RESCIENCE C

Replication / Program Debloating

# [Re] Effective Program Debloating via Reinforcement Learning

Denis Maurel[1], Jérôme Fillioux[1], and Dan Gugenheim[1]
[1]onepoint sud-ouest RD, Bordeaux, France

**Edited by**
Olivia Guest

**Reviewed by**
Wouter Kouw
Esther Mulwa

**Received**
10 February 2021

**Published**
28 March 2022

**DOI**
10.5281/zenodo.6255131

**A partial replication of**
-> K. Heo et al. "Effective Program Debloating via Reinforcement Learning." en. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto Canada: ACM, Jan. 2018, pp. 380–394.

## 1 Introduction

Many factors can lead to the apparition of useless code: the reuse of previously developed code, the import of entire libraries when only a subset of their functionalities is required, or even the simple evolution of the project through its development. This phenomenon has to be considered along with the global and dramatic increase in software size and complexity, which therefore increases the probability of apparition of useless code. A code with parts whose absence does not change the whole software behaviour is said to be bloated.
While usual static analysis tools can efficiently find unused code, it is far more difficult to identify code which is actually traversed but useless in practice. How can one determine if a portion of the code is useful without running the code itself ? That's why useless code has to be distinguished from unused (or dead) code.
In this context, methods such as Delta Debugging [2] along with its improved hierarchical version [3] has been designed to automatically identify which portion of the code is useful or not. Removing useless code can improve code maintainability and readability while preventing external attack using undetected vulnerabilities [1].
In this paper, we have tried to reproduce the results presented by [1] which introduced a version of the previously mentioned algorithms [3] improved through the use of Reinforcement Learning (RL) [4]. RL is here meant to provide an effective method to understand the underlying code structure during the debloating process. It has been selected because it integrates itself naturally in the "trial and error" based debloating process, while other traditional Machine Learning methods would have required a training phase taking place before the debloating process.
At each step of the process, the debloating method tests one subprogram among a pool of candidates. The longer the original program, the bigger the size of the pool of candidates. The point of the new RL part of the process is to improve the global efficiency of the method by ranking the potential subprograms to test depending on their chances to both pass the test and to be minimal. A code is minimal when there is no more useless


Correspondence should be addressed to Denis Maurel (d.maurel@groupeonepoint.com)
The authors have declared that no competing interests exists.
Code is available at https://gitlab.com/onepoint/recherche-et-developpement/optimus/chisel-replication – DOI 10.5281/zenodo.4530318.
Open peer review is available at https://github.com/ReScience/submissions/issues/48.

code in it. While the results obtained by the authors clearly surpass those obtained by state-of-the-art tools such as C-REDUCE [5] and PERSES [6] when debloating C code, the article lacks a numerical comparison of the improvements obtained by their Reinforcement Learning part.
It has to be noted that the aim of this paper is not to clarify if the results presented by the original paper can be reproduced. The point is rather to demonstrate that the algorithm can be reproduced while completing some caveats in the its original description, namely the necessity to have a two phases algorithm, the debloating process requiring runs to get a truly minimal result and the special case of the hyperparameter $\gamma = 0$. The analysis of its results is then extended with a focus on the impact of the RL part of the method.

# 2 Methods

## 2.1 Reproductibility of the original results

While authors of the original paper present results of their algorithm on C code, we have taken the liberty to test the method on Python programs. This choice has been motivated by the hypothesis that the improvements brought by the RL part were agnostic regarding the programming language being debloated, the definition of the algorithm not involving the use of language specific features. Here, improvements are considered in terms of number of saved algorithm steps.
The method being reproduced comes from an extensive reading of the article along with some clarifications provided by the authors of the original paper. The source of the program provided by the original[1] paper have not been used for its reproduction.
Here are the specifications used for the experiments :

- the program has been developed using Python 3.8.
- the programs being debloated were compatible with Python 3
- the computer used to perform the tests had the following specifications: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz and 16GB of RAM

