Program Debloating via Stochastic Optimization
Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso
Georgia Institute of Technology
{qxin6, wardballoon, qrzhang}@gatech.edu, orso@cc.gatech.edu

ABSTRACT
Programs typically provide a broad range of features. Because different typologies of users tend to use only a subset of these features, and unnecessary features can harm performance and security, program debloating techniques, which can reduce the size of a program by eliminating (possibly) unneeded features, are becoming increasingly popular. Most existing debloating techniques tend to focus on program-size reduction alone and, although effective, ignore other important aspects of debloating. We believe that program debloating is a multifaceted problem that must be addressed in a more general way. In this spirit, we propose a general approach that allows for formulating program debloating as a multi-objective optimization problem. Given a program to be debloated, our approach lets users specify (1) a usage profile for the program (i.e., a set of inputs with associated usage probabilities), (2) the factors of interest for debloating, and (3) the relative importance of these factors. Based on this information, the approach defines a suitable objective function for associating a score to every possible reduced program and aims to generate an optimal solution that maximizes the objective function. We also present and evaluate Debop, a specific instance of our approach that considers three objectives: size reduction, attack-surface reduction, and generality (i.e., the extent to which the reduced program handles inputs in the usage profile provided). Our results, albeit still preliminary, are promising and show that our approach can be effective at generating debloated programs that achieve a good trade-off between the different debloating objectives considered. Our results also provide insights on the performance of our general approach when compared to a specialized single-goal technique.

ACM Reference Format:

1 INTRODUCTION
Modern software systems are increasingly complex and feature-rich. Typically, however, different users use different features, and a significant fraction of these features are rarely used at all [13]. Because this unneeded functionality can severely harm performance, cost energy, and raise security issues [24], software debloating techniques, which can remove such unnecessary features, are becoming increasingly popular. Most existing debloating techniques (e.g., [12, 23]) take as input a (bloated) program \( p \) and a set of inputs \( I \) for \( p \), and their only goal is to produce the smallest reduced program \( p' \) that behaves correctly (i.e., like \( p \)) for \( I \).

We believe that this is a limited view of debloating. When performing debloating in a more realistic scenario, there would typically be different, possibly conflicting goals at play. For example, it may be acceptable for a debloated program to be unable to handle 10% of the inputs in \( I \) if that can result in an 80% reduction of its possible vulnerabilities. Similarly, there may be cases in which generality (i.e., the extent to which the reduced program can handle inputs in \( I \)) is the most important factor, and should therefore be pursued at the expense of program-size reduction. To account for these situations, and provide a more general approach to debloating, we propose a novel, general formulation of program debloating as a multi-objective optimization problem.

Our approach allows for specifying a debloating task in terms of the following elements: the program to be debloated, \( p \); a set of inputs for \( p \), possibly with associated usage probabilities (i.e., a usage profile); the factors of interest for the debloating task; and the relative importance of these factors, expressed as weights. Based on this information, the approach generates an objective function \( O \) that encodes the relevant factors and their weights. It then uses \( O \) to compute a score for every possible reduced program during the search for an optimal solution—the solution that maximizes \( O \).

We also present Debop, a specific instance of our general debloating approach that supports three goals: maximizing size reduction, minimizing attack surface (code that can be leveraged for attacks), and maximizing generality. In addition to \( p \) and a user profile for \( p \), Debop therefore takes as input weights that indicate the relative importance of the three goals and uses them to define \( O \).

Because it is generally infeasible to enumerate all possible reduced versions of \( p \), Debop cannot find an optimally debloated program through an exhaustive search. Instead, our technique leverages a Markov Chain Monte Carlo (MCMC) sampling technique [11] for effectively exploring the search space. The MCMC sampling algorithm, guided by the objective function \( O \), samples a number of reduced versions of \( p \) and reports the one with the best value for \( O \). This version, which we call \( p_{\text{deb}} \), is an approximation of the actual optimal solution.

To evaluate our approach, we performed a case study in which we used Debop to compute debloated versions of a Unix utility using different combination of weights for its three goals. This proof-of-concept evaluation confirms that Debop can effectively generate debloated programs that achieve good trade-offs in the presence of multiple, conflicting goals. We also applied a state-of-the-art, specialized debloating technique to the same Unix utility and compared the performance of the two techniques. The result of this second study show that, although Debop may pay a price for
its generality, it produces better overall results when considering multiple goals. Moreover, Debop can be improved along many dimensions, as we discuss when describing our future work.

