root cause of the bug. We introduce Algorithm 0 called stubbed binary reduction, a modified version of binary reduction [KP19], on the stub action and reduction points.

Algorithm 0: Stubbed Binary Reduction

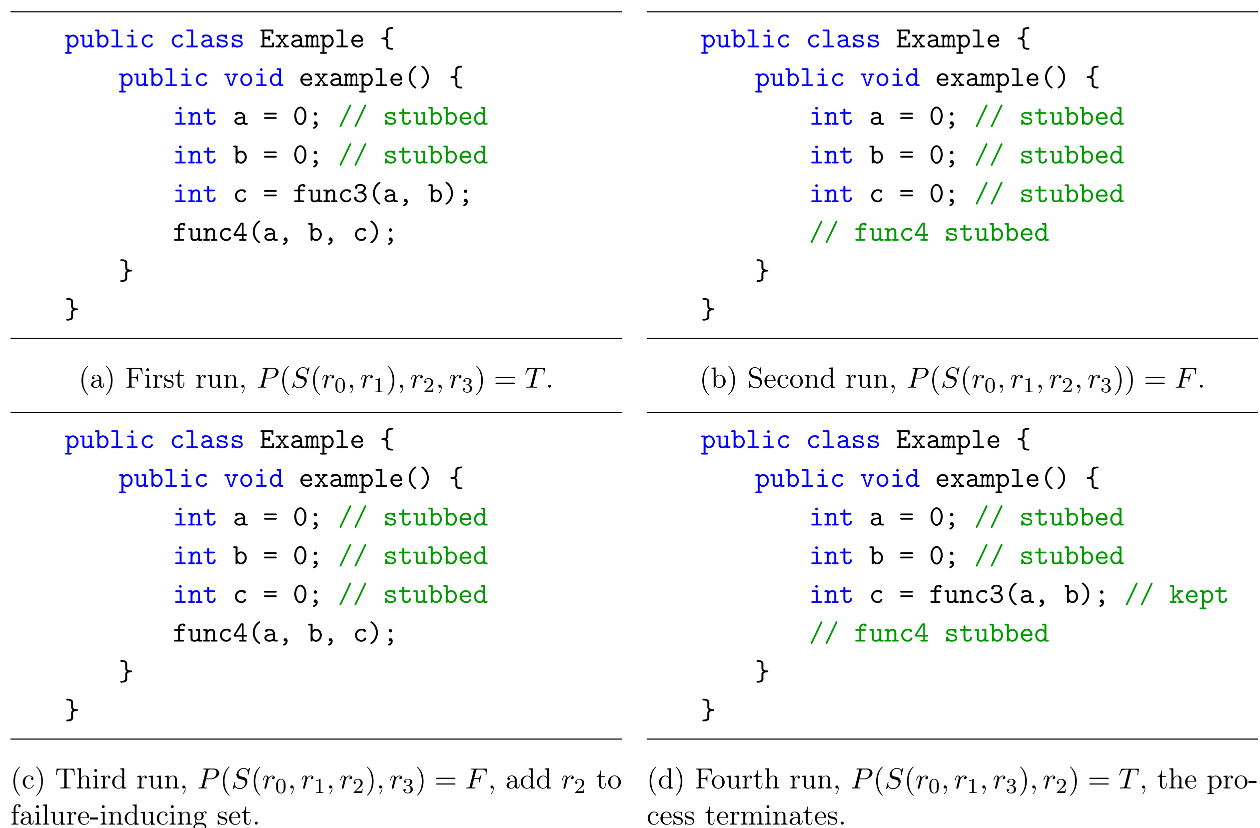
Input: Monotonic reduction point and environment search space $R = (r_i, e_i)$
Stub function $S : (r, e) \to r$
Predicate $P : r^{|R|} \to \{T, F\}$ where $P(\{r_1, \cdots, r_I\}) = T$
Output: Failure inducing set of reduction points F
Data: A current failure inducing set $F \leftarrow \emptyset$
A current stubbed reduction point cache $C \leftarrow \emptyset$
Current sorted search space $D \leftarrow R$, denote $D_j = (r_j, e_j)$
while $\neg P(F \cup \{S(D \cup C)\})$ **do**
 $q \leftarrow \min q$ st. $q > 0 \land P(F_r \cup \{r_j : j \leq q\} \cup \{S(D_k) : k > q\})$
 $F \leftarrow F \cup \{r_q\}$
 $C \leftarrow \{r_j : j > q\}$
 $D \leftarrow \{D_q : j < q\}$
end
return F

