Program Debloating via Multi-Granularity Stochastic Optimization

JINRAN TANG, School of Computer Science, Wuhan University; Hubei Luojia Laboratory; State Key Lab. for Novel Software, Nanjing University, China
SHUFAN GONG, School of Computer Science, Wuhan University, China
HAO CHEN, School of Computer Science, Wuhan University, China
QI XIN[*], School of Computer Science, Wuhan University; Hubei Luojia Laboratory; State Key Lab. for Novel Software, Nanjing University, China
XIAOYUAN XIE, School of Computer Science, Wuhan University, China
JIFENG XUAN, School of Computer Science, Wuhan University, China

Code reduction and program generality are two key factors affecting the effectiveness of feature-based debloating. These two factors are inherently in tension: reducing more code typically decreases program generality, and vice versa. Therefore, effective debloating must strike a good tradeoff between them. To this end, a previous technique, Debop, addresses debloating as an optimization problem. It uses an objective function to quantify, for any debloated program that it generates, the reduction, generality, and their tradeoff, and employs a Markov-Chain-Monte-Carlo-based sampling approach for stochastic optimization, aiming to find the best sample, that is, the debloated program with the highest tradeoff score. Unfortunately, because Debop uses a very simple mutation model, which reduces or recovers only one primitive statement (often a line of code) for sampling, its search ability is significantly limited, especially when the program is large and in scenarios where code reduction is prioritized over generality. To improve the stochastic search for better debloating, we propose Mop, a new optimization-based technique that uses three mutation models targeting different code granularities for sample generation based on the code coverage (derived from the program execution with the feature-characterizing inputs), the coverage segments (exercised by unique subsets of features), and the statements. The use of these models accounts for both static code structure and dynamic program execution for sample generation and enables both aggressive and fine-grained exploration for improved optimization. However, as the optimization is guided by a set of inputs serving as the usage profile, it may mistakenly delete robustness-related code that is not covered by these inputs. Therefore, after the optimization, Mop also performs fuzzing-guided code augmentation to enhance the robustness of the optimized program.

To assess the effectiveness of Mop, we implemented a prototype of the approach and applied it to an existing benchmark of 25 programs and a real-world database management system PostgreSQL for debloating. Our results are encouraging, as they show that Mop can produce debloated programs with different tradeoffs and achieve a tradeoff score that is on average 15% (and in many cases 30%) higher than Debop's. We also found that Mop's code reduction ability is at least comparable to that of the state-of-the-art reduction-oriented

[*]Corresponding author.

Authors' Contact Information: Jinran Tang, School of Computer Science, Wuhan University; Hubei Luojia Laboratory; State Key Lab. for Novel Software, Nanjing University, China, jinrantang@whu.edu.cn; Shufan Gong, School of Computer Science, Wuhan University, China, sfgong@whu.edu.cn; Hao Chen, School of Computer Science, Wuhan University, China, howie.c@whu.edu.cn; Qi Xin, School of Computer Science, Wuhan University; Hubei Luojia Laboratory; State Key Lab. for Novel Software, Nanjing University, China, qxin@whu.edu.cn; Xiaoyuan Xie, School of Computer Science, Wuhan University, China, xxie@whu.edu.cn; Jifeng Xuan, School of Computer Science, Wuhan University, China, jxuan@whu.edu.cn.

ACM 1557-7392/2026/1-ART
https://doi.org/10.1145/3796513

techniques and that it can produce robustness-enhanced programs. Overall, the results demonstrate Mop's improved proficiency for optimization-based debloating.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Maintaining software**.

Additional Key Words and Phrases: Program Debloating, Stochastic Optimization, Multi-Granularity Optimization

ACM Reference Format:
Jinran Tang, Shufan Gong, Hao Chen, Qi Xin, Xiaoyuan Xie, and Jifeng Xuan. 2026. Program Debloating via Multi-Granularity Stochastic Optimization. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 62 pages. https://doi.org/10.1145/3796513

1 Introduction

Modern software is complex and is becoming increasingly so [37, 46]. To serve a wide range of users, complex software often contains an abundance of features. However, different users tend to use different features [36]. For a certain user, many features are rarely, if ever, used. This is evidenced by a study conducted by Quach et al. [74] that shows, for a variety of modern software systems, only a subset of the features (and less than 40% of the code) is exercised in their typical usage. The large amount of unneeded features contributes to code bloat. Code bloat is pervasive in modern software systems [25, 43, 74], and it is harmful in many ways [2, 15, 81, 97, 98]. Notably, it increases the attack surface, as any vulnerabilities existing in the bloated code can be exploited for malicious purposes [11, 81]. To reduce code bloat, debloating techniques have emerged [10, 11, 18, 35, 38, 64, 66, 67, 71, 72, 75, 96]. Among these, a prominent group referred to as feature-based debloating techniques [96] targets the removal of the unneeded features [4, 6, 7, 16–18, 35, 47, 52, 62, 71, 83, 86, 88, 94, 96].

Current feature-based debloating techniques often use a set of inputs for feature specification. They focus on code reduction and tend to eliminate as much unneeded code as possible to produce a minimal program behaving correctly for the inputs. However, debloating in this view is limited, as in more realistic scenarios, there would typically be different, possibly conflicting goals at play. For example, it may be acceptable for a debloated program to fail to handle 10% of the inputs corresponding to uncommon features if doing so results in 80% of reduction in its potential vulnerabilities. Likewise, there may be scenarios in which *generality* (i.e., the extent to which the debloated program can handle all inputs) is the primary concern and should therefore be prioritized over program-size reduction.

In light of this observation, Xin et al. formulated debloating as an optimization problem where they identified reduction and generality as two key factors affecting the effectiveness of feature-based debloating [94]. These two factors are inherently in tension: reducing more code typically removes more features, thereby decreasing program generality, and vice versa. Consequently, an effective debloating technique must pursue a good tradeoff between them. The proposed approach Debop [94] is designed around this idea. It performs stochastic search, aiming to find the debloated program with the best reduction-generality tradeoff.

For debloating, Debop takes as input a target (bloated) program and a usage profile of it, as a set of inputs representing various usage scenarios of the program and can be curated based on tests or collected usage data from users. Debop generates debloated programs by selectively removing statements of the target program guided by an objective function, which quantifies, for each debloated program, the code reduction (measured by the numbers of statements and ROP gadgets deleted), the program generality (measured by the fraction of inputs in the usage profile correctly handled), and the tradeoff between reduction and generality (measured by a weighted sum of the two factors). The goal then is to search within a pool of candidate debloated programs for the one with the highest objective value, which indicates the best tradeoff. Intuitively, by performing debloating in this way, Debop can identify features that are rarely used but exercise a large amount of code. By eliminating such features (and the corresponding code), Debop offers benefits in significantly reducing program size and attack surface while not considerably undermining the program's generality.

Because enumerating all candidate programs to find the optimal solution is infeasible due to the enormity of the pool, Debop uses a Markov-Chain-Monte-Carlo-based (or MCMC-based) approach for sampling. Following the MCMC principle, Debop performs a sequential, mutation-based process to obtain samples (debloated programs) as follows. Starting with the initial sample derived from the original program, Debop creates a new sample by mutating the current sample through either deleting a statement of it or recovering (adding back) one that has been deleted. Then Debop evaluates the new sample by computing its objective value and comparing the value with that of the current sample to decide whether to accept the new sample or not. Once accepted, the new sample becomes the current sample, and the exploration continues. Finally, after the sampling process terminates, Debop reports as output the sample with the highest objective value.

Despite the potential regarding the stated goal, Debop's effectiveness for optimization is very limited. The use of a very simple mutation model, which targets only one primitive statement (often just a line of code) for mutation, can result in a weak exploration for a program even with medium size. Consider a program that has $1,000$ statements. Because each statement can be either preserved or deleted, the size of search space is $2^{1,000}$, which makes an effective exploration extremely challenging. Moreover, the simple one-statement model can impede the deletion of any large code chunks — a statement-by-statement removal of a large chunk enabled by the model may not be feasible from an optimization perspective, as a statement removal can result in a considerable generality drop but a limited reduction gain and thus be rejected. As we will show in the evaluation section, Debop often struggles in pruning a large amount of code, especially for large programs. On average, its optimization process reduced 11% of the code. For large programs with more than 32K LoC, only 3% code is pruned, even when reduction is prioritized over generality.

Our work. To address the limitation and improve the effectiveness of stochastic search, we developed Mop (Program Debloating via Multi-Granularity Stochastic Optimization), a new stochastic-optimization-based debloating approach. Like Debop, Mop uses an MCMC-based sampling strategy for debloating with the goal of finding a debloated program with the best reduction-generality tradeoff. The key difference between Debop and Mop is that, instead of using a simple statement-based model to generate samples, Mop adopts three models targeting different code granularities: code coverage, coverage segments, and statements. Inspired by Hierarchical Delta Debugging (HDD) [60], Mop uses a multi-granularity strategy to enable coarse-to-fine-grained search for effective space exploration. Starting with coarse-grained exploration, Mop quickly narrows down a promising subspace, which it can focus on next for fine-grained search. In this way, Mop avoids costly fine-grained searches in suboptimal spaces so as to improve the optimization.

For debloating, Mop takes as input a (bloated) program p and a usage profile as a set of inputs I representing the various usage scenarios of p. It additionally allows the user to provide an optional I', a subset of I that specifies the features the debloated program must preserve, and two weights α and β controlling the preferences between two types of reduction (i.e., program size reduction and attack surface reduction) and between reduction and generality, respectively. Mop's debloating goes through a pre-processing step, which identifies a favorable initial program for optimization, a three-stage optimization process, and a post-processing step, which enhances the robustness of the optimized program through fuzzing-guided code augmentation. As with Debop, Mop uses an objective function that quantifies, for each reduced program that it generates, the reduction, generality, and their tradeoff, and as the output of optimization, it reports the sample with the highest objective value, indicating the best tradeoff.

Pre-processing step. The debloating process begins with a pre-processing step where Mop creates an initial program as a favorable starting point for optimization. The initial program can be either p_{top} or p_{base}, which Mop obtains by preserving all the code exercised by I or I' and removing all other code not exercised by the inputs. In the subsequent optimization process, Mop can then focus on pruning code from p_{top} or adding back code from p to p_{base} to generate samples. The way Mop chooses the initial program is by comparing the objective values of p_{top} and p_{base} and selecting the one with the higher value. Intuitively, this is the program closer to the optimal solution. After creating the initial program, Mop performs optimization in three stages for debloating, each with a different mutation model. The output of the previous stage is the input of the following.

Stage-I optimization. In the first stage, Mop opts for an aggressive MCMC-based exploration. It uses a mutation model that eliminates or retains the coverage code induced from the program execution with a randomly chosen input $i \in I$ to generate a sample. Using the model, Mop essentially enables an exploration that decides which inputs in I should be handled by the debloated program. The exploration is aggressive, as for an input chosen not to handle, the model has a chance of eliminating the input-exercising code all at once. The model also facilitates fast exploration. This is because, with the model, Mop does not have to repeatedly execute every reduced program against the inputs to evaluate generality. Rather, it does program profiling once to keep track of the coverage induced by each input, and later for optimization, it can compute generality just by checking how many inputs have been chosen to handle.

Stage-II optimization. The coverage model adopted in the first stage enables a fast and aggressive exploration. There are however missed optimization opportunities. Because the model targets the coverage code for mutation, it does not assess how smaller code units (for example a part of the coverage) affect the optimization result. Moreover, because the model focuses on the generality change led by one input, it ignores other mutations that affect generality to a larger degree based on multiple inputs. To address these weaknesses, in the second stage, Mop uses another model that targets coverage segments for mutation. To identify the coverage segments, Mop creates a partition of all the (primitive) statements from the original program, each comprising statements exercised by a unique subset of I. The statements in a segment are semantics-related, as they are exercised by the same inputs. For mutation, Mop selects a segment and preserves or deletes its statements altogether, depending on whether the segment exists in the current sample or not.

Stage-III optimization. A coverage segment that contains multiple statements is still not small enough. For a more fine-grained exploration, Mop uses a model that mutates only one (primitive) statement. With this model, Mop has a chance of eliminating single statements that do not or can only slightly affect the generality, which is beneficial for fine-grained optimization.

Post-processing step. By the end of the optimization process, Mop can produce a program with poor robustness, as it can remove robustness-related code (e.g., error-handling code) that is deemed not important from a feature-based optimization perspective but is actually critical. To further enhance program robustness, Mop employs CovF's strategy [96] for fuzzing-guided code augmentation. Specifically, Mop leverages the black-box fuzzing tool Radamsa [5] to generate a set of fuzzed inputs triggering the crash and hang behaviors of the debloated program. It then augments the program by adding back code from the original program exercised by the fuzzed inputs to avoid such behaviors and similar failures. Finally, Mop reports the augmented program as the debloating result.

Evaluation. We implemented a prototype of Mop and applied it to a benchmark of 25 subject programs [96] used to evaluate existing techniques [35, 71, 94] together with a real-world database management system PostgreSQL (v12.14) [68] with over 936K LoC. Our results show that Mop can produce debloated programs with various tradeoffs. On average, it can prune nearly 70% of the original program's code while retaining 58% of the generality to achieve a good tradeoff. Moreover, with the must-handle inputs specifying features that must be preserved in the debloated program, Mop can avoid programs with low generality. A comparison of Mop with Debop, the state-of-the-art optimization-based debloating technique, shows that Mop achieves a tradeoff score that is about 15% higher than Debop. By excluding the influence of the pre-processing step to focus on core optimization ability, we found that Mop obtains a 30% higher tradeoff score. The improvement is even more pronounced when reduction is prioritized over generality. While Mop is an optimization-based technique, to have a deep understanding of its reduction ability, we also compared Mop with state-of-the-art reduction-oriented techniques Chisel and Razor, which we used to produce debloated programs based on the inputs that the debloated programs generated by Mop can correctly handle. We found that Mop achieved better reduction than Chisel and its reduction ability is at least comparable to Razor's. Finally, the evaluation of Mop's post-processing step demonstrates its effectiveness in enhancing program robustness. Overall, we believe that Mop represents a step forward towards better optimization-based debloating.

The main contributions of this paper are as follows.

(1) A new stochastic-optimization–based debloating technique Mop that improves the reduction–generality tradeoffs through a three-stage, multi-granularity optimization strategy, leveraging coverage-based, coverage-segment–based, and statement-based mutation models.
(2) A comprehensive evaluation on 26 programs demonstrating Mop's improved optimization ability over Debop, its effectiveness in code reduction, and its proficiency in robustness enhancement.
(3) An artifact that includes the prototype of Mop we implemented along with the experiment data and testing scripts we used available at [12].

2 Definitions

In this section, we provide the definitions of the terms used in this paper.

2.1 Compound and Primitive Statements

A program p can be parsed into an abstract syntax tree (AST) and it contains a set of statements S. A *statement* is code defined in a function body. It is either a *block* (e.g., the body of a while loop), which can be represented as a list of elements $\{e_1, e_2, \ldots, e_n\}$, or an element in the list structure of a block. A statement s can be mapped to a node in the AST, and it can have other statement(s) as its parent or children. A statement can be either *compound* or *primitive*. A compound statement has other statements as its children. Formally, we have

$$compound(s) \text{ iff } s \in S \land \exists s' \in S(s' \in children(s)), \tag{1}$$

where S is the set of all the statements in p and $children(s)$ denotes the children of s as a set. For example, a while-statement is a compound statement, as it contains other statements (e.g., an if-statement) in the loop body. In contrast, a primitive statement does not have any children. The definition of a primitive statement is shown below.

$$primitive(s) \text{ iff } s \in S \land \nexists s' \in S(s' \in children(s)) \tag{2}$$

Here again, S is the set of all the statements and $children(s)$ denotes the children of s.

Because our approach focuses on reducing primitive statements for optimization, we use *statement* to refer to primitive statement in this paper, unless we give a special note. It should be emphasized that our approach does not ignore compound statements for debloating. After pruning primitive statements, the approach uses an existing dead code elimination (DCE) method [96] to remove the dangling compound statements (do, for, if, switch, and while statements) that have empty bodies and side-effect-free conditions and other unneeded definitions.

2.2 Coverage

Let p be the program and i be an input. We define the *full coverage* induced from the execution of p with i, denoted as $FullCov(p, i)$, to be a set containing all the compound and primitive statements exercised in the execution. The *primitive coverage* $PrimCov(p, i)$, or simply $Cov(p, i)$, is the set of all the primitive statements that are exercised in the execution. We use *coverage* to refer to the primitive coverage in this paper.

2.3 Coverage Input Set

Let p be the program, s be a statement of p, and I be a set of inputs. We define the *coverage input set* for s based on p and I, denoted as $CovInputSet(p, I, s)$, to be a subset $I' \subseteq I$, where, for each input i' in I', the coverage derived from the execution of p with i' includes s. A formal definition of $CovInputSet(p, I, s)$ is shown below.

$$CovInputSet(p, I, s) = \{i \mid i \in I \land s \in Cov(p, i)\} \tag{3}$$

Intuitively, $CovInputSet(p, I, s)$ is a subset of I that exercises s in the execution. Note that $CovInputSet(p, I, s)$ can be empty, as there can be no inputs of I that exercise s.