## 2.2 Description of the Data

The original paper applied its method on 10 UNIX utility programs, taking `tar-1.14` as a motivating example. It also uses an arbitrary script as the property test function to determine if a program is valid or not. If a generated program is valid, the arbitrary script returns the boolean value true, and false otherwise. In this paper, we suggest the use of the unit tests (UT) already available in the project folder as an additional way to test the validity of a program. If a program passes all its UT, it is considered valid. On the opposite, if it fails at least one UT, it is considered as invalid.
It has to be noted that the results obtained using UT are linked to the quality of the test suite. If the test coverage is not sufficient, or if the tests quality is not good enough, the debloating process may remove essential parts of the program. The assertion of test quality being a research domain in itself [7], it will not be detailed in this article.
This choice of running UT is motivated by the fact that they are defined to represent the low-level behavior the software is supposed to have. This definition is well aligned with the software debloating problem definition. If a program passes all its UT, all its low level features are supposed to be usable. However, as being previously said, this approach requires the software test suite to cover all the required functionalities.
Another motivating point of this approach is that, as python is an interpreted language, all the time usually spent to compile C programs can be spent to run the UT. The final

[1] https://github.com/aspire-project/chisel

**Table 1**. Python libraries to debloat

| name | version | testing method |
|---|---|---|
| EXTRAS | 1.0.0 | unit tests (22) |
| TYPED_AST | 1.4.1 | unit tests (15) |
| STEVEDORE | 2.0.1 | unit tests (84) |
| SCIKIT-LEARN | 0.23.1 | arbitrary script (2) |
| ASTOR | 0.8.1 | arbitrary script |

advantage of this approach is that one does not have to develop an additional test script to valid a program behavior.
To ensure that the debloating process can be done in an acceptable amount of time, the software candidates debloated in this paper will be some python libraries with test suites running in less than ten minutes. In order to compare both approaches and to assert the original method reproductibility, some tests have also been performed using the original property testing script method. This latter method will be preferred for libraries with too many UT to be tested in a reasonable amount of time.
The target libraries along with the method used to test their validity can be found in Table 1. Regarding the debloating process of SCIKIT-LEARN[2] and ASTOR[3]

- **SCIKIT-LEARN**: the point is to demonstrate that a big proportion of the library can be debloated when one uses only a single model. As a consequence, two testing scripts are defined : the first one training a linear regression model and the second one training a DBSCAN model. The point of those tests is to determine how efficient the system is dealing with a code base with an important proportion of useless code (regarding the testing script).

- **ASTOR** : ASTOR is a library used by our implementation to translate an Abstract Syntax Tree object (obtained with the ast python module) into actual code. The point of this experiment is to determine what is the minimal code which can be used by our implementation to effectively debloat a program. The program debloated by the test function is a toy problem in which a single file contains a useless function which has to be removed. A program passes the test if the debloated file obtained by the property testing script is the same as the one obtained with the complete ASTOR library.

## 2.3 Clarifications on the debloating process

**Two phases algorithm —** In this section are presented some clarifications on the exact steps of the original algorithm. This is the result of an exchange with the authors of the original paper. While not being specified by the authors, the points presented here have been used to get the results of the original algorithm.
The main advantage provided by the hierarchical structure introduced by the Hierarchical Delta Debugging algorithm [3] is to split the whole debloating process into subprocesses, each one being responsible for a level in the tree. However, besides mentioning that their model is based "on the syntax-guided hierarchical delta debugging algorithm" [1], the authors do not specify in their pseudo code how the hierarchical component is used in practice.
The debloating process is actually made of two phases: the first one to select code elements which have to be kept, and the second one to debloat each element separately. In this context, a code element is the definition of a variable, a class definition or a method/function. By first selecting which elements have to be kept, the algorithm skips

[2] https://scikit-learn.org/stable/
[3] https://github.com/berkerpeksag/astor

the debloating of all the elements which are useless by definition. When the elements are selected, the algorithm performs a local debloating on each element. The idea behind this is that the code of a method does not rely on the code of another one to work. This dramatically reduce the number of steps the algorithm takes to globally debloat a file.