The idea behind Algorithm 0 is to do binary reduction while transforming the input to remove dependencies. The algorithm starts with testing stubbing all remain reduction points. If the test succeeds, return the current failure-inducing set. Otherwise, do binary searching to find the local minimum point: if any reduction point is stubbed, the predicate no longer holds. The algorithm assumes the input sequence is monotonic according to the predicate

$$\forall q, r \in \{1, \cdots, |R|\}, q \geq r \Rightarrow P(\{D_i : i \leq r\} \cup \{S(D_i) : i > r\}) \leq P(\{D_i : i \leq q\} \cup \{S(D_i) : i > q\})$$

In actual processing, the condition is easy to meet since we can select individual reduction points or carefully order the sequence to maintain monotonicity.

Taking Figure 3.1 as a running example again. The initial test stubs all reduction points, and the predicate returns false. Inside the loop, as Figure 3.3 shows, stubbed binary reduction runs binary searching for the last element (r_2) where if it gets stubbed will make the predicate fail. It then adds the failure-inducing element to the set F and starts the next run. Finally, it shows with failure-inducing set elements, and the rest elements all stubbed the predicate passes, which means the algorithm now terminates and returns the failure-inducing set.



(a) First run, $P(S(r_0, r_1), r_2, r_3) = T$.

(b) Second run, $P(S(r_0, r_1, r_2, r_3)) = F$.

(c) Third run, $P(S(r_0, r_1, r_2), r_3) = F$, add r_2 to failure-inducing set.

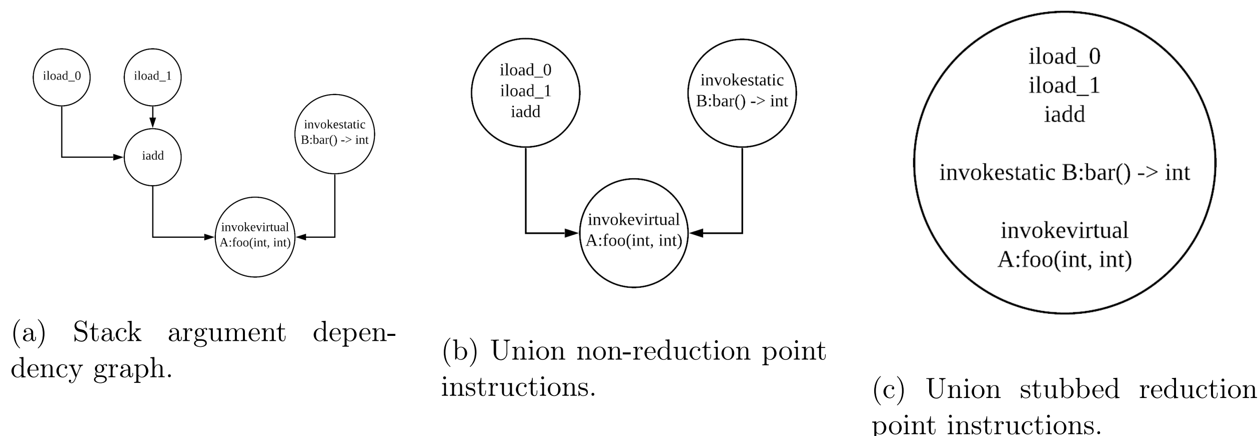
(d) Fourth run, $P(S(r_0, r_1, r_3), r_2) = T$, the process terminates.

Figure 3.3: Running stubbed binary reduction on Code example 3.1

3.1.1 Implementation towards JVM bytecode and decompiler predicate

We select `invoke` instructions (method calls) as our target reduction points for implementation towards JVM bytecode and decompiler predicate. Because of the nature of the JVM bytecode, the `invoke` instructions are independent, and the reduction point sequence is monotonic.