2.4 Coverage Segment

A *coverage segment*, or simply *segment*, is a set of statements that share the same coverage input set. Let p be the program, S be the set of all the statements of p, I be a set of inputs, and C be some non-empty coverage input set chosen from all possible candidates. A formal definition of a segment, denoted as $CovSeg(p, S, I, C)$, is shown below.

$$\begin{gathered} CovSeg(p, S, I, C) = \{s \mid s \in S \land CovInputSet(p, I, s) = C\}, \\ \text{where } C \in \{CovInputSet(p, I, s) \mid s \in S\} \land C \neq \emptyset \end{gathered} \tag{4}$$

Note that a segment in this definition cannot be empty, as it must include some statement(s) exercised by the input(s) from an non-empty C.

2.5 Coverage Segment Set

Given the program p, the set of statements S in p, and the set of inputs I, we define the *coverage segment set*, or simply *segment set*, denoted as $CovSegSet(p, S, I)$, to be the set of coverage segments, each collected based on a candidate coverage input set. Formally, we have

$$CovSegSet(p, S, I) = \{CovSeg(p, S, I, C) \mid C \in \{CovInputSet(p, I, s) \mid s \in S\} \land C \neq \emptyset\}. \tag{5}$$

In the second-stage, Mop's model selects a segment from the segment set for mutation and generates a new sample.

2.6 Coverage, Top, and Base Programs

Let p be a program and I be a set of inputs. We define the *coverage program* with respect to p and I, denoted as $pcov(p, I)$, to be a program containing every primitive and compound statement exercised by I along with all the code (including for example a global struct definition) that is not a statement (either primitive or compound).

When the set of inputs I_u represents the usage profile used for optimization-based debloating, we refer to the corresponding coverage program as the *top program*, denoted as p_{top}, which is also equivalent to $pcov(p, I_u)$.

Likewise, when the set of inputs I_b represents the must-handle inputs that must be handled correctly by the debloated program, we refer to the corresponding coverage program as the *base program*, denoted as p_{base}, which is also $pcov(p, I_b)$.

2.7 Code Reduction

The *code reduction*, or simply *reduction*, quantifies the amount of code that has been removed. Following the previous work [94], we measure the code reduction by *size reduction* and *attack-surface reduction.* Given a target program p (which can be either the original program p_{ori} or the top program p_{top}) and the reduced program p', the definition of size reduction, denoted as $sred \in [0, 1]$, is shown below.

$$sred(p, p') = \frac{size(p) - size(p')}{size(p)}, p \in \{p_{ori}, p_{top}\} \tag{6}$$

In this formula, $size(\cdot)$ quantifies the size of a program. In our experiment, we used the number of all primitive and compound statements contained in a program's source code to measure sizes.

We define the attack-surface reduction $ared \in [0, 1]$ in a similar manner.

$$ared(p, p') = \frac{attksurf(p) - attksurf(p')}{attksurf(p)}, p \in \{p_{ori}, p_{top}\} \tag{7}$$

In this formula, $attksurf(\cdot)$ quantifies a program's attack surface. Following existing work [35, 71, 94], we used the number of Return Oriented Programming (ROP) gadgets [81] contained in a program's executable (binary code) to measure the attack surface of a program. An ROP gadget is a sequence of machine instructions that end with a return instruction. These instructions can be exploited by an attacker, who takes advantage of a vulnerability in the program (e.g., a buffer overflow) to hijack the control-flow and execute malicious code [19]. In the experiment, we used the ROPgadget tool [78] to count the number of ROP gadgets.

Finally, we define the code reduction, denoted as $red(p, p') \in [0, 1]$, to be a weighted sum of size reduction $sred(p, p')$ and attack-surface reduction $ared(p, p')$.

$$red(p, p') = (1 - \alpha) \cdot sred(p, p') + \alpha \cdot ared(p, p'), p \in \{p_{ori}, p_{top}\} \tag{8}$$

In the formula, $\alpha \in [0, 1]$ is the weight factor.

Note that we consider not only the original program but also the top program to compute reduction. Using the original program, we can quantify for a debloated program how much code is reduced. The use of the top program allows us to quantify how much code is indeed reduced by the optimization process. Since Mop's stochastic search does not involve changing code not contained in the top program (as we will detail later in the paper), code reduction based on the top program is more interesting, as it reveals Mop's optimization ability.

2.8 Program Generality

The *program generality*, or simply *generality*, quantifies the extent to which a reduced program behaves correctly (i.e., as the original program does) for the inputs provided in the usage profile. Formally, let p be the original program, p' be a reduced program, and I be a set of inputs, the generality of p' with respect to p and I, denoted as $gen(p, p', I)$, is defined below.

$$gen(p, p', I) = \Sigma_{i_k \in I} pr_k \cdot T(i_k) \tag{9}$$

The generality is calculated as the weighted number of inputs for which p' behaves correctly. $T(i_k)$ is an indicator function showing whether p behaves correctly for input i_k. A formal definition of $T(i)$ is shown below.

$$T(i) = \begin{cases} 1, & p(i) = p'(i) \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Here we use $p(i)$ to denote the output of running p with i. In formula (9), pr_k denotes the probability associated with input i_k showing the importance of the input. Assuming all inputs have equal importance, then the generality is reduced as the fraction of inputs for which the reduced program behaves correctly. Formally, we have

$$gen_{eq}(p, p', I) = \frac{\Sigma_{i_k \in I} T(i_k)}{|I|}, \tag{11}$$

where gen_{eq} is the generality computed based on the assumption that all inputs in I are of equal importance and $|I|$ is the number of inputs in I.

2.9 Reduction-Generality Tradeoff

Following the previous work [94], we quantify the tradeoff between reduction and generality as the weighted sum of the two factors. Let p be the target program (either the original program p_{ori} or the top program p_{top}), p' be a reduced program, and I be the set of inputs. We used the formula below to define tradeoff, denoted as $O(p, p', I)$.

$$O(p, p', I) = (1 - \beta) \cdot red(p, p') + \beta \cdot gen(p, p', I), p \in \{p_{ori}, p_{top}\} \tag{12}$$

In this formula, $\beta \in [0, 1]$ is a weight used to control the relative importance between reduction and generality.

2.10 Optimization-Based Debloating

We follow the previous work [94] to define the goal of optimization-based debloating. Let p be the original program, p' be a reduced program from $Sub(p)$, a set of all possible reduced programs, I be the set of inputs, and $O(p, p', I)$ be an objective function that computes the reduction-generality tradeoff based on p, p', and I. The goal of optimization-based debloating is to generate the reduced program $p_{deb} \in Sub(p)$ with the best tradeoff value, that is, the maximum objective function value. A formal definition is shown below.

$$p_{deb} = \underset{p' \in Sub(p)}{\arg\max}\ O(p, p', I) \tag{13}$$

3 Approach Overview with an Example

In this section, we use an example to illustrate how Mop works and why Mop can outperform Debop.

Mop takes as input the original program p, a set of inputs I showing how p is used in various cases, an optional subset of the inputs representing features that any debloated program must preserve, and the preference controlling weights α and β (both default to 0.5). For debloating, Mop starts by identifying an initial program and then performs MCMC-based optimization in three sequential stages, aiming to generate a debloated program p' achieving the best tradeoff between reduction and generality. To further improve the robustness of p', Mop first generates fuzzed inputs I' to validate p' and identifies inputs $I'' \in I'$ that cause p' to crash or hang. Leveraging these inputs, Mop augments p' by collecting the code covered by the original program p when executing I'' and

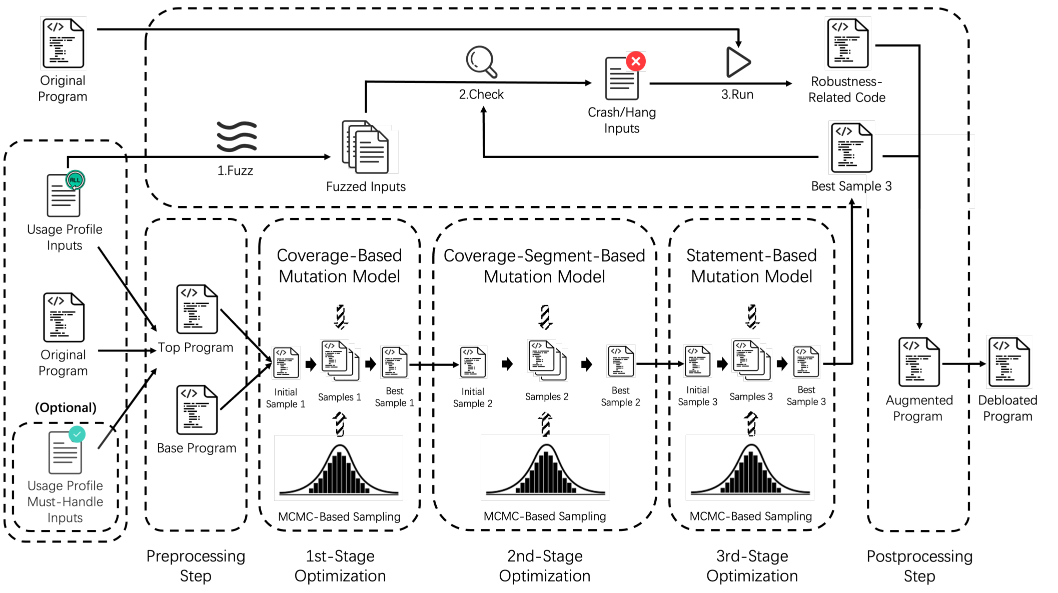


Fig. 1. An overview of Mop.

adding the covered code back to p' to generate p''. Finally, Mop reports the augmented program p'' as output. Figure 1 shows an overview of Mop.

We choose as an example a utility program Tar (v1.14) from an existing benchmark [96] to show how Mop works in a specific usage scenario and explain why Mop outperforms Debop. Tar is a widely used utility for archiving and extracting files and updating and listing files of the archives.

Consider that James is a farmer who owns multiple farms. He lives in the city, but his farms are located in remote mountainous areas. If James wants to know the current status of his farms, he can communicate with the farm workers online. Unfortunately, due to signal and network issues, communication can be unreliable in the mountainous area. To make things easier, James plans to equip each farm with embedded devices serving as monitors for collecting various information including the temperature, humidity, climate, and plant growth. These pieces of information will be sent to James' terminal server at fixed time intervals. For an efficient use of the hardware environment provided by the embedded devices, James wants the systems installed in the devices to be lightweight enough.

James uses a lightweight POSIX Linux-embedded system. Due to the unreliable network, the longer the data transmission time, the more likely transmission failures may occur. So James chooses to compress the data to reduce its size. In the Linux system, James uses Tar to archive various monitoring information and compresses it through *gzip* or *bzip2*. James thinks that Tar should at least contain two features regarding creating the tar files (using the *-cvf* option) and extracting tar files (using *-xvf*). However, a program with only two features is not ideal, as its generality is very limited. James hopes to improve the generality as much as possible without however excessively increasing the program size. Unfortunately, James does not understand which features are implemented by which source code, nor does he know how to remove more code while retaining the program's generality as much as possible. Therefore, he uses Mop to solve the problem.

James has a set of inputs (95 in total) curated based on the test cases associated with the program and the user manual of Tar. These inputs exercise a variety of commonly used features such as decompressing files, archiving files, compressing files, listing files of a tar file, extracting files with wildcards, extracting files created after a certain time, and excluding files from a certain directory. Among these, James identifies 9 must-handle inputs exercising two features about creating the archives (each of these inputs uses the *-cvf* option) and extracting files from the archives (using the *-xvf* option). James considers reduction and generality as equally important in his case, and he sets β, which controls the preference between reduction and generality, as 0.5. He also sets α as 0.5 to give the same weight for size reduction and attack surface reduction. Taking as input the original program, the 95 usage profile inputs, the 9 must-handle inputs, and the controlling weights α and β both as 0.5, Mop generates a reduced version of Tar, achieving a 81.2% size reduction while still behaving correctly (i.e., as the original program does) for 91.4% of the inputs. James is happy with this result, as Mop gives him a debloated program by removing a significant amount of code while preserving most of the original program's generality.

Next, we will show how Mop generates the debloated program for *tar-1.14*.

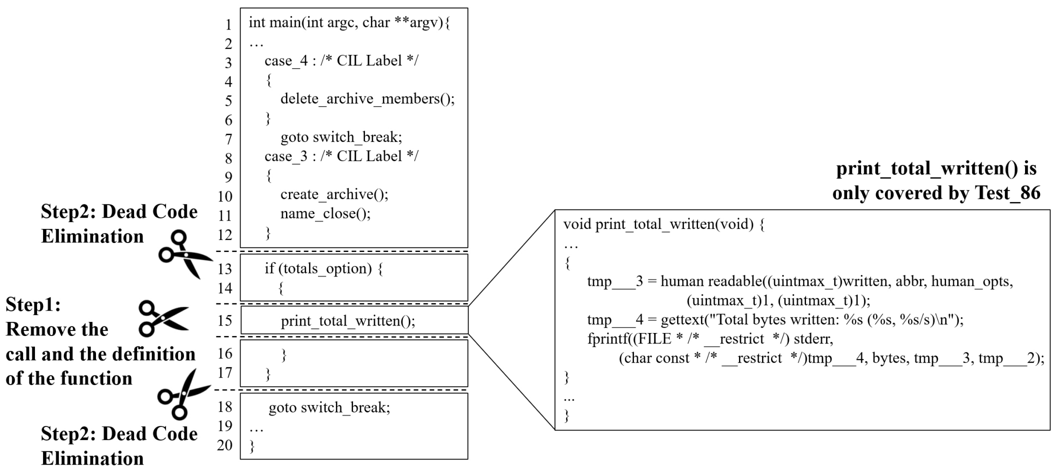
3.1 How does Mop Debloat Tar?

Mop's debloating process goes through a preprocessing step, a three-stage optimization, and a postprocessing step.

Preprocessing step. The preprocessing step identifies an initial program for the subsequent optimization. Specifically, Mop first generates the top program p_{top} and the base program p_{base} of Tar, and compares their objective values to determine which should be the initial program (the initial sample). The way Mop generates these programs is by removing statements that have not been exercised in the original program's execution against the profile inputs and the must-handle inputs, respectively. In our example, because the objective value of p_{top} is higher than that of the p_{base} (0.54 vs. 0.37), Mop chooses p_{top} as the initial program, whose size has been reduced by 73.8%. In the following optimization stages, Mop quantifies the size or attack surface reduction for a reduced program by comparing it against p_{top}.

Note that for optimization it is fine and actually advantageous to use p_{top} rather than the original program as the initial program. The removal of code not exercised by the profile inputs does not affect generality. In our example, by removing code not preserved in p_{top}, Mop eliminates a feature of Tar designed to handle GNU format files. This involves the removal of the whole function body of *to_base256()*, which converts the negative value to a base-256 representation suitable for tar headers and is only invoked when GNU format files are provided. The feature is not needed, as James only uses Tar in POSIX systems.

Stage-I Optimization. Next Mop operates in three stages for optimization. In the first stage, Mop performs the MCMC-based stochastic search using a coverage-based mutation model. Mop first identifies as non-removable the statements exercised in the execution of the original program against the 9 must-handle inputs and ensures that these statements are preserved throughout the debloating process. Then Mop obtains a set of coverages. Each coverage is identified based on one of the remaining 86 inputs and contains all statements exercised by the input. To generate a new sample, Mop randomly selects a coverage and either removes the coverage from the current sample or adds it back to the sample, depending on whether the selected coverage has been preserved in the current sample or not. When Mop chooses to remove a coverage led by input i, to ensure that other inputs I_{other} whose coverages have already been preserved can be handled correctly, Mop removes the statements exclusively covered by i and not by any of I_{other}. For example, as Figure 2 shows, the statement $print\_total\_written()$; (line 15) is only covered by *Test_86*, whereas the



```
 1  int main(int argc, char **argv){
 2  …
 3    case_4 : /* CIL Label */
 4    {
 5      delete_archive_members();
 6    }
 7      goto switch_break;
 8    case_3 : /* CIL Label */
 9    {
10      create_archive();
11      name_close();
12    }
13    if (totals_option) {
14      {
15        print_total_written();
16      }
17    }
18    goto switch_break;
19  …
20  }
```

```
void print_total_written(void) {
…
{
    tmp___3 = human readable((uintmax_t)written, abbr, human_opts,
                    (uintmax_t)1, (uintmax_t)1);
    tmp___4 = gettext("Total bytes written: %s (%s, %s/s)\n");
    fprintf((FILE * /* __restrict */) stderr,
          (char const * /* __restrict */)tmp___4, bytes, tmp___3, tmp___2);
}
...
}
```

Step2: Dead Code Elimination
Step1: Remove the call and the definition of the function
Step2: Dead Code Elimination
print_total_written() is only covered by Test_86

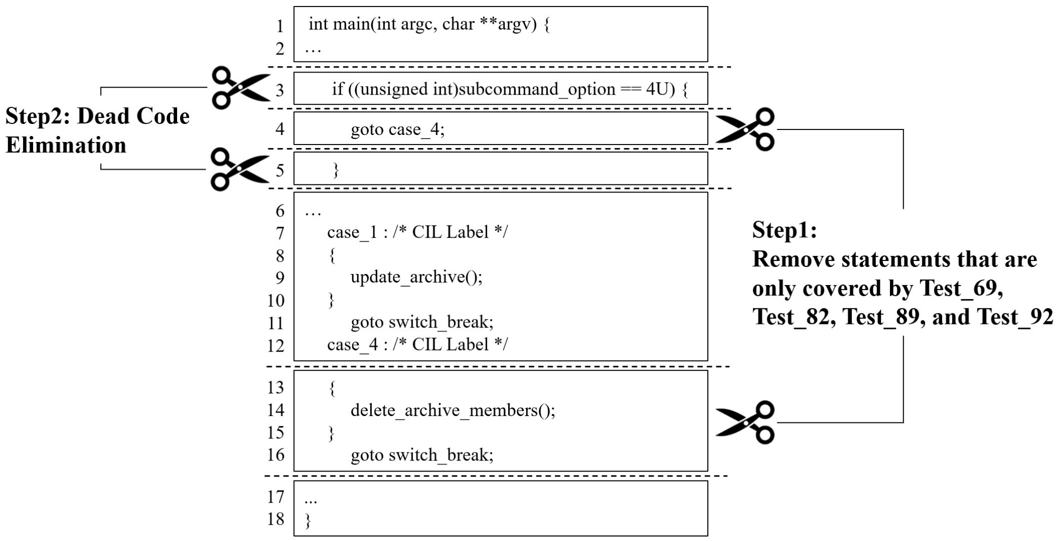
Fig. 2. Mop removes statements exercised by the input *Test_86* in the first stage.

two statements *create_archive*(); (line 10) and *name_close*(); (line 11) are covered by 35 inputs including *Test_86*. If Mop chooses to remove the coverage led by *Test_86*, it removes the statement *print_total_written*(); but not the other two.