**Multi run debloating —** The use of a hierarchical structure (also known as a tree) for the debloating process has a side effect which is not mentioned in the original article regarding the order in which the elements are analyzed. Indeed, an element of a higher level in the tree might be kept because it is used by an other element deeper in the hierarchy. However, the algorithm might find that this deeper element is actually useless, and so removes it. While this deeper element has been removed, the higher one still remains, and is no longer necessary. To be sure to remove it, the algorithm has to be run again on the first debloated code. A toy problem subject to this problem is presented on Fig.1.

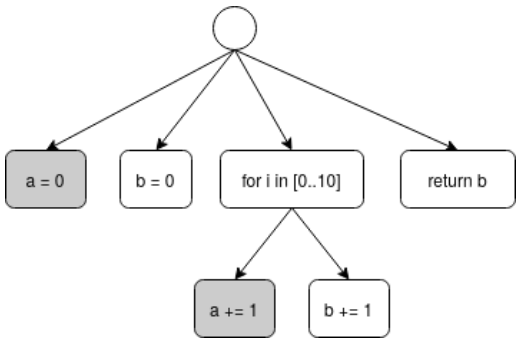


**Figure 1.** Small program subject to the multi run problem. While both shaded nodes are useless, only the one present in the for loop will be removed by the system. This comes from the fact that when the variable definition node is analysed, it can't be removed because the variable it defines is used by the second shaded node. The second node being deeper in the tree, the first one will not be analyzed after the second has been removed.

A consequence of this point is that if one wants to ensure that the code is fully debloated, the process has to be run at least 2 times : one to debloat the program, and another one to ensure that there is nothing left to remove. In practice, the first run always remove the majority of the useless code. Because of this latter points, the following experiments have been made with only one application of the algorithm, the underlying hypothesis being that the impact of the use of RL was independent of the number of applications of the algorithm.

**Reinforcement Learning —** We think that a clarification has to be made regarding the use of Reinforcement Learning. The authors mentions that their best performances is obtained when setting $\gamma = 0$. The point of the $\gamma$ parameter is to define the range of rewards taken into account by the system. The higher the value of $\gamma$, the farther the system will "look" in the future. Setting $\gamma = 0$ defines a very specific case of the Reinforcement Learning paradigm for which the authors of [4] say that "the agent is myopic in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose [the action at time $t$] so as to maximize only [the reward at time $t + 1$]". This configuration can be seen as a multi-armed bandit, where the reward is being maximized after each action. This configuration has the advantage of being cost efficient, in that it does not require further computation to consider future rewards. As a trade-off, it lacks the long-term view of the original Reinforcement Learning method, thus prefering exploitation to exploration. It may be important to note that the value $\gamma = 0$ has been set by the authors using experimental results regarding their method efficiency.

**Table 2.** Results of the debloating program with and without Reinforcement Learning regarding the number of iterations taken and the total process duration.

| | Number of steps | | | Process duration | | |
|---|---|---|---|---|---|---|
| name | RL | no RL | Gain | RL | no RL | Gain |
| EXTRAS | 219 | 227 | **-3.4%** | 57.7 | 62.4 | **-7.5%** |
| TYPED_AST | 188 | 194 | **-3.3%** | 41.1 | 42.8 | **-3.9%** |
| STEVEDORE | 1110 | 1163 | **-4.6%** | 339.0 | 351.7 | **-3.6%** |
| SCIKIT-LEARN (linear regression) | 3870 | 4466 | **-13.4%** | 252.1 | 275.3 | **-8.4%** |
| SCIKIT-LEARN (DBSCAN) | 7617 | 8450 | **-9.9%** | 549.9 | 591.3 | **-7.0%** |
| ASTOR | 1479 | 1624 | **-8.9%** | 109.6 | 113.4 | **-3.3%** |

# 3 Results

## 3.1 Reproducibility

With the exception of the points mentioned beforehand which have been easily solved with the help of the authors, the algorithm has been quite straightforward to reproduce. Here again, it is useful to note that the point of this paper is not to reproduce the results of the original papers, but rather to complete them by exploring some missing points. Because the training of the Decision Trees [8] used by our implementation of the algorithm is not totally deterministic, each debloating library has been debloated 10 times and the results have been averaged. For the version without RL, only one run has been performed because, in that case, the algorithm is totally deterministic. The Decision Tree algorithm used here is the one provided by the SCIKIT-LEARN library[4].