Though we can replace all method call arguments with a sequence of `pop` instructions, the outcome bytecode is non-natural, and decompilers could generate more errors in terms of the artificial bytecode input. We notice that the dependencies between method calls and arguments are clear under the JVM stack-based virtual machine. Hence, we record the argument dependencies as a dependency graph and do union on the resulting graph to minimize the dependency relationship. The process is shown in Figure 3.4. Figure 3.4a shows the original dependency graph derived from stack argument dependencies, and Figure 3.4b demonstrates the initial union process. We ignore all edges from reduction points and union all other instructions with one outcoming vertex in this step. In Figure 3.4c, we show that if `B:bar` function is stubbed, we can safely union it with `A:foo`.



(a) Stack argument dependency graph.

(b) Union non-reduction point instructions.

(c) Union stubbed reduction point instructions.

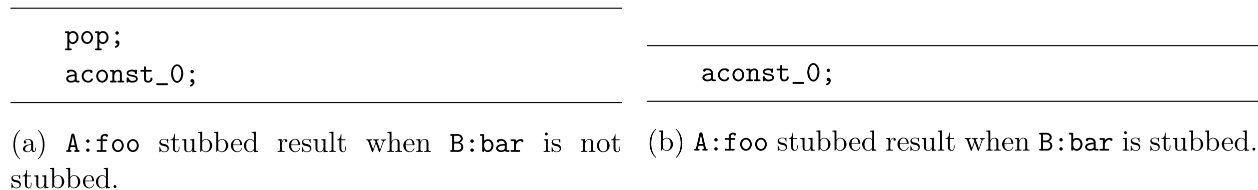
Figure 3.4: Dependency graph union process

Based on the nature of JVM bytecode, we can define stub function $S(r.e)$ for decompilers

as

$$S(r,e) = \text{remove(union argument instructions)} \tag{3.1}$$
$$= \text{pop(non-union arguments)} \tag{3.2}$$
$$= \text{insert(return type instruction)} \tag{3.3}$$

Figure 3.5 demonstrates the stubbed result for `A:foo`. The first argument is always unioned with `A:foo`, so no `pop` instruction is needed. When `B:bar` is not stubbed, we need to pop that argument, otherwise we can just ignore that since `B:bar` is stubbed and removed along with `A:foo`.



```
pop;
aconst_0;
```

(a) `A:foo` stubbed result when `B:bar` is not stubbed.

```
aconst_0;
```

(b) `A:foo` stubbed result when `B:bar` is stubbed.

Figure 3.5: `A:foo` stubbed result.

3.2 Multi-option Binary Reduction

Binary reduction [KP19] provides an efficient way to reduce monotonic sequence by binary searching. However, each reduction item in the sequence has only two options: exist or removed. We extend the process to handle multiple options for a single reduction point by forcing options to be monotonic.

For each reduction point (r_i, e_i), we define reduction group $G(r_i) = g_0, \cdots, g_{n_i}$ representing the options corresponding to the reduction point. For example, a function call could be preserved, inlined or removed. To apply multi-option binary reduction, we assume the reduction group to be monotonic according to predicate P:

$$\forall q, r \in \{0, \cdots, n\}, q \geq r \Rightarrow P(g_r) \leq P(g_q)$$

namely means the latter options are always more likely to preserve the program errors than the former ones. We define helper function M

$$M(G, r, D) = \begin{cases} \max i \text{ st. } i > 0 \land g_i \in G(r) \land g_i \in D & \text{if } \exists g_i \in D \\ g_0 & \text{otherwise} \end{cases}$$

returns the choice of the option given the searching sequence and reduction group. If none of the other options are present, we use base g_0 as our option. We also define a helper predicate P_h receiving group function, reduction points, and option sequence and returning the predicate result:

$$P_h(G, R, D) = P(\{M(G, r, D) : r \in R\})$$

Algorithm 1: Multi-option Binary Reduction.