In this example, Mop removes the coverage code exercised by *Test_86*, which uses the option *–totals* to print the size of the generated file. The program execution induced by *Test_86* invokes *print_total_written*() (line 15) in the *main*(·) function to print the size of the tar file. By choosing not to handle the input, Mop removes the statement *print_total_written*();, all statements in the function body of *print_total_written*(), and many other statements (not displayed in Figure 2) that are exercised only by *Test_86*. Mop focuses on pruning primitive statements, and at the end of the stage, it removes the *if* statement (line 13) with an empty body using an existing dead code eliminator [96]. Mop computes and compares the tradeoff scores before and after the code removal, and considers the removal to be beneficial, since the resulting program fails to handle only one more input, but the code reduction is significant — 107 statements and 78 ROP gadgets are removed. James is fine with a debloated version of Tar not handling *Test_86* for printing the size of the generated tar files, as he can simply use *ls* as an alternative.

Because Mop performs coverage-based optimization, it can easily track which of the inputs the current sample can correctly handle and does not have to dynamically run the sample with all the inputs to compute generality. For this reason, Mop is fast. It takes only about 4 seconds to generate a sample, and Mop generated 304 samples in just 20 minutes. In this stage, Mop produced a debloated program whose size and attack surface are 32.7% and 22.5% smaller than the initial sample. The generality score of the program is as high as 0.95.

Stage-II Optimization. In the second stage, Mop performs MCMC-based optimization with a coverage-segment-based mutation model. It first identifies the coverage segment set of the initial sample, which is the sample with the best tradeoff (the highest objective value) identified in the previous stage. The way Mop obtains the segment set is by first identifying, for each statement of the initial sample, its coverage input set, which is the set of inputs that exercise the statement, and then grouping the statements based on the coverage input sets. Each group comprises statements that share the same coverage input set and is identified as a segment. For example, as shown in Figure 3, the statements *goto case*_4; (line 4), *delete_archive_members*(); (line 14), and *goto switch_break*;



```
1   int main(int argc, char **argv) {
2   …
3       if ((unsigned int)subcommand_option == 4U) {
4           goto case_4;
5       }
6   …
7       case_1 : /* CIL Label */
8       {
9           update_archive();
10      }
11          goto switch_break;
12      case_4 : /* CIL Label */
13      {
14          delete_archive_members();
15      }
16          goto switch_break;
17  ...
18  }
```

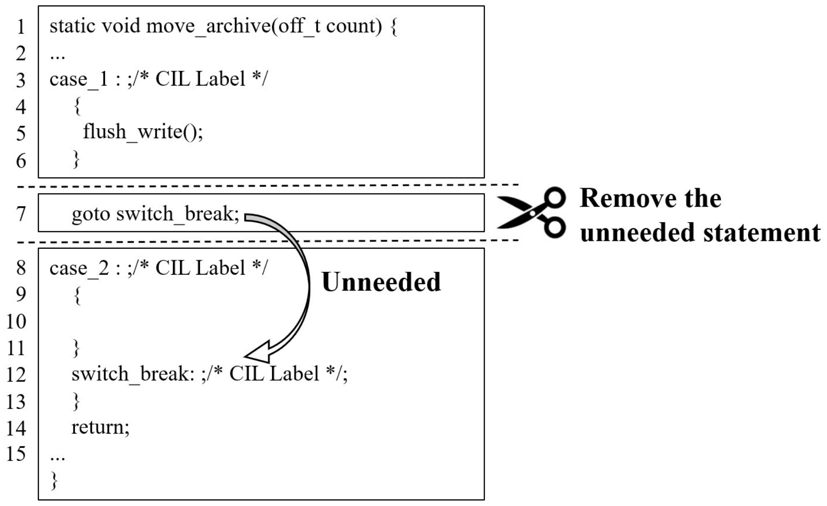
Step2: Dead Code Elimination
Step1:
Remove statements that are only covered by Test_69, Test_82, Test_89, and Test_92

Fig. 3. Mop removes the coverage segment whose coverage input set is {*Test_69*, *Test_82*, *Test_89*, *Test_92*} in the second stage.

(line 16) are all covered by *Test_69, Test_82, Test_89*, and *Test_92* and not by other inputs. The three statements together with many others not displayed in the figure but are exercised by the four inputs are grouped into a segment. Unlike how Mop mutates a coverage in the first stage, when Mop chooses to mutate a coverage segment, it can simply remove or recover all statements contained in this segment without performing any additional check, as there is no statement overlap between different segments.

While the first stage focuses on mutating a coverage derived from the execution with one input, the second stage allows Mop to change statements affecting multiple inputs. In this example, by removing the segment containing the three statements (lines 4, 14, and 16) and many others, Mop prunes the code needed for handling the four inputs *Test_69*, *Test_82*, *Test_89*, and *Test_92*, which represent a feature of deleting files and folders in a tar archive. The first stage can however struggle to eliminate this feature by removing the coverages led by these inputs individually: Each time Mop selects one of the four coverages to delete, the resulting sample can be easily rejected. This is because the four inputs represent a similar feature and the overlap of their coverages is significant, and as a result, the reduction gain introduced by the removal of the small set of non-overlapping statements in one coverage is minimal and often outweighed by the generality loss. For example, by attemping to remove the coverage led by *Test_69* from the initial sample (the top program), Mop can actually only delete 8 non-overlapping statements and produce a new sample whose size reduction gain is only 0.75%. Because the tradeoff score drops, the sample can possibly be rejected. In comparison, by removing the segment, Mop produces a new sample with a 2.1% generality loss while achieving a 5.5% increase of size reduction and 30.2% increase of attack surface reduction. Overall, because the tradeoff score of the new sample is higher, Mop chooses to accept the sample for further exploration.

At the end of the stage, Mop produced a debloated program whose size and attack surface are 37.1% and 30.5% smaller than the initial sample. The generality score of the program is as high as 0.91.



```
1   static void move_archive(off_t count) {
2   ...
3   case_1 : ;/* CIL Label */
4     {
5       flush_write();
6     }
7     goto switch_break;
8   case_2 : ;/* CIL Label */
9     {
10
11    }
12    switch_break: ;/* CIL Label */;
13    }
14    return;
15  ...
    }
```

Fig. 4. Mop removes an unneeded statement in the third stage.

Stage-III Optimization. The previous two stages remove or add back a set of statements (either a coverage or a segment) for mutation in pursuit of aggressive optimization. The last stage aims for fine-grained exploration, using a model that mutates a single statement for sample generation. The rationale is that by using the model for optimization Mop can progressively remove some "extra" statements that do not affect generality, especially those that become unneeded after the optimization of the previous stages. For example, the statement *goto switch_break*; (line 7) in Figure 4 is unneeded once the code under *case*_2 is fully pruned. In the previous two stages, Mop did not delete the statement, as the statement is exercised by 45 inputs, and the removal of it can significantly affect generality. The last stage of optimization increases the reduction score by 3% with no generality loss.

Postprocessing Step. After the three-stage exploration, Mop obtains a debloated version of Tar as the optimization result. For robustness enhancement, Mop first employs the Radamsa fuzzer to generate 950 fuzzed inputs (10 fuzzed inputs per original input) and then tests the debloated program against them. The debloated program crashed or hung on 397 inputs. Using these crash/hang-triggering inputs, Mop performs augmentation by identifying the original program's code exercised by the inputs and adding it back to the debloated program. The code augmentation leads to a very slight drop of the size-reduction ratio, from 81.6% to 81.2%. However, the augmented program can now correctly handle all the 397 inputs and avoid similar crashes and hangs.

3.2 The weakness of Debop's approach

For comparison with Mop, we slightly modified Debop—the state-of-the-art optimization-based debloating technique—to create Debop-M, which supports debloating with the must-handle inputs, a capability not provided by the original Debop. We applied Debop-M to Tar for debloating based on the same inputs, the same must-handle inputs, and the same α and β values. Surprisingly, we found that Debop-M deleted zero statement and finally reported the initial sample p_{top} as the debloating output.

This result reveals Debop-M's weakness in achieving an effective tradeoff between reduction and generality for debloating. By using a mutation model that removes only one statement at a

```
Algorithm 1: Mop's debloating algorithm.
Input  : $p$: the original program
         $\alpha$: the weight controlling the preferences between size reduction and attack surface reduction
         $\beta$: the weight controlling the preferences between reduction and generality
         $k$: the weight adjusting the sample acceptance difficulty
         $I$: the set of inputs (usage profile)
         $B$: the set of must-handle inputs (optional)
         $t_1$: the maximum time (in minutes) for 1st-stage optimization
         $t_2$: the maximum time (in minutes) for 2nd-stage optimization
         $t_3$: the maximum time (in minutes) for 3rd-stage optimization
Output : $p_b$: the debloated program
1  Function $main(p, \alpha, \beta, k, I, B, t_1, t_2, t_3)$
2      $p_{top} \leftarrow getTopProgram(p, I)$
3      $p_{base} \leftarrow getBaseProgram(p, B)$
4      $PI \leftarrow getCoverage(p_{top}, I)$
5      $p_b \leftarrow getInitProgram(p_{top}, p_{base}, \alpha, \beta, k, I)$
6      $p_{b1} \leftarrow MHSearchAtCov(p_{top}, PI, p_b, \alpha, \beta, k, I, B, t_1)$
7      $p_{b2} \leftarrow MHSearchAtCovSeg(p_{top}, PI, p_{b1}, \alpha, \beta, k, I, B, t_2)$
8      $p_{b3} \leftarrow MHSearchAtStmt(p_{top}, PI, p_{b2}, \alpha, \beta, k, I, B, t_3)$
9      $p_{br} \leftarrow enhanceRobFuzz(p, p_{b3}, I)$
10     return $p_{br}$
```

time to generate a new sample, Debop-M can only achieve a slight reduction gain. However, there can be a significant generality loss when some key statement is removed. When that happens, Debop-M will reject the sample, as the objective value drops (because of the substantial decrease of generality). The problem is that the slight reduction gain is often not comparable to a generality loss, incurring frequent sample rejection and ineffective exploration.

4 Our Approach

In this section, we elaborate on Mop's debloating approach.

Algorithm 1 details the approach. Mop takes as input (1) a (bloated) program p, (2) two weight parameters α and β controlling the preferences between size reduction and attack surface reduction and between reduction and generality respectively, (3) a constant k introduced for adjusting the sample acceptance difficulty, (4) a set of inputs I as the usage profile for generality quantification, (5) an optional set of must-handle inputs B (as a subset of I) for which the debloated program must behave correctly (i.e., as the original program does), and (6) the time budgets, t_1, t_2, and t_3, which specify the maximum exploration time (in minutes) for the three optimization stages respectively.

For debloating, Mop starts by creating the top (line 2) and base (line 3) programs, identifying the set of coverages (line 4) each comprising the statements exercised by an input in I, and comparing the top and base programs to determine the initial sample p_b for optimization. Mop next performs optimization in three stages by calling $MHSearchAtCov(\cdot)$, $MHSearchAtCovSeg(\cdot)$, and $MHSearchAtStmt(\cdot)$ sequentially (lines 6-8). The debloated program (sample) reported as the output of each stage is the sample with the highest objective value (the best tradeoff) found in that stage. The initial sample p_b is the input sample for the first stage. The output of the first stage is

Algorithm 2: The Metropolis-Hastings algorithm.

```
Input  : $s_i$: the initial sample
         $f$: probability density function
         $q$: transition matrix
         $t$: the maximum time for exploration
Output : $S$: a set of samples
1  $s_{curr} \leftarrow s_i$
2  $time_s \leftarrow getCurrTime()$
3  repeat
4      $s_{new} \leftarrow$ mutate $s_{curr}$ by adding random noise
5      $ratio \leftarrow f(s_{new}) \cdot q(s_{curr}, s_{new}) / f(s_{curr}) \cdot q(s_{new}, s_{curr})$
6      $rn \leftarrow$ get a uniform random number                // $rn \in [0, 1)$
7      if $rn < ratio$ then
8          $S \leftarrow S \cup s_{new}$
9          $s_{curr} \leftarrow s_{new}$
10     $time_c \leftarrow getCurrTime()$
11 until $time_c - time_s \geq t$
12 return $S$
```

the input sample for the second stage, and the third-stage optimization starts with the output of the second stage. Finally, to enhance the robustness of the optimized program (the output of the third stage), Mop employs a current blackbox fuzzer Radamsa [5] to generate a set of fuzzed inputs based on I and tests the program with them. For inputs exposing crash and hang behaviors, Mop identifies missing code exercised by these inputs from the original program and adds it back to the output program for augmentation (line 9). The augmented program is reported as the debloating result (line 10).

Next, we show as the background knowledge how the MCMC approach works and then detail in sequence the initial sample generation, Mop's three-stage optimization based on MCMC, and Mop's fuzzing-guided method applied to the optimized program for robustness improvement.

4.1 The Markov-Chain-Monte-Carlo (MCMC) and Metropolis-Hastings (MH) Algorithms

Since it is generally infeasible to enumerate every reduced program in the search space, given its exponential size, Mop performs stochastic search, using an MCMC-based approach, to find a close-to-optimal solution.

The MCMC algorithm is a sampling-based approach commonly used to estimate properties, such as mean and variance, of a given probability distribution (whose probability density function is known). This approach performs a sequential process to draw samples from the distribution, where the generation of a new sample only depends on the previous sample.

The Metropolis-Hastings (MH) algorithm is a commonly used MCMC-based approach. This algorithm works by generating new samples through mutation (adding random noise to the current sample) and probabilistically accepting these samples. Algorithm 2 shows the sampling process of the MH algorithm. It takes as input (1) the initial sample s_i, (2) a probability distribution defined by a probability density function f, (3) a time budget t, and (4) the transition matrix of probability q that indicates the transition probability between samples. This algorithm starts by using s_i to initialize the current sample s_{curr} (line 1) and getting the starting time $time_s$ (line 2). Next, it iteratively

Algorithm 3: Initial sample selection.

```
Input  : $p_{top}$: the top program
         $p_{base}$: the base program
         $\alpha$: the weight controlling the preference between size reduction and attack surface reduction
         $\beta$: the weight controlling the preference between reduction and generality
         $k$: the weight adjusting the sample acceptance difficulty
         $I$: the set of inputs (usage profile)
Output : $p_{init}$: the initial sample
1  Function $getInitProgram(p_{top}, p_{base}, \alpha, \beta, k, I)$
2      $initDScore \leftarrow calculateDensity(p_{top}, p_{top}, \alpha, \beta, k, I)$
3      $newDScore \leftarrow calculateDensity(p_{top}, p_{base}, \alpha, \beta, k, I)$
4      if $newDScore \geq initDScore$ then
5          $p_{init} \leftarrow deadCodeEliminate(p_{base})$
6      else
7          $p_{init} \leftarrow deadCodeEliminate(p_{top})$
8      return $p_{init}$
9  Function $calculateDensity(p_{top}, p', \alpha, \beta, k, I)$
10     $p'_d \leftarrow deadCodeEliminate(p')$
11     $red \leftarrow getReductionScore(p_{top}, p'_d, \alpha)$
12     $gen \leftarrow getGenerality(p_{top}, p'_d, I)$
13     $dScore \leftarrow getDensityScore(gen, red, \alpha, \beta, k)$
14     return $dScore$
```

generates new samples (lines 3-11). In each iteration, it generates a new sample s_{new} by adding random noise to s_{curr} (line 4). To decide whether to accept s_{new} or not, it computes the density ratio $ratio$ based on the density function f and the transition matrix of probability q shown at line 5. If a random number rn between 0 and 1 is smaller than $ratio$, the algorithm accepts s_{new} (line 6-7). This shows that when s_{new} has a higher density value than s_{curr} and the transition from s_{curr} to s_{new} is more probable than the other way, s_{new} has a higher chance to be accepted. By accepting samples this way, this algorithm can collect more samples from higher-density regions of the distribution while occasionally visiting and collecting samples from lower-density regions. This explains why the MH algorithm can generate samples that approximate the given distribution effectively. When a new sample s_{new} is accepted, the algorithm adds it to the sample set S and updates s_{curr} (lines 8-9). When the time budget t is exhausted, the algorithm stops and returns the sample set S (lines 10-12).