## 3.2 Results

Because no statement is made in the original paper regarding the gain in terms of number of iteration and duration, an acceptable hypothesis would be that the RL based program will reduce the number of steps taken for an optimization, as presented in the original paper on the toy example. However, the gain in terms of time consumption is not as clear. The time used to update and use the model may exceed the time from the saved iterations.
The results of the experiments are presented in Table 2.
It appears that the version of the program with Reinforcement Learning reduces both the number of steps and time consumption in every tested case. The top scores in terms of number of steps are obtained for SCIKIT-LEARN and ASTOR, both examples being debloated using arbitrary scripts. While the underlying explanation of this phenomenon is not obvious, we set the hypothesis that the RL agent learns more quickly which part of the code is or is not important using arbitrary test script. This hypothesis is motivated by the fact that an arbitrary script covers a smaller proportion of the target code compared to a complete test suite. This point allows the agent to focus earlier on mandatory parts of the code, improving its predictions sooner compared to an agent trained using UT.
While those results are useful to get a first overview of the possible improvements, it is important to note that the process duration improvements are relative to the Decision Tree algorithm implementation which is used. For these experiments, the Decision Tree implementation provided by the SCIKIT-LEARN library has been used.

[4] https://SCIKIT-LEARN.org/stable/modules/tree.html

# 4 Conclusion

The algorithm presented in [1] introduces a new Reinforcement Learning based part to the original Hierarchical Delta Debugging [3] which is intended to limit the number of tests to run by training a model to prioritize the programs to be tested.
This paper presents some clarifications on three points of the original algorithm. The first is that the method implements two distinct phases, the first one to debloat high-level components and the second one to debloat each component individually. The second point specifies that the algorithm has to be run at least two times to ensure that the code obtained is actually minimal. Finally, the third point precises that taking $\gamma = 0$ is a very particular case of Reinforcement Learning for which the trained agent can be likened to a multi-armed bandit. This paper also introduces the use of unit tests as a property test function.
While no results has been presented in the original paper regarding the numeric gain either in terms of number of steps or time consumption, the algorithm is presented to outperform industrial debloating tools such as C-Reduce [5] and Perses [6].
This paper completes the original one by providing numeric proves of the gain obtained by the newly defined Reinforcement Learning part. It has to be noted that, because experiments were conducted using a different language and different softwares, this paper only reproduces the model architecture and not the model experimental results. The experiments show that the algorithm is both reproducible using some further clarifications and improving the original Hierarchical Delta Debugging.

# References

1. K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. “Effective Program Debloating via Reinforcement Learning.” en. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. Toronto Canada: ACM, Jan. 2018, pp. 380–394.
2. A. Zeller and R. Hildebrandt. “Simplifying and Isolating Failure-Inducing Input.” In: **IEEE Transactions on Software Engineering** 28.2 (2002), pp. 183–200.
3. G. Misherghi and Z. Su. “HDD: Hierarchical Delta Debugging.” en. In: **Proceeding of the 28th International Conference on Software Engineering - ICSE '06**. Shanghai, China: ACM Press, 2006, p. 142.
4. R. S. Sutton. **Reinforcement Learning**. English. 1992.
5. J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. “Test-Case Reduction for C Compiler Bugs.” en. In: (), p. 11.
6. C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su. “Perses: Syntax-Guided Program Reduction.” en. In: **Proceedings of the 40th International Conference on Software Engineering**. Gothenburg Sweden: ACM, May 2018, pp. 361–371.
7. H. Zhu, P. A. Hall, and J. H. May. “Software unit test coverage and adequacy.” In: **Acm computing surveys (csur)** 29.4 (1997), pp. 366–427.
8. L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. **Classification and regression trees**. CRC press, 1984.