This paper makes the following contributions:

- A novel, general formulation of program debloating as a multi-objective optimization problem that accounts for the presence of multiple, possibly conflicting debloating goals.
- An instance of our approach, Debop, that (1) supports multiple goals expressed in terms of program size, attack surface, and generality, and (2) computes approximated solutions to the (debloating) optimization problem via MCMC sampling.
- A proof-of-concept evaluation that shows the potential usefulness of our approach and provides insight on the trade-offs involved in program debloating.
- An implementation of Debop that is publicly available, together with our experiment infrastructure and data, for replication, at https://sites.google.com/view/debop19.

2 BACKGROUND

Debloating. Let \( p \) be a deterministic program and \( I \) be a set of unique inputs for \( p \) with associated usage probabilities (i.e., a usage profile for \( p \)). We use \( p(i) \) to denote the result of running \( p \) with input \( i \). Given \( p \) and \( I \), debloating is the process of removing code from \( p \) to produce a reduced program \( p' \). We say that \( p' \) can handle an input \( i \) if \( p(i) = p'(i) \).

Program Representation. A program \( p \) consists of a sequence of program statements: \( p = \{s_1, s_2, \ldots, s_n\} \). We define \( Sub(p) \) as the set of all possible reduced versions of \( p \). We represent each program \( p' \in Sub(p) \) as an \( n \)-bit bi-vector \( W(p') = b_1, b_2, \ldots, b_n \), where \( b_i = 1 \) when the corresponding statement \( s_i \) is present in \( p' \), and 0 otherwise. (For \( p, b_1 \) is 1 for \( 1 \leq i \leq n \).)

3 OUR TECHNIQUE: DEBOP

As we mentioned in Section 1, our general approach formulates program debloating as a multi-objective optimization problem defined in terms of \( p, I \), and a set of weighted goals. In this section, we present Debop, a specific instance of this general approach that (1) supports three goals expressed in terms of program size, attack surface, and generality, and (2) computes approximated solutions to the (debloating) optimization problem via MCMC sampling.

Debop characterizes a reduced program \( p' \in Sub(p) \) in terms of size reduction, \( sr(p, p') \), attack-surface reduction, \( ar(p, p') \), and generality, \( g(p, p') \), computed as:

\[ sr(p, p') = \frac{sz(p) - sz(p')}{sz(p)}, \]

\[ ar(p, p') = \frac{as(p) - as(p', p)}{as(p)}, \]

\[ g(p, p') = \frac{1}{1 + \exp(\beta \\cdot O(p, p'))} \]

where \( sz(p) \) is the size of \( p \) (e.g., in LOCs); \( as(p, p') \) measures the attack surface of \( p \) (e.g., in terms of number of gadgets [22]); \( pr_{ij} \) is the probability associated with the input \( i \in I \); and \( T(ik) \) is an indicator function whose value is 1 when \( p'(ik) = p(ik) \) and 0 otherwise. Debop aims to generate a reduced program \( p'_{deb} \) that maximizes size reduction, attack-surface reduction, and generality (with different weights for the different goals). To do so, it first computes a general measure of reduction, \( r(p, p') \), by combining \( sr(p, p') \) and \( ar(p, p') \):

\[ r(p, p') = (1 - \alpha) \cdot sr(p, p') + \alpha \cdot ar(p, p'), \]

where \( \alpha \) is a weight between 0 and 1 that denotes the relative importance of \( sr(p, p') \) and \( ar(p, p') \). It then computes an objective function \( O(p, p') \) defined as the weighted sum of reduction \( r(p, p') \) and generality \( g(p, p') \):

\[ O(p, p') = (1 - \beta) \cdot r(p, p') + \beta \cdot g(p, p'), \]

where \( \beta \) is also a weight between 0 and 1. We call the value this function computes the \( O \)-score. We can now formulate Debop’s debloating task as an optimization problem:

\[ p_{deb} = \arg \max_{p' \in Sub(p)} O(p, p') \]

By using different values for \( \alpha \) and \( \beta \), one can generate solutions with different trade-offs between \( sr(p, p'), ar(p, p'), \) and \( g(p, p') \).

Solving this optimization problem by enumerating each \( p' \in Sub(p) \) and identifying the one with the highest \( O \)-score is not feasible, given the exponential size of \( Sub(p) \). Therefore, Debop leverages stochastic search, and specifically a Markov Chain Monte-Carlo (MCMC) sampling method, to generate an approximate \( p_{deb} \).