Following the previous work [80, 94], we use the density function f shown below for an effective MH-based (also MCMC-based) exploration.

$$f(p') = \frac{1}{Z} exp(k \cdot O(p, p')). \tag{14}$$

The constant k in Formula 14 is used to adjust the difficulty of accepting new samples. Z is a normalizing constant that ensures that the sum of density values for all programs is 1 [32, 80].

4.2 Initial Sample Selection

Before using the MH algorithm for stochastic search, Mop generates an initial sample. Algorithm 3 details the approach. For initial sample generation, Mop takes as input the top program p_{top}, the base program p_{base}, the weights α, β, and k, and the inputs I. The algorithm starts with calculating the density values *initDScore* and *newDScore* of the input programs p_{top} and p_{base} (lines 2-3). Then it compares the density values *initDScore* and *newDScore* (line 4). If *newDScore* is higher than *initDScore*, the dead code eliminator tool [96] is employed to remove the dead code of p_{base} and generates the initial program p_{init}. Otherwise, it applies the same process to p_{top} and generates p_{init} (lines 4-7). Finally, p_{init} is returned as the initial sample (line 8).

Anytime Mop needs to get a density value for a sample p', it calls the utility function *calculate Density*$(\cdot)$ (lines 9-14) by first eliminating any dead code of p' (line 10), then getting the reduction and generality scores (lines 11 and 12), and finally computing the density value based on the scores (line 13).

4.3 Stochastic Optimization Based on Coverages

Function *MHSearchAtCov*$(\cdot)$ of Algorithm 4 details the first stage for optimization. In this stage, Mop performs MCMC-based stochastic search using a coverage-based mutation model.

The algorithm takes as input p_{top} (the top program), PI (the set of coverages induced from the execution of p over the usage profile inputs), p_i (the initial sample), α, β, and k (the weights), B (the set of must-handle inputs), I (the set of inputs in usage profile), and t_1 (the time budget for this stage). As output, it reports the sample with the highest density value (also the highest objective value) found in this stage.

The algorithm first gets the starting time $time_s$ and calculates the density value *currDScore* of the initial sample p_i (lines 2-3). Then it sets the current sample p_{curr} as p_i and *bestDScore*, the density value of the best sample, as *currDScore*, the current value (lines 4-5). After that, the algorithm converts the set of coverages into a list and initializes an array *bitvec* whose size is the list size (lines 6-7). Mop uses the array to track whether a coverage led by an input has been chosen to preserve in the current sample. If the coverage π_i derived from the execution against an input is retained by the current sample, then *bitvec*$[i]$ is 1, otherwise *bitvec*$[i]$ is 0. The algorithm initializes *bitvec* based on the coverage retained in the initial program p_i (lines 8-16). Because the statements exercised by inputs of B should be preserved in the debloated program, Mop ensures that the values of array items representing the coverages led by B are 1 (lines 13-14).

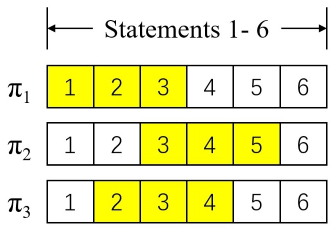


Fig. 5. An example of coverage overlap. Yellow cells are statements retained in the coverage (π_1, π_2, or π_3).

Next the algorithm uses a loop (lines 17-45) for its MCMC-based sample generation. The sampling process continues until the time budget t_1 is exhausted. In each iteration, Mop generates a new sample and decides whether to accept it or not. The way it generates a sample is by selecting a coverage led by an input and then using its mutation model to either delete the coverage code (i.e., the statements in the coverage) from the current sample or retain the statements in the sample.

Algorithm 4: Stochastic search based on coverage (stage-I optimization).

```
Input   :$p_{top}$: the top program
         $PI$: the set of coverages
         $p_i$: the initial sample
         $\alpha$: the weight controlling the preference between size reduction and attack surface reduction
         $\beta$: the weight controlling the preference between reduction and generality
         $k$: the weight adjusting the sample acceptance difficulty
         $B$: the set of must-handle inputs (optional)
         $I$: the set of inputs (usage profile)
         $t_1$: the maximum time (in minutes) for optimization
Output  :$p_b$: the best sample
1  Function $MHSearchAtCov(p_{top}, PI, p_i, \alpha, \beta, k, I, B, t_1)$
2      $time_s \leftarrow getCurrTime()$
3      $currDScore \leftarrow calculateDensity(p_{top}, p_i, \alpha, \beta, k, I)$
4      $p_{curr} \leftarrow p_i$                                    // Initialize current sample $p_{curr}$
5      $bestDScore \leftarrow currDScore$
6      $pi\_list \leftarrow toList(PI)$
7      $bitvec \leftarrow$ new int$[pi\_list.size()]$
8      if $getGenerality(p_{top}, p_i, I) == 1$ then
                                                                    // $p_i$ is top program w/o dead code
9          for $int\ i = 0; i < pi\_list.size(); i++$ do
10             $bitvec[i] \leftarrow 1$
11     else
12         for $int\ i = 0; i < pi\_list.size(); i++$ do
13             if $pi\_list.get(i)$ is in $getCoverages(p_{top}, B)$ then
14                 $bitvec[i] \leftarrow 1$
15             else
16                 $bitvec[i] \leftarrow 0$
17     repeat
18         $searchRange \leftarrow \{\}$                     // Indices of coverages suitable for mutation
19         for $int\ i = 0; i < pi\_list.size(); i++$ do
20             if $pi\_list.get(i)$ is not in $getCoverages(p_{top}, B)$ and
                 $!isAccidentallyRetained(pi\_list.get(i), pi\_list, bitvec)$ then
21                 $searchRange \leftarrow searchRange \cup i$
22         $idx \leftarrow getRandomIdx(searchRange)$
23         $bitvec[idx] \leftarrow 1 - bitvec[idx]$                                         // Flip a bit
24         $S \leftarrow \{\}$                                                    // The set of statements
25         for $int\ i = 0; i < pi\_list.size(); i++$ do
26             if $bitvec[i] == 1$ then
27                 $S \leftarrow S \cup getPrimitiveStmts(pi\_list.get(i))$
28         $p_{new} \leftarrow getReducedProgram(p_{top}, S)$
29         if $p_{new}$ compiles then
30             $dScore \leftarrow calculateDensity(p_{top}, p_{new}, \alpha, \beta, k, I)$
31             $accept \leftarrow false$
32             if $random() < (dScore \cdot q(p_{curr}, p_{new}) / currDScore \cdot q(p_{new}, p_{curr}))$ then
33                 $accept \leftarrow true$
34             if $accept$ then
35                 $currDScore \leftarrow dScore$
36                 $p_{curr} \leftarrow deadCodeEliminate(p_{new})$
37                 if $dScore > bestDScore$ then
38                     $p_b \leftarrow p_{curr}$
39                     $bestDScore \leftarrow dScore$
40             else
41                 $bitvec[idx] \leftarrow 1 - bitvec[idx]$                                 // Revert the bit
42         else
43             $bitvec[idx] \leftarrow 1 - bitvec[idx]$                                     // Revert the bit
44         $time_c \leftarrow getCurrTime()$
45     until $time_c - time_s \geq t_1$
46     return $p_b$
```

A naive mutation model would randomly select an input from I (but not in B) and obtain its coverage for mutation. Unfortunately, this can lead to invalid mutation due to the accidental code preservation caused by coverage overlap. To understand this, consider Figure 5, which shows three coverages, π_1, π_2, and π_3, that cover statements 1-3, 3-5, and 2-4 respectively. Suppose that π_1 and π_2 have been preserved in the current sample. Because the overlap of π_1 and π_2 includes all statements of π_3, π_3 is accidentally preserved in the sample as well. Later, if π_3 were to be chosen for mutation, the statements of π_3 (i.e., statements 2-4) would not be deleted, because π_1 and π_2 have already been preserved and the deletion can invalidate the preservation of π_1 and π_2. To address this issue, Mop computes a valid search range of the coverages by identifying those have been accidentally preserved in the current sample and others that must be preserved (corresponding to the must-handle inputs) and ignoring them all for mutation (lines 18-21).

Then the algorithm randomly selects a coverage from the range to mutate and flips the corresponding bit in *bitvec* (lines 22–23). It obtains a new sample p_{new} by scanning the coverages flagged as 1 (lines 24-27) and pruning p_{top} by removing statements not contained in any of those coverages (line 28).

After having a new sample p_{new}, the algorithm checks whether it compiles (line 29). It p_{new} does not compile, the algorithm reverts the flip (line 43) and continues sampling. Otherwise, the algorithm computes the density value *dScore* of p_{new} (line 30) and the density ratio (line 32) and uses a random number to decide whether to accept p_{new}, following the MCMC sample acceptance principle. Because each coverage in the search range of p_{curr} has an equal chance of being mutated to generate p_{new}, the transition probability $q(p_{curr}, p_{new})$ is computed as $1/|SearchRange_{curr}|$, where $|SearchRange_{curr}|$ is the number of coverages in the search range computed based on p_{curr} at lines 18-21. Likewise, the transition probability $q(p_{new}, p_{curr})$ is $1/|SearchRange_{new}|$, where $SearchRange_{new}$ is the search range that Mop computes based on p_{new} in a similar way. If p_{new} is accepted, the algorithm updates *currDScore* and p_{curr} (lines 34-36). And if this is a better sample than p_b (the current best sample) with a higher density value, Mop also updates *bestDScore* and the best sample p_b (lines 37–39). If p_{new} is rejected, Mop reverts the bit flipped (line 41). Then, it gets the current time $time_c$ and checks if the time budget t_1 is exhausted (lines 44-45). Finally, Mop returns p_b, the sample with the highest density value (line 46).

4.4 Stochastic Optimization Based on Coverage Segments

In the second stage, Mop performs coverage-segment-based stochastic optimization. The overall approach is as follows. Mop first obtains the segment set, and it selects and mutates a segment in the set for new sample generation. Like what it does in the first stage, Mop uses an array to keep track of the segments preserved in the current sample, and randomly selects a segment for mutation. It keeps sampling until the time budget is used up and reports the best sample as output.

Algorithm 5 shows how Mop works in this stage in detail. Mop first gets the starting time $time_s$ and calculates the density value *currDScore* of the starting sample p_i (lines 2-3), which is the best sample found in the first stage. Mop initializes the current sample p_{curr} as p_i and *bestDScore*, which saves the highest density value, as *currDScore* (lines 4-5). Then it calls function $toPrimitiveStmtsList(\cdot)$ (line 6) to get the statements of p_{top} that are not exercised by B. The algorithm converts the coverages into a list *pi_list* and excludes from the list any coverages derived from program execution against inputs in B (line 7). It also converts I into a list *I_list* for indexing purpose (line 8). Next, Mop generates the segment set, which consists of all the candidate segments for mutation. The way it does this is by first labeling each statement with the set of inputs that exercise it (the coverage input set) and then grouping statements with the same labels into segments. More specifically, to achieve this, Mop creates a map *stmt_label* of key-value pairs where the key is a statement and the value is the coverage input set (lines 9-15). It initializes an empty set for each statement (lines 10-11) and then checks each input individually to see whether the input exercises the statement. If the input does, the index of it is added to the set (lines 12-15). After creating the map, Mop gets *label_set* containing all the values (coverage input sets) in the map (line 16). It then

Algorithm 5: Stochastic search based on coverage segments (stage-II optimization).

```
   Input  : $p_{top}$: the top program
            $PI$: the set of coverages
            $p_i$: the starting program
            $\alpha$: the weight controlling the preference between size reduction and attack surface reduction
            $\beta$: the weight controlling the preference between reduction and generality
            $k$: the weight adjusting the sample acceptance difficulty
            $B$: the set of must-handle inputs (optional)
            $I$: the set of inputs (usage profile)
            $t_2$: the maximum time (in minutes) for optimization
   Output : $p_b$: the best sample
1  Function $MHSearchAtCovSeg(p_{top}, p_i, PI, \alpha, \beta, k, I, B, t)$
2      $time_s \leftarrow getCurrTime()$
3      $currDScore \leftarrow calculateDensity(p_{top}, p_i, \alpha, \beta, k, I)$
4      $p_{curr} \leftarrow p_i$
5      $bestDScore \leftarrow currDScore$
6      $S_{top} \leftarrow toPrimitiveStmtsList(p_{top}, B)$
7      $pi\_list \leftarrow toList(PI, B)$
8      $I\_list \leftarrow toList(I)$
9      $stmt\_label \leftarrow \{\}$
10     for int $i = 0; i < S_{top}.size(); i++$ do
11         $stmt\_label.put(i, \{\})$            // Initialize an empty label for each statement
12     for int $i = 0; i < pi\_list.size(); i++$ do
13         for int $j = 0; j < S_{top}.size(); j++$ do
14             if $S_{top}.get(j)$ is in $getPrimitiveStmts(pi\_list.get(i))$ then
15                 $stmt\_label.put(j, stmt\_label.get(j).add(getInputIdx(pi\_list, i, I\_list)))$
16     $label\_set \leftarrow getValueSet(stmt\_label)$
17     $segment \leftarrow \{\}$
18     $CS \leftarrow []$                        // Coverage segment set as a list
19     for each $label$ in $label\_set$ do
20         for each $tuple$ in $stmt\_label$ do
21             if $tuple.getValue() == label$ then
22                 $segment.add(S_{top}.get(tuple.getKey()))$
23         $CS.add(segment)$
24         $segment.clear()$
25     $bitvec \leftarrow$ new int$[CS.size()]$
26     for int $i = 0; i < CS.size(); i++$ do
27         $bitvec[i] \leftarrow 0$
28     $init(bitvec, p_i)$                    // Initialize bitvec using $p_i$
29     $p_b \leftarrow p_{curr}$
30     repeat
31         $idx \leftarrow getRandomIdx(bitvec.size)$
32         $bitvec[idx] \leftarrow 1 - bitvec[idx]$       // Flip a bit
33         $S \leftarrow \{\}$
34         for int $i = 0; i < bitvec.size; i++$ do
35             if $bitvec[i] == 1$ then
36                 $S \leftarrow S \cup CS.get(i)$
37         $p_{new} \leftarrow getReducedProgram(p_{top}, S)$
38         if $p_{new}$ compiles then
39             $dScore \leftarrow calculateDensity(p_{top}, p_{new}, \alpha, \beta, k, I)$
40             $accept \leftarrow false$
41             if $random() < (dScore \cdot q(p_{curr}, p_{new}) / currDScore \cdot q(p_{new}, p_{curr}))$ then
                                                       // $random() \in [0, 1)$
42                 $accept \leftarrow true$
43             if $accept$ then
44                 $currDScore \leftarrow dScore$
45                 $p_{curr} \leftarrow deadCodeEliminate(p_{new})$
46                 if $dScore > bestDScore$ then
47                     $p_b \leftarrow p_{curr}$
48                     $bestDScore \leftarrow dScore$
49             else
50                 $bitvec[idx] \leftarrow 1 - bitvec[idx]$   // Revert the bit
51         else
52             $bitvec[idx] \leftarrow 1 - bitvec[idx]$       // Revert the bit
53         $time_c \leftarrow getCurrTime()$
54     until $time_c - time_s \geq t_2$
55     return $p_b$
```

uses a loop (lines 19-24) to group the statements having the same labels into segments and save the results in *CS*. Each element of *CS* is a coverage segment, which is a set of statements.

For new sample generation, Mop initializes an array *bitvec* whose size is the number of segments contained in *CS* (lines 25-28). Depending on whether a segment is contained in the initial sample p_i or not, Mop sets a bit of *bitvec* as either 0 or 1 (line 28). It initializes the best sample p_b as the current sample p_{curr} (line 29). Next, Mop uses a loop for MCMC-based sampling until the time budget t_2 is exhausted (lines 30-54) and reports the best sample as the output (line 55). Unlike what it does in Algorithm 4 for coverage-based sampling, Mop does not have to consider anything about the overlap of segments, as every segment contains a distinct set of statements (the overlap between segments is empty).