Input: Monotonic reduction point and environment search space R
Reduction group function $G : r \rightarrow \text{set}\langle r \rangle$
Predicate $P : r^{|R|} \rightarrow \{T, F\}$ where $P(\{r_1, \cdots, r_I\}) = T$
Output: Failure inducing set of reduction options F
Data: A current failure inducing set $F \leftarrow \emptyset$
Current sorted search space $D \leftarrow \texttt{flatten}\{G(r_i) \setminus \{g_0\} : i \in \{1, \cdots, |R|\}\}$

```
while ¬P_h(G, R, F) do
    q ← min q st. q > 0 ∧ P_h(G, R, F ∪ {D_j : j ≤ q})
    F ← F ∪ {D_q}
    D ← {D_q : j < q}
end
return F
```

The idea behind Algorithm 1 is to flatten all options into a sequence and do binary reduction on the sequence. Since the option group are monotonic, the total sequence is also monotonic. We use naive `flatten` function here since, for common cases making option groups close to each other can help the total running time. If the options are likely to

be reduced, we can switch to a round-robin way to add options and keep the sequence monotonic.

3.2.1 Application on Java Programming Language

Thanks to the single inheritance nature of the Java programming language, we observe a natural application of multi-option binary reduction: method arguments. When a method parameter has `java.lang.Object` as its ancestor, we can choose some of its parent types as reduction options.

To avoid invalid upcasts of the method parameters, some preprocessing is required. First, we need to taint the instructions in the method body based on the potential method arguments they use. Second, we should restrict the parent candidates based on the potential interacting instructions.

Taint step If an `aload` instruction loads a parameter, we taint the instruction with the color corresponding to the parameter. Since the local argument slot in JVM is fixed, the only propagators are `dup` instructions. After running this step until a fixed point, we have the information about which instructions are sources of the local arguments.

Restrict step From the method invoking and field manipulation instructions in the method body, we can soundly restrict the argument parameter. Figure 3.6 and 3.7 demonstrates the restrictions we pose on the argument types. Note that we do over-approximation on type casting and arrays by forcing the argument type not to be reduced because of the nature of the Java language. After collecting all restrictions, we examine the lowest superclass of the argument type and choose it as our reduction base. The other superclasses become our reduction group options.

I(C)	reversed inheritance chain of class C
$\mathrm{F}_o(C, N)$	first class type in $I(C)$ contains the field named N
$\mathrm{F}_t(C, N)$	the type of the field named N in $F_o(C, N)$
$\mathrm{M}_o(C, N, D)$	first class type in $I(C)$ contains the method named N with description D
$\mathrm{M}_t(C, N, D, i)$	i-th argument type of the method refered in $M_o(C, N, D)$
P(V)	Boolean result of if the value V is tainted
T(V)	the type of the argument corresponding to the tainted color of value V
$\mathrm{T}_o(V)$	the original type of the argument corresponding to the tainted color of value V

Figure 3.6: Denotations used in Figure 3.7

$$\begin{aligned}
\text{putstatic}(C, N, V) &\Rightarrow P(V) \Rightarrow T(V) <: F_t(C, N) \\
\text{getfield}(C, N) &\Rightarrow P(C) \Rightarrow T(C) <: F_o(C, N) \\
\text{putfield}(C, N, V) &\Rightarrow (P(C) \Rightarrow T(C) <: F_o(C, N)) \wedge (P(V) \Rightarrow T(V) <: F_t(C, N)) \\
\text{invoke(C, N, D, V1, ...)} &\Rightarrow (P(C) \Rightarrow T(C) <: M_o(C, N, D)) \wedge (P(V_i) \Rightarrow T(V_i) <: M_t(C, N, D, i)) \\
\text{checkcast}(C, V) &\Rightarrow T(V) <: T_c(V) \\
\text{instanceof}(C, V) &\Rightarrow T(V) <: T_c(V) \\
\text{aastore}(A, I, V) &\Rightarrow T(V) <: T_c(V)
\end{aligned}$$

Figure 3.7: Restrictions derived from instructions.

CHAPTER 4

Evaluation

This section presents an empirical evaluation of using stubbed binary reduction and multi-option binary reduction to break dependencies in Java bytecode. We have implemented those techniques as an external plugin for Java reducer J-Reduce [Kal20]. In the experience, we will take J-Reduce output bytecode and do further bytecode modifications to allow further reduction in J-Reduce.