MCMC & Metropolis-Hastings Algorithm. MCMC is a sampling technique [11] for estimating the properties of a distribution \( d \) by drawing samples from it. To use MCMC in our context, we need to define \( d \) and its density function. We can define the density function \( f(p, p') \) for \( d \) using the objective function \( O(p, p') \) as:

\[ f(p, p') = \frac{1}{Z} \exp(k \cdot O(p, p')), \]

where \( k \) and \( Z \) are constants [11, 20].

Our goal is to draw a sufficient number of samples \( S \) in proportion to \( d \) and infer properties of \( d \) based on \( S \). Intuitively, this means that more samples with higher density values (and thus higher \( O \)-scores) should be drawn from \( d \) than samples with lower density values. In our context, a sample is a reduced program \( p' \in Sub(p) \). To draw such samples, we use the Metropolis-Hastings (MH) algorithm [7] with the probability of accepting a new sample \( p'_{i+1} \) given a current sample \( p'_{i} \) defined as:

\[ A(p'_{i} \rightarrow p'_{i+1}) = \min \left( 1, \frac{f(p_{i+1}, p'_{i+1}) \cdot q(p'_{i+1}, p_{i+1})}{f(p, p') \cdot q(p_{i+1}, p'_{i+1})} \right), \]

(2)

where \( q(p'_{i+1}, p'_{i}) \) is the proposal distribution of transforming one sample \( p'_{i} \) into another \( p'_{i+1} \). When the transformation is symmetric (i.e., \( q(p'_{i+1}, p'_{i}) = q(p'_{i}, p'_{i+1}) \)), the acceptance probability becomes:

\[ A(p'_{i} \rightarrow p'_{i+1}) = \min \left( 1, \frac{\exp(k \cdot O(p, p_{i+1}))}{\exp(k \cdot O(p, p_{i}))} \right). \]

(3)

The general MH algorithm starts with an initial sample \( p'_{0} \). It then iteratively generates a new sample \( p'_{i+1} \) from the current one \( p'_{i} \) by mutating it, and compares the density values \( f(p'_{i+1}) \) and \( f(p'_{i}) \). If \( f(p'_{i+1}) > f(p'_{i}) \), it accepts \( p'_{i+1} \) and uses \( p'_{i+1} \) as the current sample. Otherwise, it can still accept \( p'_{i+1} \) by acceptance probability \( A(p'_{i} \rightarrow p'_{i+1}) \) to avoid being trapped in local maxima. If \( p'_{i+1} \) is rejected, \( p'_{i} \) will be used again as the current sample.

Algorithm 1 presents the specific MH algorithm used within Debop. The algorithm takes as input a program \( p \), the number of samples \( n \) to generate, and weights \( \alpha \) and \( \beta \), and generates as output a list of samples. The algorithm first obtains \( stms \), the program statements of \( p \) (line 1), sets the current sample \( ps \) as \( p \) (line 2) and...
We performed a case study to investigate two research questions:

Algorithm 1 Sampling algorithm.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textit{samples} \leftarrow get the statements of \textit{p}</td>
</tr>
<tr>
<td>2</td>
<td>\textit{ps} \leftarrow \textit{p}</td>
</tr>
<tr>
<td>3</td>
<td>\textit{dval} \leftarrow \textit{getDensityValue}(\textit{ps}, \textit{alpha}, \textit{beta})</td>
</tr>
<tr>
<td>4</td>
<td>\textit{samples} \leftarrow {}</td>
</tr>
<tr>
<td>5</td>
<td>\textit{i} \leftarrow 0</td>
</tr>
<tr>
<td>6</td>
<td>\textit{while} \textit{i} &lt; \textit{n} \textit{do}</td>
</tr>
<tr>
<td>7</td>
<td>\textit{stmt} \leftarrow \textit{randomly select one statement from \textit{stmts}}</td>
</tr>
<tr>
<td>8</td>
<td>\textit{if} \textit{ps} contains \textit{stmt} then</td>
</tr>
<tr>
<td>9</td>
<td>\textit{new}_{ps} \leftarrow \textit{deblock}(\textit{ps}, \textit{stmt})</td>
</tr>
<tr>
<td>10</td>
<td>\textit{else}</td>
</tr>
<tr>
<td>11</td>
<td>\textit{new}_{ps} \leftarrow \textit{severstips}(\textit{stmt})</td>
</tr>
<tr>
<td>12</td>
<td>\textit{new}<em>{dval} \leftarrow \textit{getDensityValue}(\textit{new}</em>{ps}, \textit{alpha}, \textit{beta})</td>
</tr>
<tr>
<td>13</td>
<td>\textit{accept} \leftarrow \textit{false}</td>
</tr>
<tr>
<td>14</td>
<td>\textit{r} \leftarrow \textit{random()}</td>
</tr>
<tr>
<td>15</td>
<td>\textit{if} \textit{r} &lt; \textit{(new}_{dval}/\textit{dval}) then</td>
</tr>
<tr>
<td>16</td>
<td>\textit{accept} \leftarrow \textit{true}</td>
</tr>
<tr>
<td>17</td>
<td>\textit{if} \textit{accept} then</td>
</tr>
<tr>
<td>18</td>
<td>\textit{samples} \leftarrow \textit{samples} \cup \textit{(new}_{ps})</td>
</tr>
<tr>
<td>19</td>
<td>\textit{ps} \leftarrow \textit{new}_{ps}</td>
</tr>
<tr>
<td>20</td>
<td>\textit{dval} \leftarrow \textit{new}_{dval}</td>
</tr>
<tr>
<td>21</td>
<td>\textit{i} \leftarrow \textit{i} + 1</td>
</tr>
<tr>
<td>22</td>
<td>\textit{return} \textit{samples}</td>
</tr>
</tbody>
</table>

computes its density value \textit{dval} (line 3), and initializes the result list \textit{samples} and a sample counter \textit{i} (lines 4 and 5).

The algorithm then generates samples iteratively (lines 6–21). At each iteration of the loop, it mutates the current sample \textit{ps} to generate a new sample \textit{new}_{ps}. It does so by (1) randomly selecting a program statement \textit{stmt} from \textit{stmts} and (2) either removing \textit{stmt} from \textit{ps} or adding \textit{stmt} back to \textit{ps} (lines 7–11). Because the transformation is symmetric, as each program statement has the same chance of being selected for removal or add-back, the algorithm uses the simplified equation 3 for computing the acceptance probability. After generating the new sample \textit{new}_{ps}, the algorithm computes its density value \textit{new}_{dval} (line 12) and compares it to the current \textit{dval} to decide whether to accept \textit{new}_{ps} (lines 13–16). If \textit{new}_{ps} is accepted, the algorithm saves it into \textit{samples} and updates \textit{ps}, \textit{dval}, and the sample counter \textit{i} (lines 17–21). Finally, the algorithm returns the list of samples, among which \textit{Debop} selects as \textit{pdeb} the sample with the highest \textit{O-score}.

4 PROOF-OF-CONCEPT EVALUATION

We performed a case study to investigate two research questions:

- **RQ1**: Does \textit{Debop} generate debloated programs with different size reduction, attack-surface reduction, and generality trade-offs, when provided with different \textit{alpha} and \textit{beta} values?
- **RQ2**: How does \textit{Debop} compare with a specialized, single-goal debloating approach?

**Implementation.** We implemented a prototype of \textit{Debop} in C++. \textit{Debop} uses Clang [3] to build an AST (abstract syntax tree) for \textit{p} and identify \textit{p}'s statements. To measure the size of a program, \textit{Debop} compiles it (with GCC v. 7.4.0, -O3 option) and counts the number of bytes in the resulting executable. Then, to measure the attack surface, \textit{Debop} counts the number of ROP (Return-Oriented Programming) gadgets [22] in the executable using the ROPGadget tool [5]. An ROP gadget is a sequence of machine instructions that can be exploited for ROP attacks [6].

**Experiment Setup.** As a benchmark for our case study, we used \textit{mkdir} (version 5.2.1, ~28 KLOCs), a Unix utility for creating new directories used in related work [12]. As inputs (i.e., usage profile) for \textit{mkdir}, we used the 26 tests provided with the program plus 2 tests provided with its BusyBox version [1]. (BusyBox contains a stripped-down version of several Unix utilities.) We used a timeout of 1s for each test run, to handle cases in which a reduced program did not terminate. For simplicity, we assigned equal probability to all inputs. Based on early experimentation, we configured Algorithm 1 so that it generated 1,000 samples and used \textit{k} = 50 for computing density values.