The way of computing transition probability $q(\cdot)$ in this stage is different from that in the first stage. To explain how the computation works, let us first define a remove weight and an add-back weight for the mutation of each coverage segment.

$$remove_i = \frac{StmtNum_i}{CoverTestsNum_i}, \;\; add_i = \frac{1}{remove_i}. \tag{15}$$

In Formula 15, $remove_i$ and add_i are the weights for removing and adding back the coverage segment i. $StmtNum_i$ and $CoverTestsNum_i$ are the numbers of statements contained in that segment and the number of inputs exercising these statements (i.e., the size of their coverage input set), respectively. Using a remove weight defined in this way, Mop is more likely to remove a segment containing more statements that are exercised by fewer inputs. Such a removal is beneficial from an optimization perspective, as it supports generating a sample with the improved reduction-generality tradeoff by increasing the amount of code reduced while minimizing the decrease of generality. Conversely, for code add-back, Mop aims to minimize code preservation while significantly increasing generality, and it uses the reciprocal of the remove weight for add-back-based mutation. Depending on whether a segment $covseg_i$ has been preserved in the current sample, Mop uses either $remove_i$ (if preserved) or add_i (if not preserved) as the weight for $covseg_i$. Finally, Mop computes $Prob_i$ as the normalized weight for $covseg_i$, as shown in Formula 16.

$$Prob_i = \frac{weight_i}{\sum\limits_{0<i<n} weight_i}, weight_i \in \{remove_i, add_i\} \tag{16}$$

In this formula, n is the total number of coverage segments. Note that because the weights of the coverage segments can change in different iterations, Mop re-computes the weights in each iteration for new sample generation. If the current sample is p_{curr} and Mop mutates $covseg_i$ to generate a new sample p_{new}, the transition probability $q(p_{curr}, p_{new})$ is $Prob_i$. $q(p_{new}, p_{curr})$ is computed in a similar way.

4.5 Stochastic Optimization Based on Statements

Function $MHSearchAtStmt(\cdot)$ of Algorithm 6 details how Mop performs stochastic optimization with a statement-based mutation model. Mop starts by getting the starting time $time_s$, calculating the density value *currDScore* of the initial sample p_i, setting p_{curr} as p_i and *bestDScore* as *currDScore* (lines 2-5). Next Mop collects a set S_{top} of statements that are exercised by I but not by B as the candidates for mutation. Like what it does in the previous stages, Mop uses an array *bitvec* to track the preservation of each statement. The size of *bitvec* is the number of statements (lines 6-9). Mop also computes S_i, which contains all statements preserved in the starting program p_i and not exercised by B (line 10), and sets the best sample p_b as p_{curr} (line 11). The loop from lines 12-15 is used to update *bitvec*. For the statements in p_i, the corresponding bit values are 1. The bit values remain 0 for other statements that are in p_{top} but not in p_i.

Algorithm 6: Stochastic search based on statements (stage-III optimization).

```
Input   :$p_{top}$: the top program
         $p_i$: the starting program
         $PI$: the set of coverages
         $\alpha$: the weight controlling the preference between size reduction and attack surface reduction
         $\beta$: the weight controlling the preference between reduction and generality
         $k$: the weight adjusting the sample acceptance difficulty
         $B$: the set of must-handle inputs (optional)
         $I$: the set of inputs (usage profile)
         $t_3$: the maximum time (in minutes) for optimization
Output  :$p_b$: the best sample
1  Function $MHSearchAtStmt(p_{top}, p_i, PI, \alpha, \beta, k, I, B, t)$
2      $times_s \leftarrow getCurrTime()$
3      $currDScore \leftarrow calculateDensity(p_{top}, p_i, \alpha, \beta, k, I)$
4      $p_{curr} \leftarrow p_i$
5      $bestDScore \leftarrow currDScore$
6      $S_{top} \leftarrow toPrimitiveStmtsList(p_{top}, B)$
7      $bitvec \leftarrow$ new int$[S_{top}.size()]$
8      for $int\ i = 0; i < S_{top}.size(); i++$ do
9          $bitvec[i] \leftarrow 0$
10     $S_i \leftarrow toPrimitiveStmtsList(p_i, B)$
11     $p_b \leftarrow p_{curr}$
12     for $int\ i = 0; i < bitvec.size; i++$ do
                                                              // Initial bitvec
13         for $int\ j = 0; j < S_i.size(); j++$ do
14             if $S_i.get(j) == S_{top}.get(i)$ then
15                 $bitvec[i] \leftarrow 1$
16     repeat
17         $idx \leftarrow getRandomIdx(bitvec.size)$
18         $bitvec[idx] \leftarrow 1 - bitvec[idx]$                          // Flip a bit
19         $S \leftarrow \{\}$
20         for $int\ i = 0; i < bitvec.size; i++$ do
21             if $bitvec[i] == 1$ then
22                 $S \leftarrow S \cup S_{top}.get(i)$
23         $p_{new} \leftarrow getReducedProgram(p_{top}, S)$
24         if $p_{new}$ compiles then
25             $dScore \leftarrow calculateDensity(p_{top}, p_{new}, \alpha, \beta, k, I)$
26             $accept \leftarrow false$
27             if $random() < (dScore \cdot q(p_{curr}, p_{new}) / currDScore \cdot q(p_{new}, p_{curr}))$ then
                                                              // $random() \in [0, 1)$
28                 $accept \leftarrow true$
29             if $accept$ then
30                 $currDScore \leftarrow dScore$
31                 $p_{curr} \leftarrow deadCodeEliminate(p_{new})$
32                 if $dScore > bestDScore$ then
33                     $p_b \leftarrow p_{curr}$
34                     $bestDScore \leftarrow dScore$
35             else
36                 $bitvec[idx] \leftarrow 1 - bitvec[idx]$                  // Revert the bit
37         else
38             $bitvec[idx] \leftarrow 1 - bitvec[idx]$                      // Revert the bit
39         $time_c \leftarrow getCurrTime()$
40     until $time_c - time_s \geq t_3$
41     return $p_b$
```

In what follows, Mop uses a loop for stochastic optimization (lines 16-40), similar to what it does in the previous stage. Mop uses Formulae 15 and 16 to compute the remove and add-back weights for the mutation of each statement and the transition probability. Because in this stage a target element for mutation is a statement rather than a coverage segment, $StmtNum_i$ in Formula 15 is always 1, and the weight formula is reduced to Formula 17 shown below.

Algorithm 7: Robustness Improvement

```
Input  : $p$: the original program
         $p_d$: the debloated program
         $I$: the set of inputs (usage profile)
Output : $p_{br}$: the debloated program with improved robustness
1  Function $enhanceRobFuzz(p, p_d, I)$
2      $T_{ch} \leftarrow getCrashHangInputs(p_d, I)$     // Generate fuzzed inputs causing $p_d$ to crash/hang
3      $S_T \leftarrow \{\}$     // Initialize empty set for covered statements
4      for each $t_{ch} \in T_{ch}$ do
5          $S_t \leftarrow getCoveredPrimitiveStmts(p, \{t_{ch}\})$     // Get statements covered by input $t$
6          $S_T \leftarrow S_T \cup S_t$
7      $S_d \leftarrow toPrimitiveStmts(p_d)$     // Get primitive statements in $p_d$
8      $S_{br} \leftarrow S_d \cup S_T$
9      $p_{br} \leftarrow getReducedProgram(p, S_{br})$
10     $p_{br} \leftarrow deadCodeEliminate(p_{br})$
11     return $p_{br}$
```

$$remove_i = \frac{1}{CoverTestsNum_i}, \quad add_i = \frac{1}{remove_i}. \tag{17}$$

4.6 Robustness Enhancement of the Debloated Program

Function $enhanceRobFuzz(\cdot)$ in Algorithm 7 takes as input the original program p and the debloated program p_d, which is the best sample found in the previous optimization process. Following CovF's method [96], Mop employs the black-box fuzzing tool Radamsa [5] to generate fuzzed inputs based on the inputs in the usage profile and runs p_d against the fuzzed inputs to identify its robustness issues. To enhance the robustness of p_d, the algorithm reintroduces the statements from p that are exercised by the fuzzed inputs that have caused p_d to crash or hang. The output is p_{br}, a debloated program with improved robustness.

Specifically, the algorithm first generates fuzzed inputs, identifies those that cause p_d to crash or hang, using $getCrashHangInputs(\cdot)$, and records such inputs in T_{ch} (line 2). By default, all the profile inputs are used as seeds for fuzzing, and 10 fuzzed inputs are generated for each seed. Following CovF's method, $getCrashHangInputs(\cdot)$ relies on the exit code and execution timeout to determine if p_d crashes or hangs. The algorithm initializes an empty set S_T used to collect statements of p exercised by crash/hang-triggering inputs (line 3). For each input $t_{ch} \in T_{ch}$, it uses $getCoveredPrimitiveStmts(\cdot)$ to retrieve the statements of p that are executed when running with t_{ch} and adds the statements to S_T (lines 4–6). Then, the algorithm extracts the set S_d of statements preserved in p_d using $toPrimitiveStmts(\cdot)$ (line 7). By taking the union of S_d and S_T, it obtains S_{br}, which comprises statements to retain in the final enhanced program (line 8). Given the original program p and the statements S_{br} to retain, the algorithm calls $getReducedProgram(\cdot)$ to produce the robustness-enhanced program p_{br} by retaining statements in S_{br} and removing those not to retain (line 9). Recall that Mop operates by retaining and removing primitive statements. It relies on a dead code eliminator to reduce any redundant compound statements. So finally, the algorithm calls *deadCodeEliminate*$(\cdot)$ to eliminate any redundant code (such as the dangling if-statement and unused variables) of p_{br} (line 10) and returns the debloated program (line 11).

5 Evaluation

To assess the effectiveness of Mop, we have implemented it in a prototype tool and applied it to a previous benchmark [96] that contains 25 programs and a large program *postgresql-12.14* [61] for debloating. *postgresql-12.14* is an open-source, highly extensible, and powerful database system containing nearly 1M lines of code. We seek to answer the following research questions:

- **RQ1**: What are the reduction, generality, and tradeoff scores of the debloated programs generated by Mop? With different α and β values controlling the debloating preferences, can Mop generate programs with different scores?
- **RQ2**: How does Mop compare with Debop in terms of the reduction, generality, and tradeoffs achieved?
- **RQ3**: How does Mop compare with Chisel and Razor in terms of the size and attack-surface reductions?
- **RQ4**: How effective is each of Mop's optimization stages?
- **RQ5**: How effective is Mop's post-processing step in improving the robustness of the debloated program?

The first four RQs (i.e., RQ1-RQ4) focus on assessing Mop's optimization and reduction abilities without considering robustness improvement. RQ1 investigates the effectiveness of Mop in terms of the reduction, generality, and tradeoff scores under different α and β weights; RQ2 compares Mop with Debop [94], the state-of-the-art stochastic optimization-based debloating approach; In RQ3, we investigate the reduction abilities of Mop by comparing it with Chisel [35] and Razor [71], two state-of-the-art reduction-oriented techniques. In this comparison, we used Chisel and Razor to produce debloated programs based on the inputs correctly handled by Mop's debloated program and compared their sizes and attack surfaces; In RQ4, we perform a component analysis by assessing the contribution of each of Mop's optimization stages and conducting ablation experiments. The last RQ (i.e., RQ5) specifically assesses Mop's effectiveness in improving the robustness of the debloated programs. We did experiments to understand Mop's ability to enhance the debloated program's resilience against unexpected inputs.

We note that the O-Score metric, which quantifies the tradeoff between reduction and generality, is a core metric, as it represents the overall effectiveness of debloating from an optimization perspective. In RQ1, RQ2, and RQ4, where we evaluated Mop's and Debop's optimization abilities, we reported the O-Score results. We additionally showed the reduction and generality results separately in these RQs to help the readers understand how the tradeoffs are achieved under different weights. Because RQ3 focuses on the reduction abilities and RQ5 investigates robustness, we do not report the O-Scores in these RQs.

5.1 Implementation

Mop performs debloating at the source code level. We implemented a prototype of Mop using a combination of C++, Java, Bash, and Python. Mop uses the llvm-cov tool [55] to get the coverage π_i induced from the program's execution with each input i by tracking the statements exercised by i. Specifically, for each input, Mop uses a file to save its coverage, which contains the execution status of each line of code in the program p. If a line of code is exercised by an input, the line is marked as 1 in the coverage file; otherwise, it is marked as 0. To understand which statement is exercised by which input, Mop established a mapping from each statement to its starting and ending lines of code. To get this mapping, Mop uses *Clang* (v.9.0.0) [54] to build the abstract syntax tree (AST) for p and traverses the AST to identify all the primitive statements and keep track of the starting and ending positions (in terms of the line and column numbers in the source file) for each statement. For debloating, Mop targets the removal of code in p's methods. To generate p_{top}, it deletes the lines of code contained in the bodies of methods defined in p and not exercised by any inputs. Similarly, to generate p_{base}, Mop deletes the method code not exercised by any must-handle inputs (optionally provided as a subset of the given inputs). We reused code from the Debop and DomGad tools to implement the MCMC algorithm used for each optimization stage in Mop.

To compute scores of any reduced program generated during optimization, Mop calculates size reduction by measuring the fraction of statements removed. Mop does not use program bytes for this purpose because the removal of statements, although beneficial, can often lead to negligible variance of binary size, and tiny size changes can easily cause sample rejection and impede effective exploration, especially for fine-grained optimization (e.g., the third stage of Mop). Before calculating size reduction, Mop eliminates dead code with a utility of the Cov tool [96] for an accurate quantification of program size. As with the previous approaches [35, 71, 94], to measure the attack surface, Mop counts the number of ROP (Return Oriented Programming) gadgets contained in the program binary using the ROPgadget tool [78]. To obtain a program binary, Mop uses *Clang* to compile the source code with the *-O3* option. To generate the fuzzed inputs that guide code augmentation for the robustness enhancement of debloated programs, Mop leverages the black-box fuzzer Radamsa [5]. By default, all the inputs in the usage profile are used as the seeds for fuzzing. Mop also allows fuzzing based on a specified set of inputs.

5.2 Experimental Setup

We next introduce the benchmark programs, the evaluation metrics, the techniques used for comparison, the setup work that ensures the programs and inputs can work correctly for the experiment, the configuration of the tools, and the experiment environment.

5.2.1 Benchmark Programs and Inputs. The benchmark consists of 10 utility programs, 15 SIR programs, and the PostgreSQL program v12.14 (*postgresql-12.14*).

The 10 utility programs are used in the evaluation of many previous debloating techniques [35, 71, 95, 96]. Following their practice, for these programs, we used the merged source files provided in [73]. A merged source file is an all-in-one-file version of the program with all the source code merged by the CIL merger [22]. In the benchmark provided in [73], each utility program is associated with 10 sets of inputs collected from different online websites. We included all these inputs in the usage profile of the utility for debloating.

In addition to the 10 utility programs, our benchmark also includes 15 SIR programs used in the previous study [96] representing various types of programs from the lexical analyzer (printtokens) to the Unix shell (bash-2.05) and to the text editor (vim-5.8). In the original benchmark [73], each program is associated with two sets of inputs, train and test, containing hundreds to tens of thousands of inputs representing various usage scenarios. We considered the union of train and test as the usage profile of each program for its debloating. Also, for these programs, we used the merged source files provided in [73].

Finally, our benchmark has an additional C-based application *postgresql-12.14* [68] that contains over 936K LoC. The size of the application is about 6 times that of the largest remaining program in the benchmark. We included *postgresql-12.14* to investigate how Mop and other techniques handle a large program. We searched for inputs associated with *postgresql-12.14* provided in the official GitHub repository [68]. Following the suggestion provided in the README file for testing [70], we considered all 306 regression tests and 94 isolation tests [69] provided in the repository as the usage profile inputs. The 306 regression tests comprise 193 core tests, 61 tests designed for the embedded SQL preprocessor for C programs, and 52 tests that enable users to write PostgreSQL functions in four languages: SQL, Perl, Tcl, and Python. Because the CIL tool and its merger [21] fail to generate a single source file for *postgresql-12.14* (they have not been maintained for years), we use the original PostgreSQL program for debloating.

Table 1. The benchmark programs used in our evaluation.

Type	Program	LOC	#Func	#Stmt	#Inputs
Small	PRINTTOKENS2	824	19	341	4058
	PRINTTOKENS	1069	18	396	4073
	REPLACE	938	21	416	5542
	SCHEDULE2	604	16	238	2710
	SCHEDULE	537	18	211	2650
	TCAS	382	9	162	1608
	TOTINFO	586	7	265	1052
Medium	BZIP2-1.0.5	11782	97	6154	59
	CHOWN-8.2	7081	122	3765	111
	DATE-8.21	9695	78	4228	174
	FLEX-2.5.4	15518	162	6704	670
	GREP-2.19	22706	315	10977	145
	GREP-2.4.2	16203	131	8437	806
	GZIP-1.2.4	8694	91	4049	81
	GZIP-1.3	8882	97	4287	213
	MKDIR-5.2.1	5056	43	1804	50
	RM-8.4	7200	135	3835	84
	SED-4.1.5	18866	247	9179	370
	SORT-8.16	14264	233	7805	117
	SPACE	8215	136	4376	13549
	TAR-1.14	30477	473	13995	84
	UNIQ-8.16	7020	65	2086	72
Large	BASH-2.05	60892	1003	27646	845*
	MAKE-3.79	32492	248	12901	1832
	VIM-5.8	141956	1699	66080	975
	POSTGRESQL-12.14	936892	19098	155027	400

* The number of inputs used in the previous evaluation [96] is 1061. We did not use 216 (1061-845) of the inputs, as we found that, for these inputs, the top program of BASH-2.05 failed to behave correctly possibly due to the sub-processes created by the program that make the llvm-cov tool's code tracking inaccurate.