4.1 Experiment Setup

Benchmarks We use the benchmarks and artifacts of the J-Reduce [Kal20] project, which are collections of 222 Java bytecode instances where the decompilers can produce the Java source code, but the code fails to compile. The artifact targets on three most popular Java decompilers: CFR [Ben, version 0.132], Fernflower [Sc, commit 8be977e76] and Procyon [Str, version 0.5.30].

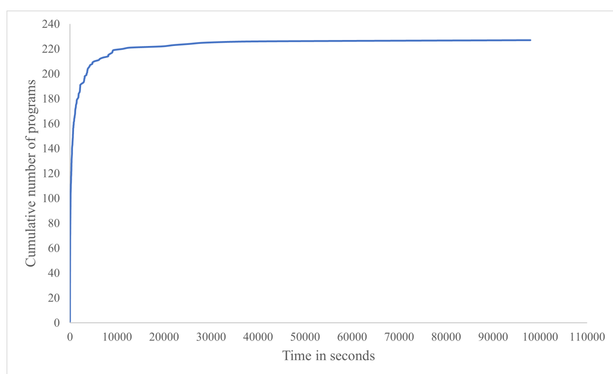
Implementation Our implementation is written in the Java language and leverages the ASM [asm, version 9.0] library for stack argument dependency analysis and other static analyses. We use the latest release of JReduce [Kal, commit ff4bc47] binary. The whole process works as follows: we take the output bytecode of JReduce and run our stubbed binary reduction, then feed them back to JReduce for final reduction; after that, we again provide the output after stubbed binary reduction and JReduce for our multi-option binary reduction implementation, and finally use JReduce to reduce unused dependencies exposed

by our modifications. We find that doing multi-option binary reduction alone is nearly useless since without function calls reduced, the parameters are highly likely to be preserved.

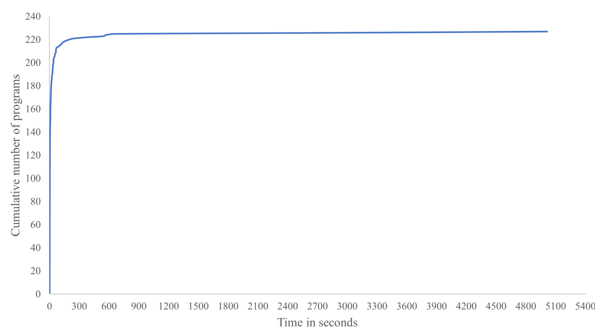
Machine The whole experiment is conducted on a machine with four Intel 2.0GHz i5 CPUs and 16 GB memory. Note that though the reducer is single-threaded, the decompilers will automatically take all CPUs and the majority of the time is the cost of running decompiler predicates.

4.2 Statistics

Time The time cost in reduction is highly dependent on the complexity of input and the running time of the predicate. The number of reduction points is proportional to the number of running time if the predicate runs in a nearly fixed time. Figure 4.1 demonstrates the distribution of time cost. The median time cost is 225.916 seconds for a stubbed binary reduction on function calls and 3.187 seconds for a multi-option binary reduction on method parameters. As we expect, the complexity of reduction for method parameters is much smaller than the complexity for reducing function calls, as we have many restrictions on method parameters, and the method definitions are often restricted to accept few arguments.