For RQ1, we used \textit{Debop} to debloat \textit{mkdir} using all provided inputs and in multiple trials with different values for \textit{alpha} (0.25, 0.5, and 0.75) and \textit{beta} (0.1, 0.2, ..., 0.9), for a total of 27 combinations. We did not include the two extreme cases of \textit{beta} being 0 and 1. When \textit{beta} is 0, \textit{Debop} would simply return an empty program. Similarly, when \textit{beta} is 1, \textit{Debop} would simply return the original program. We also did not include the two cases of \textit{alpha} being 0 and 1, so as to consider both size reduction and attack-surface reduction when debloating. It took \textit{Debop} on average 2.3 hours to finish each trial.

To the best of our knowledge, \textit{Debop} is the first multi-objective debloating technique, so there are no ideal baseline techniques. Nevertheless, to get a better understanding of \textit{Debop}'s strengths and weaknesses, in RQ2 we compared it with a specialized, single-goal debloating technique. Specifically, we selected \textit{Chisel} [12], a state-of-the-art debloating technique based on delta debugging, and used its existing implementation [2]. \textit{Chisel} can only generate a reduced program that handles all the provided inputs. Therefore, to perform a fairer comparison, for each of the 27 trials, we (1) logged the exact set of inputs that \textit{Debop} could correctly handle and (2) provided \textit{Chisel} with the same inputs when running it within that trial. In other words, we ran \textit{Chisel} on the set of "optimal" inputs that were identified as part of \textit{Debop}'s solution.

4.1 Results: RQ1

Figure 1 shows \textit{Debop}'s results for \textit{alpha} values from 0.25 (left) to 0.75 (right). As \textit{alpha} increases (i.e., attack-surface reduction is weighted more), the reduction score (triangle.gray line) gets closer to the attack-surface reduction score (box/orange line). Also, when \textit{beta} increases from 0.1 to 0.9 (i.e., generality is weighted more), the generality score increases, while the reduction score decreases. These results confirm that \textit{Debop} indeed allows for exploring different trade-offs while debloating.

There are cases, however, in which generality decreases and/or reduction increases when \textit{beta} increases (e.g., when \textit{alpha} = 0.25 and \textit{beta} changes from 0.4 to 0.5). This could be due to the fact that some test inputs cover more program statements (i.e., some inputs are more "important"). When \textit{beta} is less critical (e.g., \textit{beta} = 0.5), omitting those test inputs but increasing the reduction has a more significant impact on improving the \textit{O-score}. Also, reduced programs with less statements do not necessarily handle less inputs than reduced programs with more statements (e.g., if a failing path is added).

4.2 Results: RQ2

Table 1 shows the results of comparing \textit{Debop} and \textit{Chisel} in terms of size and attack-surface reduction of the debloated programs they generate. Due to space limits, we only show, for each value of \textit{alpha}, the ratios of \textit{Debop}'s reduction scores over \textit{Chisel}'s reduction scores, presented as averages and standard deviations computed...
We introduced a general formulation of program debloating as a multi-objective optimization problem and proposes an MCMC-based technique to solve the problem, as our work does. Our work is also related to techniques that use MCMC sampling to tackle other problems, such as superoptimization [20], and more broadly to techniques that improve software using genetic algorithms [16].

6 CONCLUSION AND FUTURE WORK

We introduced a general formulation of program debloating as a multi-objective optimization problem. We have also proposed a specific instance of this general approach, DeboP, that considers three objectives (size reduction, attack-surface reduction, and generality) and leverages MCMC sampling to approximate an optimally debloated program. Finally, we showed the feasibility and potential usefulness of our approach through a proof-of-concept evaluation.

In future work, in addition to performing a more extensive empirical evaluation, we will investigate ways to improve Algorithm 1 through the use of (1) mutation elements other than statements and (2) hierarchical mutations based on program structure. We also plan to investigate other sampling algorithms (e.g., Gibbs sampling [10]) and other stochastic search algorithms (e.g., genetic programming [15]) for optimization. In terms of longer-term and broader future research directions, the approach we proposed is general and can be applied in areas other than debloating. In particular, we plan to investigate the use of our approach in the context of resource adaptation [8], energy reduction [21], and program repair [9].

7 ACKNOWLEDGMENTS

We thank the authors of Chisel [12] for making their tool available. This work was partially supported by NSF, under grants CCF-1563991 and 1917924, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Google, IBM Research, and Microsoft Research.

REFERENCES