Table 1 shows the 26 programs in terms of their size types (*Type*) as *small*, *medium*, and *large*, the names (*Program*), the number of lines of code in their source files (*LOC*), the number of functions defined (*#Func*), the number of statements (*#Stmt*), and the number of inputs (*#Inputs*) representing the usage profile of the program. *Small* programs are the 7 SIR programs classified as small in the previous work [96]. As shown in Table 1, the sizes of these programs are much smaller than those of the other programs. *Large* programs are *bash-2.05*, *make-3.79*, and *vim-5.8*, which are the utility and SIR programs that are of the largest sizes, and *postgresql-12.14.* The remaining 15 programs are *Medium.*

We slightly adapted some programs, their inputs, and the testing environment to ensure that program execution with the inputs is valid and that the third-party tools that Mop relies on for debloating can work as expected. Specifically, because the *mkdir-5.2.1* program can create an abundance of garbage folders as the execution result, we wrote a small program that regularly deletes the garbage folders with the *rm* command to avoid excessive disk space consumption while debloating. Also, in cases where the folders are created outside of a target directory due to the misbehavior of any debloated *mkdir*, we created a workaround to ensure the deletion of garbage folders by using *rsync* to back up a clean version of the root directory and restore it regularly.

Some inputs I_o of *chown-8.2* involve changing the ownership of a target file from a current non-root user to the root user. For the debloating experiment, we ran Mop and other techniques as the root user in a Docker environment [24] (as we will explain in Section 5.2.5). Because this environment can change the original owner of the target files contained in I_o to the root user, we adapted the script of each input of I_o by setting the owner of the target files back to a non-root user before executing *chown-8.2*.

Some inputs of *date-8.21* (e.g., *date 091619552014*) that involve converting a long number into a date can change the current system time. The system will undo the change shortly. The time change, though short, can cause problems for the program execution with other inputs. To fix it, we adapted the scripts of these inputs by adding *hwclock -s* to force system time synchronization.

A possible misbehavior of the debloated programs of *bzip2-1.0.5*, *gzip-1.2.4*, *gzip-1.3*, and *tar-1.14* is that the program fails to compress a target file as requested but instead compresses a non-target directory such as the working directory or the whole root directory. To avoid such a detrimental effect, we changed the debloating script. Before running the utilities, we set the files and directories that will not be changed during the debloating process in the working directory together with */bin* and */usr* in the root directory as immutable with *chattr +i*. Once the debloating process is done, we revert these changes with *chattr -i*.

For two programs *vim-5.8* and *bash-2.05*, the llvm-cov tool used by Debop and Mop fails to track all the statements exercised by certain inputs accurately. In particular, *vim-5.8* uses a fork function to spawn child processes when running with certain inputs, and llvm-cov does not accurately track code exercised by the child processes. We fixed the issue by tracking all the statements exclusively covered by the forked processes of *vim-5.8* and avoided their deletion by Mop and Debop in the experiment. *bash-2.05* has a similar problem, and we fixed it in a similar way. For 216 inputs of *bash-2.05*, we found that llvm-cov does not accurately record the coverage. Because we do not understand the failure of llvm-cov, we discarded the inputs. More minor details about benchmark setup can be found at [13]. We will explain how we selected the must-handle inputs in Section 5.2.4.

5.2.2 Simplified Notations of the Metrics. For the ease of presentation, we use simplified notations of the metrics used to evaluate a debloated program. Given the original program p_{ori}, the top program p_{top}, and a reduced program p', we use the notations presented in Table 2.

5.2.3 Tools for Comparison. We compared Mop with Debop, the state-of-the-art optimization-based debloating technique that also uses inputs to represent features. Since the original Debop does not take an optional set of must-handle inputs, we implemented Debop-M, a variant of Debop that uses must-handle inputs and preserves in the debloated program statements exercised by these inputs. When the must-handle inputs are not provided in our experiment, we compared Mop with Debop. In other experiments, we compared Mop with Debop-M.

We note that there are other optimization-based techniques [8, 34, 95] not included for comparison. Specifically, we excluded DomGad [95], another generality-aware debloating technique, as the technique is subdomain-based and does not rely on specific inputs for generality quantification. We also excluded MoMS [8] and PolyDroid [34]. The former is a semi-automated technique that

Table 2. Simplified notations for metrics used to evaluate a debloated program p' relative to the original program p_{ori} and the top program p_{top}.

Metric Type	**Relative to p_{ori}**	**Relative to p_{top}**
SR-Score	SR-Score$_{\text{ori}}$ = sred(p_{ori}, p')	SR-Score$_{\text{top}}$ = sred(p_{top}, p')
AR-Score	AR-Score$_{\text{ori}}$ = ared(p_{ori}, p')	AR-Score$_{\text{top}}$ = ared(p_{top}, p')
R-Score	R-Score$_{\text{ori}}$ = red(p_{ori}, p')	R-Score$_{\text{top}}$ = red(p_{top}, p')
G-Score	G-Score = gen(p_{ori}, p', I)	NA
O-Score	O-Score$_{\text{ori}}$ = $O(p_{\text{ori}}, p')$	O-Score$_{\text{top}}$ = $O(p_{\text{top}}, p')$

Notes: The G-Score is computed only relative to p_{ori}, measuring generality based on the input set I, with all inputs assigned equal weight. Since p_{ori} and p_{top} exhibit identical behavior for the given inputs, only p_{ori} is used in the G-Score notation.

requires experts to manually validate features and traces. The latter is designed for Android app debloating and for performance optimization only. Neither tool is available to use.

We additionally compared Mop with Chisel [35] and Razor [71], two state-of-the-art reduction-oriented (not optimization-based) debloating techniques, to help us understand Mop's reduction ability. To enable a reasonable comparison, we used Chisel and Razor to produce debloated programs based on not all the inputs in the usage profile but those that the debloated program produced by Mop can correctly handle. We compared the sizes of the debloated programs. Note that the comparison is not direct, as Mop is not reduction-oriented.

5.2.4 Experiment Methods and Parameters. We next discuss the methods and parameters used in the experiments of optimization-based debloating performed by Mop and Debop, the comparison between Mop and the two reduction-oriented techniques, and the component analysis of Mop.

Optimization-based debloating. We ran Mop and Debop with the same time budget for debloating. Mop operates in three stages. For each of the 22 programs of small and medium sizes, we ran Mop for at most one hour, using a 20-minute budget for each stage. To have a fair comparison with Mop, we ran Debop also for at most one hour for each of these programs. Note that the time budget is applied to the optimization process only and not the generation of p_{top}, which takes less than a minute (for both Mop and Debop).

For the four large programs: *make-3.79*, *bash-2.05*, *vim-5.8*, and *postgresql-12.14*, because the program execution against all the inputs for generality quantification is more costly, a 1-hour time budget for Mop can lead to insufficient exploration. So we used a 4-hour budget for each stage and ran Mop for at most 12 hours. To ensure a fair comparison, we also used a 12-hour budget for Debop for these programs.

We used a time-based convergence condition and did not experiment with more complicated conditions for optimization based on for example manually observing the stability of the Markov chain [79] and the convergence metrics [27, 28]. The former observation method depends on a manual definition of stability and is subject to human biases, while the latter condition may not precisely determine whether the Markov chain has truly converged [28] and can cause a significant overhead due to multiple runs of the Markov chain for variance comparison [27]. We leave to future work the investigation of more effective and efficient convergence conditions for better stochastic optimization.

We ran both tools with different combinations of α and β values to investigate whether they can achieve different tradeoffs with different preferences of the debloating factors. Specifically, we experimented with three α values (0.25, 0.5, and 0.75) and nine β values (from 0.1 to 0.9) and ran Mop and Debop for a total of 27 (3×9) trials to debloat each of the 22 programs. Since a debloating run for the four large programs is more expensive (12 hours for one trial), we used three α values (0.25, 0.5, and 0.75) and fewer β values (0.25, 0.5, and 0.75).

To assess the core optimization abilities, we focused on evaluating Mop and Debop in the scenario where no must-handle inputs are provided. We chose to do so because the use of must-handle inputs can lead to a smaller search space for debloating, as the tools are designed to reduce code not exercised by such inputs, and a small search space cannot fully reflect a technique's optimization ability. We did however evaluate Mop and Debop-M in the scenario where the must-handle inputs are provided, although we do not claim the evaluation to be the focus of our analysis.

Given that a debloating run is expensive, for experiments with must-handle inputs, we only chose a subset of the programs comprising three programs (*mkdir-5.2.1*, *date-8.21*, and *tar-1.14*) from Util, three programs (*gzip-1.3*, *sed-4.1.5*, and *vim-5.8*) from LSIR, *totinfo* from SSIR, and *postgresql-12.14*. The programs from either Util or LSIR are chosen to be of diverse sizes from that set. We randomly selected *totinfo* from SSIR containing all small programs (LoC < 1.1K).

As for the preparation of the must-handle inputs, recall that each Util program is associated with 10 sets of inputs collected from online websites [96], and we chose as the must-handle inputs the set whose size in terms of the number of inputs is the closest to the average. Each SIR (either SSIR or LSIR) program is associated with two sets of inputs, train and test [96], and we chose inputs from the train set, which has a smaller number of inputs than the test set, as the must-handle inputs. For *postgresql-12.14*, we chose the 193 core regression tests as the must-handle inputs. Like what we did for the experiments without the must-handle inputs, we ran the tools for debloating in multiple trials with different α and β values and used the 1-hour and 12-hour time budgets for small/medium and large programs respectively.

To assess whether Mop achieves a statistically significant performance improvement over Debop, we conducted the one-sided Wilcoxon signed-rank test [92], which tests whether two paired samples differ significantly without assuming data normality. This choice is motivated by the following considerations. First, the Wilcoxon signed-rank test assumes that the data to compare are paired, which holds in our context, as the test is for comparing Debop's and Mop's results (specifically, their O-Scores) in the same setting (based on the same original program, the same usage profile, and the same debloating parameters). The two results (O-scores) naturally form a pair. Second, the Wilcoxon signed-rank test uses a one-sided alternative hypothesis (H_1: Mop's O-Score > Debop's O-Score) that is suitable for our context. By design, Mop represents a more advanced technique, as it uses a sophisticated exploration method, inheriting Debop's fine-grained exploration (Mop's third stage) but extending it with two more stages (Mop's first and second stages) to follow a coarse-to-fine search principle. Moreover, it uses an additional preprocessing step in identifying an advantageous initial sample for better optimization. Since Mop is more advanced, we adopt a signed-rank test with a one-sided alternative hypothesis to concentrate on testing *improvement*, specifically whether Mop is statistically more effective than Debop, rather than a two-sided test, which only focuses on *difference*. Third, the Wilcoxon signed-rank test does not assume normal distribution of data, which is also appropriate, as the O-Scores of debloated programs derived from stochastic optimization processes are not guaranteed to follow a normal distribution.

We applied the one-sided Wilcoxon signed-rank test to the O-Scores of debloated programs generated by Mop and Debop in the debloating experiments with and without must-handle inputs used. Specifically, we tested the average O-Scores per program, using results from the 26 programs without must-handle inputs and the 8 programs with must-handle inputs, across various combinations of α and β parameters. For each comparison, we report the p-value and the effect size r values [23] commonly used for the Wilcoxon signed-rank test [26].

It is also necessary to study the performances of Mop and Debop under different time budgets. Therefore, we conducted debloating experiments with varying time budgets and compared Mop's and Debop's runtime performances. We note that it is also expensive to run all 26 programs under varying time budgets. So we used the same subset of the benchmark comprising 8 programs and the same α and β values for this evaluation. We ran Mop and Debop under three relative time budgets ($0.5T$, T, and $1.5T$), where T represents the baseline time budget for each program. To focus on assessing the core optimization capabilities of Mop and Debop under these budgets, we do not use must-handle inputs for debloating.

Comparison between Mop and reduction-oriented techniques. Unlike Mop, Chisel and Razor are reduction-oriented techniques, and they do not consider anything about the reduction-generality tradeoffs for debloating. We acknowledge that Mop is different from the two techniques but nevertheless followed the previous work [94] and compared Mop with Chisel and Razor to have a deep understanding of Mop's reduction ability. We excluded the four large programs and used only the 22 remaining programs for the experiment due to the inefficiency or failure of Chisel and Razor in handling the large programs for debloating. Specifically, for three of the large programs (without *postgresql-12.14*), Chisel was highly inefficient and did not produce any reduced program within a 12-hour time limit. For *make-3.79* and *bash-2.05*, Razor generated debloated programs that do not behave correctly for all inputs, but only 9% and 81% of the inputs. For *vim-5.8*, Razor failed to generate any debloated program due to an error arising when merging the trace log files for coverage analysis. We also excluded *postgresql-12.14*, for which the CIL merger [22] fails to generate a one-file merged version. Chisel cannot handle the original multi-file program for debloating due to its problems of handling the macro syntax. Razor also failed to deal with the original program, as we found that it has problems handling the multiple binary files that depend on one another (e.g., the *execl*() function in file A calls functions from file B).

Recall that we ran Mop for the debloating of a program in 27 trials with different α and β values. Following the previous evaluation [94], we logged for each trial the exact set of inputs for which the debloated program generated by Mop behaved correctly and provided Chisel and Razor with the same inputs for debloating. We ran Chisel and Razor with the same time budget used by Mop and calculated the reduction scores achieved by all tools. In extreme cases where no inputs were logged (which means Mop generated a nearly empty program in pursuit of the maximum reduction), we did not run Chisel or Razor for comparison.

We compared the size and attack surface reduction achieved by the techniques. Since Razor is binary-based, we compiled the debloated program generated by Mop and compared its size with that of the debloated program generated by Razor in terms of the number of bytes and ROP gadgets reduced in the program binaries. Because Razor needs debug information preserved in the program binary for heuristic exploration, we compiled the original program with *Clang* using the *-g* option to obtain a binary for Razor's debloating. For a fair comparison with Razor, we also used *Clang* with the *-g* option to compile the debloated program generated by Mop. Unlike Razor, Chisel is source-based, and we used *Clang* with the *-O3* option to compile the debloated programs generated by Mop and Chisel for reduction comparison, as we did in other experiments for evaluating Mop and Debop.

Component analysis of Mop. For a component analysis of Mop, we (1) analyzed the contribution of each of Mop's optimization stages and (2) did additional ablation experiments.

For (1), we quantified the increase (in percentage) of the O-Score that each stage of Mop contributed in the debloating experiments. Let o_{init}, o_1, o_2, and o_3 be the O-Scores of the initial sample and the best samples after the 1st, 2nd, and 3rd stages. The contributions of the three stages are quantified as $\frac{o_1 - o_{init}}{o_3 - o_{init}}$, $\frac{o_2 - o_1}{o_3 - o_{init}}$, and $\frac{o_3 - o_2}{o_3 - o_{init}}$, respectively.

For (2), we created three new techniques, Mop-$S_{2,3}$, Mop-$S_{1,3}$, and Mop-$S_{1,2}$, by removing each of the optimization stages of Mop respectively. For example, Mop-$S_{2,3}$ is a variant of Mop that removes the 1st stage and uses only the last two for optimization. We ran the debloating experiments with each of Mop-$S_{2,3}$, Mop-$S_{1,3}$, and Mop-$S_{1,2}$ using the same time budget and initial sample selection strategy that Mop used. Because each new technique has two stages, we allocated each stage half of the time budget for optimization. We used no must-handle inputs for these experiments to focus on investigating the optimization capabilities of these techniques in a full search space.

We found that the selection of the initial sample can play a crucial role in the debloating process in many cases. If the initial sample is close to be the "optimal" sample (in terms of the O-Score), the subsequent optimization based on the initial sample has not much to do, leaving all techniques (almost) equally ineffective. To have a deeper understanding of the optimization abilities of each stage, we created two additional extreme settings where β values are set as 0.1 and 0.9 and an unfavorable initial sample is chosen in each setting, that is, p_{top} (i.e., the program only comprising code covered by all the inputs of the usage profile) is selected as the initial sample when β is 0.1 and p_{base} (i.e., the program only comprising code covered by the must-handle inputs from the usage profile) as the initial sample when β is 0.9. We set α as 0.5 in both settings and ran and compared the variants of Mop. The two extreme settings are created only to weaken the influence of the initial sample selection on the debloating effectiveness, which allows us to evaluate the optimization abilities of the three stages in a better way.

Robustness Improvement. To investigate Mop's robustness improvement ability, we did experiments with the previous subset of the benchmark, comprising 7 programs from the SIR (*totinfo*, *gzip-1.3*, *sed-4.1.5*, and *vim-5.8*) and Util (*mkdir-5.2.1*, *date-8.21*, and *tar-1.14*) benchmarks, together with *postgresql-12.14*. We ran Mop's post-processing step to augment the 8 debloated programs derived from the experiment where α and β are set as 0.5 and no must-handle input is considered. To obtain fuzzed inputs, Mop used as the seeds either all inputs or a subset of the inputs from the usage profile. Specifically, for each of the programs excluding *postgresql-12.14*, Mop used the same set of inputs that CovF [96] used in its previous evaluation. (Note that the seeds used by CovF comprise either the whole set or a subset of the usage profile inputs used in this evaluation.) For *postgresql-12.14*, Mop used the 194 core tests (Section 5.2.1). Then, following the experiment for evaluating CovF [96], we evaluated the augmented versions of the debloated programs by using Radamsa [5] to generate 10 fuzzed variants based on each seed input and testing the programs against the fuzzed inputs.