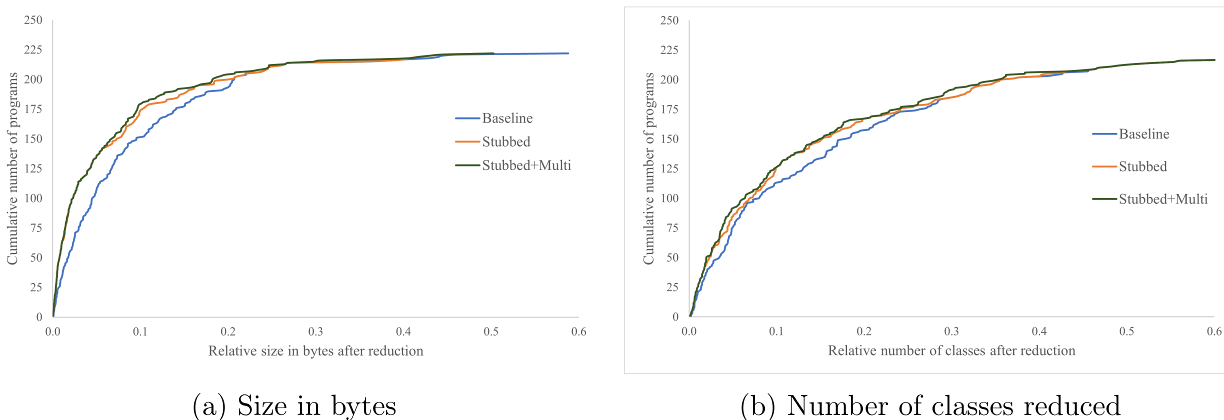
(a) Stubbed Binary Reduction



(b) Multi-option Binary Reduction

Figure 4.1: Cumulative frequency diagrams of time cost in reduction. The charts show the number of instances that terminate within x seconds. Higher is better.

Size We collect the metrics in two ways, both in the number of classes and size in bytes after reduction. We choose the artifact from JReduce [Kal20] project as our baseline. Figure 4.2 and Figure 4.3 demonstrate the cumulative results of size reduction for stubbed binary reduction and it plus multi-option binary reduction. Table 4.1 and Table 4.2 provide the overall results on the size reduced. We notice that multi-option binary reduction is good at reducing the number of classes as we anticipated. There are still some classes not reducing anymore after JReduce operation. After investigation, we find these instances are small, and it is hard to use a general way to reduce those cases.



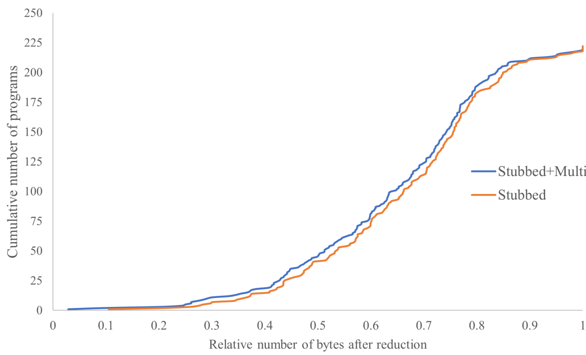
(a) Size in bytes

(b) Number of classes reduced

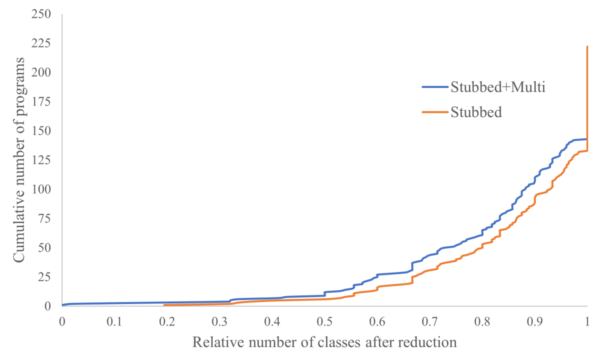
Figure 4.2: Cumulative frequency diagrams of size reduced after reduction relative to inital binary size. The charts show the number of instances that reduce x percent of size after reduction. Higher is better.

	Baseline (JReduce)	Stubbed	Stubbed + Multi
Median (bytes)	5.1%	3.3%	2.8%
Overall (bytes)	4.6%	2.9%	2.8%
Median (classes)	9.7%	8.1%	7.7%
Overall (classes)	8.4%	7.1%	6.8%

Table 4.1: Aggregated results of size reduced. The percent represents the relative size of the final output compared to the initial binary.



(a) Size in bytes



(b) Number of classes reduced

Figure 4.3: Cumulative frequency diagrams of size reduced after reduction relative to baseline. The charts show the number of instances that reduce x percent of size after reduction compared to the baseline artifact. Higher is better.