5.2.5 *Experiment Environment.*

We did the experiments in Docker containers set up on two machines running Ubuntu-18.04 with 260 GB RAM and 32 AMD-Opteron 1.4 GHz processors. We used 16 containers (8 on each machine) to run the debloating experiments for the 22 programs that are of small and medium sizes and two containers (one on each machine) for the four programs of larger sizes. We note that because we chose to run multiple tools in many trials for debloating, and even one trial can take from one to a few hours to finish, our experiments are very expensive. The machine time taken to generate all the debloated programs by all tools for all experiments is over 293 days. The actual running time is more than 80 days.

5.3 Results for RQ1: What are the reduction, generality, and tradeoff scores of the debloated programs generated by Mop? With different α and β values controlling the debloating preferences, can Mop generate programs with different scores?

5.3.1 Result of Debloating without the Must-Handle Inputs. Tables 3-5 show the average SR-Score (size reduction), AR-Score (attack surface reduction), R-Score (code reduction), G-Score (program generality), and O-Score (objective function value) of the debloated programs generated by Mop (and Debop). The average scores of Tables 3-5 were calculated by taking the arithmetic mean of the results from each experimental run, rather than directly averaging the Small, Medium, and Large values. These programs were generated in multiple experiments where no must-handle inputs are assumed and different α and β values are used. While α adjusts the relative importance of two correlated reduction factors, β is a key weight that controls the preference of reduction and generality whose tradeoff quantifies the effectiveness of debloating. We present the average scores with β values shown in different ranges. More specifically, Table 3 shows the average scores for all β values (from 0.1 to 0.9). Tables 4 and 5 present scores when β values are less than 0.5 (from 0.1 to 0.4) and are no less than 0.5 (from 0.5 to 0.9), respectively. The results of Tables 4 and 5 can help us understand how Mop behaves when reduction is weighted more than generality ($\beta < 0.5$) and vice versa ($\beta \geq 0.5$).

Note that the evaluation of Mop is based on the O-Score computed per run, and the final result is the average of those run-specific scores. The separately reported averages of reduction and generality are also computed based on their specific scores across all runs. They are for descriptive purposes and are not used to derive or validate the average O-Scores. These reduction and generality scores offer finer-grained insights into Mop's behavior and can help us assess whether Mop aligns with the specific debloating needs (e.g., maximizing reduction versus preserving generality).

According to Table 3, Mop produced debloated programs by reducing 68% statements (SR-Score$_{ori}$) and 54% attack surface (AR-Score$_{ori}$) and achieved a 0.61 reduction score (R-Score$_{ori}$) when the original program is used as the reference for reduction calculation. The generality of the debloated programs is 0.58. The tradeoff O-Score$_{ori}$ between reduction and generality is 0.76. This result shows that Mop can delete a significant amount of code while still preserving 58% generality of the original program to achieve a good tradeoff between reduction and generality.

The use of p_{top}, which contains less code than the original p, as the reference program allows us to exclude the influence of Mop's preprocessing stage when evaluating Mop's debloating effectiveness. The R-Score$_{top}$ quantifies the reduction contributed solely by the optimization process and thus reflects the core optimization ability of Mop. Our results show that Mop's R-Score$_{top}$ is 0.48, indicating that the optimization process can prune approximately half of the code while still maintaining a high generality. The O-Score$_{top}$ is 0.72 and is only slightly lower than O-Score$_{ori}$ 0.76.

When β is less than 0.5, that is, code reduction is weighted more than generality, Mop can do aggressive code pruning to produce a debloated program with much less generality. According to Table 4, Mop achieved a 0.93 reduction score (using the original program as the reference) and produced a debloated program preserving 10% generality of the original program. The tradeoff score is 0.73. This result suggests that Mop can focus on reduction when needed and prune a significant amount of code for debloating. Note that this is in the situation where no must-handle inputs are provided, and Mop can be very aggressive at code pruning and preserve a low generality of the original program in pursuit of a high objective score.

One can see that, when β is less than 0.5, Mop achieved a high reduction score. In fact, when β is small, Mop often chooses the base program p_{base} as the initial sample and performs optimization with a focus on adding back code from p_{top} to the sample to increase the sample's generality without however significantly increasing the size and the attack surface. For small and medium programs, because adding back code can easily result in a program with a substantial increase of size or attack surface (via for example the 1st- or 2nd-stage optimization, which is aggressive) or invalid programs that for example do not compile (via the 3rd-stage fine-tuned optimization), and these programs can be easily rejected during the search, Mop ends up generating debloated programs preserving little code of the original program. For the four large programs, however, the size reduction score is 0.86, and the generality of the debloated programs is 0.32. This shows that for large programs Mop can generate a debloated program preserving about 1/3 of the original program's generality while still pruning a substantial amount (or 86%) of the code.

Table 3. Average scores of debloated programs generated by Mop and Debop in experiments where a combination of nine β values (between 0.1 and 0.9) and three α values (0.25, 0.5, and 0.75) are used. **Bold** result indicates the highest score. Note that higher O-Score (O-Score$_{ori}$ or O-Score$_{top}$) is better.

Type	SR-Score$_{ori}$		SR-Score$_{top}$		AR-Score$_{ori}$		AR-Score$_{top}$		R-Score$_{ori}$		R-Score$_{top}$		G-Score		O-Score$_{ori}$		O-Score$_{top}$	
	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop
Small	**0.48**	0.10	**0.47**	0.07	**0.37**	0.09	**0.37**	0.09	**0.43**	0.10	**0.42**	0.08	0.59	**0.90**	**0.70**	0.54	**0.70**	0.54
Medium	**0.77**	0.56	**0.55**	0.14	**0.62**	0.35	**0.48**	0.11	**0.70**	0.46	**0.52**	0.13	0.56	**0.85**	**0.79**	0.71	**0.74**	0.55
Large	**0.63**	0.42	**0.41**	0.05	**0.48**	0.25	**0.31**	0.00	**0.56**	0.34	**0.36**	0.03	0.76	**0.99**	**0.74**	0.67	**0.66**	0.51
Average	**0.68**	0.41	**0.52**	0.11	**0.54**	0.27	**0.44**	0.10	**0.61**	0.34	**0.48**	0.11	0.58	**0.87**	**0.76**	0.66	**0.72**	0.55

Table 4. Average scores of debloated programs generated by Mop and Debop in experiments where a combination of four β values that are less than 0.5 and three α values (0.25, 0.5, and 0.75) are used. **Bold** result indicates the highest score. Note that higher O-Score (O-Score$_{ori}$ or O-Score$_{top}$) is better.

Type	SR-Score$_{ori}$		SR-Score$_{top}$		AR-Score$_{ori}$		AR-Score$_{top}$		R-Score$_{ori}$		R-Score$_{top}$		G-Score		O-Score$_{ori}$		O-Score$_{top}$	
	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop
Small	**0.98**	0.12	**0.98**	0.10	**0.76**	0.15	**0.76**	0.15	**0.87**	0.14	**0.87**	0.12	0.10	**0.80**	**0.68**	0.33	**0.68**	0.32
Medium	**0.99**	0.57	**0.99**	0.17	**0.95**	0.42	**0.93**	0.22	**0.97**	0.50	**0.96**	0.20	0.08	**0.67**	**0.75**	0.57	**0.74**	0.35
Large	**0.86**	0.42	**0.77**	0.05	**0.70**	0.25	**0.66**	0.00	**0.78**	0.34	**0.71**	0.03	0.32	**0.99**	**0.67**	0.50	**0.61**	0.27
Average	**0.99**	0.43	**0.98**	0.14	**0.88**	0.33	**0.86**	0.19	**0.93**	0.38	**0.92**	0.17	0.10	**0.72**	**0.73**	0.49	**0.72**	0.33

Table 5. Average scores of debloated programs generated by Mop and Debop in experiments where a combination of five β values that are no less than 0.5 and three α values (0.25, 0.5, and 0.75) are used. **Bold** result indicates the highest score. Note that higher O-Score (O-Score$_{ori}$ or O-Score$_{top}$) is better.

Type	SR-Score$_{ori}$		SR-Score$_{top}$		AR-Score$_{ori}$		AR-Score$_{top}$		R-Score$_{ori}$		R-Score$_{top}$		G-Score		O-Score$_{ori}$		O-Score$_{top}$	
	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop	Mop	Debop
Small	**0.08**	0.07	**0.06**	0.04	**0.06**	0.05	**0.06**	0.05	**0.07**	0.06	**0.06**	0.05	**0.99**	**0.99**	**0.72**	0.71	**0.71**	**0.71**
Medium	**0.60**	0.55	**0.21**	0.10	**0.37**	0.29	**0.13**	0.03	**0.48**	0.42	**0.17**	0.07	0.95	**0.99**	**0.83**	0.82	**0.74**	0.72
Large	**0.52**	0.42	**0.23**	0.05	**0.36**	0.25	**0.15**	0.00	**0.44**	0.34	**0.19**	0.03	0.98	**0.99**	**0.78**	0.75	**0.69**	0.63
Average	**0.44**	0.40	**0.16**	0.08	**0.27**	0.22	**0.11**	0.03	**0.36**	0.31	**0.14**	0.06	0.97	**0.99**	**0.79**	**0.79**	**0.73**	0.71

When β is equal to or greater than 0.5, as Table 5 shows, Mop can reduce 44% of the statements and 27% of the attack surface, producing a debloated program with a generality score of 0.97. This result shows that, when generality is of interest and weighted more than reduction, Mop can achieve effective debloating by preserving nearly the entire generality of the original program while pruning about half of the code and 27% of the attack surface. For the four large programs, Mop obtains higher size reduction and attack surface reduction scores (0.52 and 0.36) while still making the generality of the debloated programs as high as 0.98, which implies that debloating is quite beneficial for large programs.

With a high β, Mop often uses p_{top} as the initial sample for optimization. For medium and large programs, Mop reduced a significant amount of code while still finding a good tradeoff. For small programs, however, Mop was only able to prune about 8% of the code (3rd row, 2nd column in Table 5). This is because for these programs there are often thousands of inputs in the usage profile and the overlap of the coverage induced from these inputs is significant. The deletion of even a small code fragment (e.g., a statement) can easily result in a significant decrease of program generality. To ensure a high program generality (as specified by a high β), Mop ends up preserving most of the code in the original program and achieving a low reduction.

5.3.2 Result of Debloating with the Must-Handle Inputs. We also evaluated how Mop works with the must-handle inputs. When the must-handle inputs are provided, Mop will make sure to produce a debloated program that behaves correctly for these inputs. Tables 6-8 show the average scores of the debloated programs generated by Mop (and Debop-M) in the debloating experiments where the must-handle inputs are provided and a combination of three α values and β values in different ranges are used: all the nine β values (Table 6), β values less than 0.5 (Table 7), and β values that are no less than 0.5 (Table 8). Similarly, the average scores of Tables 6-8 were also calculated by taking the arithmetic mean of the results from each experimental run, rather than directly averaging the Small-sample, Medium-sample, and Large-sample values.

According to Table 6, Mop produced debloated programs with a 0.48 reduction score (R-Score$_{ori}$) using the original program as the reference. The generality score is 0.88 and the tradeoff O-Score$_{ori}$ is 0.71. Compared to the result of debloating without the must-handle inputs (Table 3), the R-Score$_{ori}$ and O-Score$_{ori}$ (also R-Score$_{top}$ and O-Score$_{top}$) decrease while the G-Score increases. Mop ended up reducing less code while debloating with the must-handle inputs, as it forces to preserve the code covered by the must-handle inputs, and as a result, the set of candidate statements Mop can reduce for optimization becomes smaller. One can see that the R-Score for small programs is close to zero. As we have explained in the previous Section 5.3.1, for these programs, a deletion of a small code fragment (sometimes even a single statement) can lead to a significant decrease of generality. For this reason, Mop did not delete much code, achieving a 0.05 *R-Score*$_{ori}$.

Table 6. Average scores of debloated programs generated by Mop and Debop-M in experiments where the must-handle inputs are provided and a combination of nine β values (between 0.1 and 0.9) and three α values (0.25, 0.5, and 0.75) are used. Small-sample: *totinfo*. Medium-sample: *mkdir-5.2.1*, *date-8.21*, and *tar-1.14*. Large-sample: *gzip-1.3*, *sed-4.1.5*, *vim-5.8*, and *postgresql-12.14*. **Bold** result indicates that the reduced program generated by this tool has a higher score. Note that higher O-Score (O-Score$_{ori}$ or O-Score$_{top}$) is better.

Type	SR-Score$_{ori}$		SR-Score$_{top}$		AR-Score$_{ori}$		AR-Score$_{top}$		R-Score$_{ori}$		R-Score$_{top}$		G-Score		O-Score$_{ori}$		O-Score$_{top}$	
	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M
Small-sample	0.06	0.06	0.02	0.02	0.04	0.04	0.01	0.01	0.05	0.05	0.01	0.01	1.00	1.00	0.53	0.53	0.50	0.50
Medium-sample	**0.78**	0.71	**0.41**	0.24	**0.47**	0.43	**0.25**	0.18	**0.63**	0.57	**0.33**	0.21	0.84	**0.89**	**0.77**	0.76	**0.63**	0.59
Large-sample	**0.54**	0.41	**0.30**	0.08	**0.43**	0.27	**0.26**	0.04	**0.48**	0.34	**0.28**	0.06	0.89	**0.95**	**0.72**	0.65	**0.63**	0.51
Average	**0.57**	0.48	**0.31**	0.13	**0.40**	0.30	**0.22**	0.09	**0.48**	0.39	**0.27**	0.11	0.88	**0.94**	**0.71**	0.68	**0.62**	0.54

Table 7. Average scores of debloated programs generated by Mop and Debop-M in experiments where the must-handle inputs are provided and a combination of four β values that are less than 0.5 and three α values (0.25, 0.5, and 0.75) are used. Small-sample: *totinfo*. Medium-sample: *mkdir-5.2.1*, *date-8.21*, and *tar-1.14*. Large-sample: *gzip-1.3*, *sed-4.1.5*, and *vim-5.8*, and *postgresql-12.14*. **Bold** result indicates that the reduced program generated by this tool has a higher score. Note that higher O-Score (O-Score$_{ori}$ or O-Score$_{top}$) is better.

Type	SR-Score$_{ori}$		SR-Score$_{top}$		AR-Score$_{ori}$		AR-Score$_{top}$		R-Score$_{ori}$		R-Score$_{top}$		G-Score		O-Score$_{ori}$		O-Score$_{top}$	
	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M
Small-sample	0.06	0.06	0.02	0.02	0.04	0.04	0.01	0.01	0.05	0.05	0.01	0.01	1.00	1.00	**0.29**	0.27	0.26	0.26
Medium-sample	**0.83**	0.74	**0.53**	0.30	**0.58**	0.55	**0.39**	0.33	**0.71**	0.65	**0.46**	0.32	0.67	**0.77**	**0.70**	0.68	**0.51**	0.43
Large-sample	**0.60**	0.38	**0.48**	0.09	**0.52**	0.27	**0.45**	0.08	**0.56**	0.33	**0.46**	0.09	0.74	**0.91**	**0.61**	0.47	**0.53**	0.29
Average	**0.62**	0.48	**0.44**	0.16	**0.48**	0.34	**0.37**	0.16	**0.55**	0.41	**0.41**	0.16	0.75	**0.87**	**0.60**	0.53	**0.49**	0.34

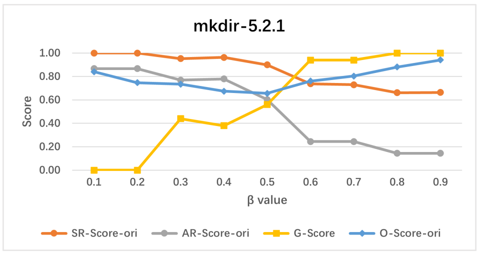
Table 8. Average scores of debloated programs generated by Mop and Debop-M in experiments where the must-handle inputs are provided and a combination of five β values that are no less than 0.5 and three α values (0.25, 0.5, and 0.75) are used. Small-sample: *totinfo*. Medium-sample: *mkdir-5.2.1*, *date-8.21*, and *tar-1.14*. Large-sample: *gzip-1.3*, *sed-4.1.5*, and *vim-5.8*, and *postgresql-12.14*. **Bold** result indicates that the reduced program generated by this tool has a higher score. Note that higher O-Score (O-Score$_{ori}$ or O-Score$_{top}$) is better.