	Stubbed	Stubbed + Multi
Median (bytes)	69.0%	67.3%
Overall (bytes)	63.8%	61.1%
Median (classes)	94.4%	90.1%
Overall (classes)	86.1%	82.8%

Table 4.2: Aggregated results of size reduced relative to baseline. The percent represents the relative size of the final output compared to the baseline artifact.

CHAPTER 5

Conclusion

This thesis presents two new approaches to break dependencies for failure-inducing input programs. The stubbed binary reduction breaks closely connected dependencies into individual components, and the multi-option binary reduction extends the binary reduction algorithm to enable monotonic options within a single run of reduction. We implement and evaluate our algorithms on JVM bytecode and decompilers, and they show the potential to beat state of the art bytecode reducer.

Our implementation now works as a plugin for existing reducers, though they can be potentially integrated. When we remove some dependencies by stubbing function calls or replacing method parameters in the program, we can simultaneously remove some dependencies and some entities in the bytecode instead of reducing bytecode in separate two runs. Furthermore, we can test our methods on normal programs and predicates, which will work as a code debloater. Meanwhile, dependencies could be unclear for program languages like pointers in C. In order to generalize our methods, we may need a general representation of semantics and dependencies similar to a general representation proposed by Perses [SLZ18].

Though applications on different languages and different predicates are still to be done, we believe our methods demonstrate the potential of fine-grained structural testcase reduction in how they can break the closely connected dependencies and dig out more reducing possibilities.

REFERENCES

[asm] "ASM: A Java bytecode engineering library." `https://asm.ow2.io/index.html`. Accessed: May 2021.

[Ben] Lee Benfield. "CFR - another Java decompiler." `http://www.benf.org/other/cfr/`. Accessed: May 2021.

[bug] "GCC Bugzilla." `https://gcc.gnu.org/bugzilla/`. Accessed: May 2021.

[Kal] Christian Gram Kalhauge. "JReduce." `https://github.com/ucla-pls/jreduce`. Accessed: May 2021.

[Kal20] Christian Gram Kalhauge. *Reporting Bugs in Metaprograms*. PhD thesis, University of California, Los Angeles, `https://escholarship.org/`, 2020.

[KP19] Christian Gram Kalhauge and Jens Palsberg. "Binary Reduction of Dependency Graphs." In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, p. 556–566, New York, NY, USA, 2019. Association for Computing Machinery.

[Lar15] Michael Larabel. "GCC Soars Past 14.5 Million Lines Of Code amp; I'm Real Excited For GCC 5." `https://www.phoronix.com/scan.php?page=news_item&px=MTg3OTQ`, 2015. Accessed: May 2021.

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler." *Commun. ACM*, **52**(7):107–115, July 2009.

[LLP19] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. "DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing." *Proceedings of the AAAI Conference on Artificial Intelligence*, **33**(01):1044–1051, Jul. 2019.

[MS06] Ghassan Misherghi and Zhendong Su. "HDD: Hierarchical Delta Debugging." In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, p. 142–151, New York, NY, USA, 2006. Association for Computing Machinery.

[RCC12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-Case Reduction for C Compiler Bugs." In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, p. 335–346, New York, NY, USA, 2012. Association for Computing Machinery.

[RH01] S. Rayadurgam and M. P. E. Heimdahl. "Coverage based test-case generation using model checkers." In *Proceedings. Eighth Annual IEEE International Conference*

and Workshop On the Engineering of Computer-Based Systems-ECBS 2001, pp. 83–91, 2001.

[Sc] Roman Schevchenko and other contributors. "Fernflower." `https://github.com/fesh0r/fernflower`. Accessed: May 2021.

[SLZ16] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. "Toward Understanding Compiler Bugs in GCC and LLVM." In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, p. 294–305, New York, NY, USA, 2016. Association for Computing Machinery.

[SLZ18] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. "Perses: Syntax-Guided Program Reduction." In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, p. 361–371, New York, NY, USA, 2018. Association for Computing Machinery.

[Str] Mike Strobel. "Procyon Java decompiler." `https://github.com/mstrobel/procyon`. Accessed: May 2021.

[ZH02] A. Zeller and R. Hildebrandt. "Simplifying and isolating failure-inducing input." *IEEE Transactions on Software Engineering*, **28**(2):183–200, 2002.