Type	SR-Score$_{ori}$		SR-Score$_{top}$		AR-Score$_{ori}$		AR-Score$_{top}$		R-Score$_{ori}$		R-Score$_{top}$		G-Score		O-Score$_{ori}$		O-Score$_{top}$	
	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M	Mop	Debop-M
Small-sample	0.06	0.06	0.02	0.02	0.04	0.04	0.01	0.01	0.05	0.05	0.01	0.01	1.00	1.00	0.64	0.64	0.63	0.63
Medium-sample	**0.76**	0.70	**0.35**	0.21	**0.42**	0.37	**0.18**	0.10	**0.59**	0.54	**0.27**	0.16	0.93	**0.95**	**0.81**	0.81	**0.70**	0.67
Large-sample	**0.51**	0.43	**0.21**	0.08	**0.38**	0.27	**0.16**	0.02	**0.44**	0.35	**0.19**	0.05	0.96	**0.98**	**0.77**	0.74	**0.67**	0.63
Average	**0.55**	0.48	**0.24**	0.12	**0.35**	0.28	**0.15**	0.05	**0.45**	0.38	**0.20**	0.09	0.95	**0.97**	**0.77**	0.76	**0.68**	0.65

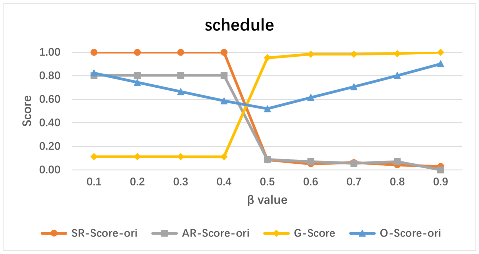
When β is less than 0.5, Mop achieves a 0.55 R-Score$_{ori}$ and a 0.75 G-Score. The R-Score$_{ori}$ is lower than that (0.93) in Table 4 (when no must-handle inputs are used for debloating) and the G-Score is significantly higher. This shows that the use of must-handle inputs can lead to the generation of debloated programs with reasonably high generality even when β is low. We note that this is good, as a debloated program with extremely low generality is often of little usefulness. The use of must-handle inputs forces the preservation of code exercised by these inputs such that debloated programs with extremely low generality are avoided. When β is not smaller than 0.5, the G-Score and O-Score$_{ori}$ in Table 8 are overall similar to those shown in Table 5 as Mop tends to generate debloated programs with a high generality regardless of whether must-handle inputs are provided or not.

5.3.3 Variance of Result for Debloating with Different Weights. In the previous sub-sections, we presented and analyzed the average scores that Mop achieved in the debloating experiments. This sub-section investigates the variance of the scores as the α and β values change.

Result variance when β values change. The β value controls the preference between reduction and generality. Overall we found that as the β value increases, that is, the generality weight gets higher, Mop can reduce less code and generate debloated programs with higher generality scores.



(a) Variance of the scores for *mkdir-5.2.1*.



(b) Variance of the scores for *schedule*.

Fig. 6. The variance of scores with different β values used for debloating (α = 0.5).

For different programs, the variance is also different. Figure 6 presents two types of variance of scores (curves) for the debloated programs that Mop can generate. Interested readers can refer to [14] for all the curves. The left curve in Figure 6, which is the result for *mkdir-5.2.1* (with no must-handle inputs used), is an example showing that the scores can gradually change as the β values increase. As one can see, as the β value gets higher, Mop is able to reduce less code (the orange and grey lines drop) to generate debloated programs whose generality scores get higher (the yellow line goes up).

The right part of Figure 6, which presents the curve for *schedule*, is however less ideal, as it shows that there are cases where the debloated programs' scores do not change (e.g., when the β value increases from 0.2 to 0.3) or they change abruptly (e.g., when β increases from 0.4 to 0.5).

For this program, Mop's exploration is less effective. The program has more than 2650 test inputs associated with it. In the first stage, for every sample that Mop generated and tested in the MCMC search, the generality variance is only tiny (1/2650), while the reduction loss can be more significant (especially when β is small and Mop adds back a large amount of code exercised by an input to the base program), which can cause an easy sample rejection. In the 2nd and 3rd stages, Mop can change a sample's generality in a more significant way, as its mutation can affect the program behavior related to more than one input. However, for small programs like *schedule*, as we have previously explained, Mop's mutation can easily result in a "broken" program that fails to process many inputs, due to the high coverage overlap of the inputs and the lack of key statements. For larger programs, the first stage may still suffer from the generality-minimal-change issue, and the last two stages may not be highly efficient due to their needs of frequent program execution for sample validation. In future work, we will investigate improving the efficiency of Mop and using a more flexible and balanced comparison of the reduction and generality variances to decide the sample acceptance for more effective stochastic optimization.

Result variance when α values change. Unlike β, which controls the preference for two competing factors (reduction and generality), α is used to adjust the weight for two correlated measures, size reduction and attack surface reduction. Due to the correlated property of these measures, which means an increase of size reduction often also implies a larger attack surface reduction, and vice versa, the way that α affects the debloating effectiveness of Mop is empirically insignificant. As a result, we do not identify a variance pattern of debloating scores as the α value changes. We believe that the actual variance can be largely subject to noise.

Mop can prune 68% code and 54% attack surface to produce a debloated program preserving 58% generality of the original program, achieving a tradeoff score 0.76. When the must-handle inputs are used to specify the must-preserve features for debloating, Mop can avoid generating programs with low generality. Moreover, with different β weights, Mop can produce debloated programs with different reduction-generality tradeoffs. The influence of α on debloating is insigificant.

5.4 Results for RQ2: How does Mop compare with Debop in terms of the reduction, generality, and tradeoffs achieved?

5.4.1 Comparison of Mop and Debop without the Must-Handle Inputs Used for Debloating. According to Tables 3-5, Mop produced debloated programs with higher objective scores (O-Score$_{ori}$ and O-Score$_{top}$) indicating better tradeoffs than Debop. Mop's O-Score$_{ori}$ is 15.2% higher than Debop's O-Score$_{ori}$. Recall that O-Score$_{ori}$ is computed based on code reduction measured against the original program (i.e., R-Score$_{ori}$). Because the optimization of Mop and Debop only involves mutating code in p_{top}, the code reduction measured against p_{top} (i.e., R-Score$_{top}$) can directly reflect how much size or attack surface is reduced by the optimization process. For this reason, O-Score$_{top}$, which is computed based on R-Score$_{top}$, can reflect the core optimization abilities of Mop and Debop. Mop's O-Score$_{top}$ is 30.9% higher than Debop's, showing that Mop has a stronger optimization ability. The superiority of Mop for optimization is attributed to its strong reduction ability. According to Table 3, Mop's R-Score$_{ori}$ (0.61) is 79.4% higher than Debop's, and its R-Score$_{top}$ (0.48) is about four times as high as Debop's R-Score$_{top}$.

When β is small (less than 0.5), which means reduction is of better interest, Mop achieved a 0.92 R-Score$_{top}$ while Debop's score is as low as 0.17. The debloated programs generated by Mop have a lower generality, which is understandable, as Mop reduced much more code than Debop, and a program with less code preserves less functionality. Overall, however, because reduction is weighted more than generality, Mop achieved a much better tradeoff than Debop: Mop's O-Score$_{ori}$ is 49.0% higher than Debop's and its O-Score$_{top}$ is 118.2% higher.

When $\beta \geq 0.5$, the optimization objective emphasizes generality (i.e., preserves more features). Under this configuration, aggressive code reduction is constrained, as more features must be retained to obtain a higher G-Score. As a result, Mop is less able to perform extensive code reduction, leading to O-Scores that are closer to those of Debop. Nevertheless, Mop still outperforms Debop, especially for large programs based on the top metric (which as we explained reflects a technique's core optimization ability). For large programs, in Table 5, Mop achieves a nearly 20% reduction score (R-Score$_{top}$), whereas Debop 's reduction score is only 3%, with similar generality scores (0.98 vs. 0.99).

The heat maps shown in Figures 7 and 8 depict in heated colors the differences between Mop's and Debop's tradeoff scores achieved in the debloating experiments with different α and β values used and for different programs. Specifically, Figure 7 shows the differences of O-Score$_{ori}$ (computed based on R-Score$_{ori}$ and G-Score) and Figure 8 shows the differences of O-Score$_{top}$ (computed based on R-Score$_{top}$ and G-Score). Each grid of the heat map displays a color quantifying $o_{mop} - o_{debop}$, where o_{mop} and o_{debop} are the objective scores achieved by the two techniques. A green grid indicates a positive difference showing that Mop's score is higher than Debop's, a red grid indicates the opposite, that is, Mop's score is lower than Debop's, and a yellow grid indicates no difference.

As one can see in Figure 7, many of the grids have green colors, indicating that Mop found a better tradeoff than Debop in many experiments. In fact, Mop achieved an objective score that is no lower than Debop in 86.54% of the experiments (grids). As we previously explained, O-Score$_{top}$ can reflect a technique's core optimization ability. One can clearly tell based on the result of Figure 8 that Mop's optimization ability is stronger than Debop's. In 96.19% of the experiments, Mop achieved a higher objective score, indicating a better tradeoff.

Mop's objective score is not always higher than Debop's, especially when β is large and the program is small. This is because, as we explained in Section 5.3.1, in these cases Mop tends to produce debloated programs with high generality, and Debop almost always produces a high-generality debloated program because its reduction ability is weak. The programs produced by Mop and Debop that both have high generality often have similar scores.

Compared to Figure 7, Figure 8, which shows the difference of tradeoff scores computed with p_{top} used as the reference program, has fewer red grids. As we explained, scores computed based on p_{top} can reflect the core optimization ability of Mop and Debop. The fact that Figure 8 has fewer red grids implies that Mop's optimization ability is stronger than Debop in many more cases.

5.4.2 Comparison of Mop and Debop-M with the Must-Handle Inputs Used for Debloating. Note that the original Debop approach does not assume any must-handle inputs, and the preceding results were from debloating experiments where no must-handle inputs are provided. To further investigate how Mop compares to Debop's approach in the scenario where the must-handle inputs are used, we designed Debop-M and used it for debloating. Tables 6-8 show that Mop has higher O-Score$_{ori}$ and O-Score$_{top}$ scores than Debop-M, indicating that Mop is still better than Debop-M even when performing debloating with the must-handle inputs. According to Table 6, Mop's O-Score$_{ori}$ and O-Score$_{top}$ for all programs are 4.4% and 14.8% higher than Debop-M's.

In Table 6, all evaluations were conducted with must-handle inputs, which designate certain features as non-removable. This introduces strict constraints on code reduction. These constraints particularly restrict the search space in coarse-grained stages, diminishing Mop's strength in aggressive debloating. For the Small-sample program, where must-handle inputs exercised 89%

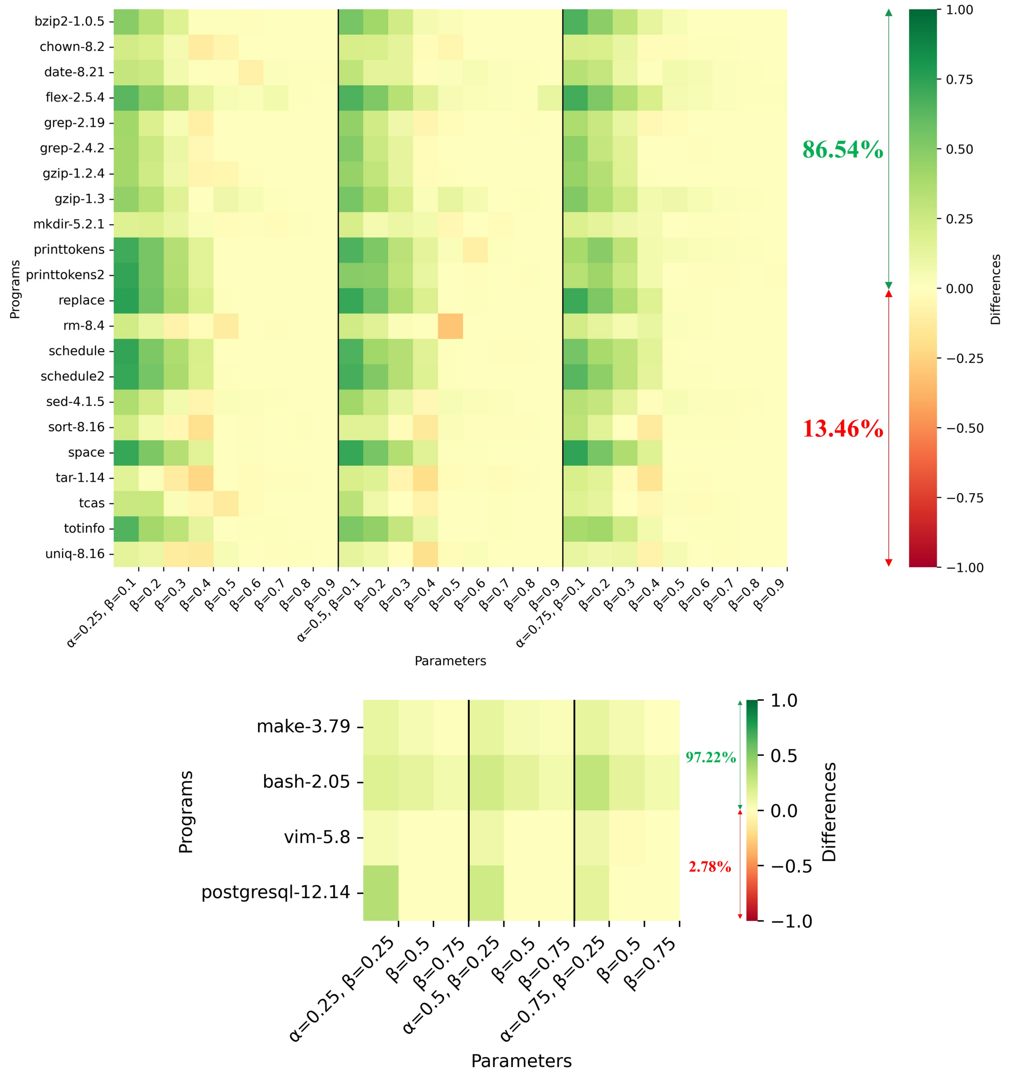
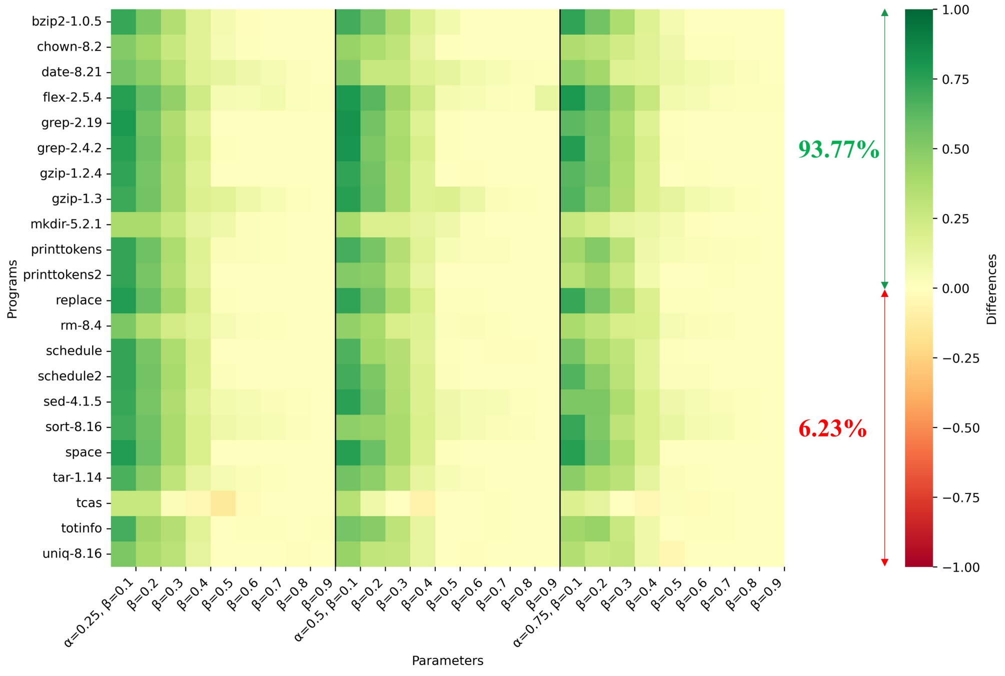


Fig. 7. The O-Score$_{ori}$ of debloated programs generated by Mop minus the O-Score$_{ori}$ of debloated programs generated by Debop. Green: Mop is better than Debop in terms of the O-Score$_{ori}$; Red: Mop is worse than Debop in terms of the O-Score$_{ori}$; Yellow: Mop and Debop have the same O-Score$_{ori}$ values.

of the statements, the search space for both Mop and Debop-M is severely constrained, resulting in comparable debloating scores. However, for Large-sample programs, Mop still significantly outperforms Debop-M, achieving a higher O-Score$_{top}$ (0.63 vs. 0.51), demonstrating its superior optimization capability in large programs.

For Medium-sample and Large-sample programs, Mop's O-Score$_{ori}$ and O-Score$_{top}$ are 5.7% and 16.7% higher than Debop-M's scores. This suggests that for complex programs, Mop is much more effective than Debop in the debloating scenarios where the must-handle inputs are provided. Using the simple mutation, Debop-M is very weak at dealing with complex programs whose search space



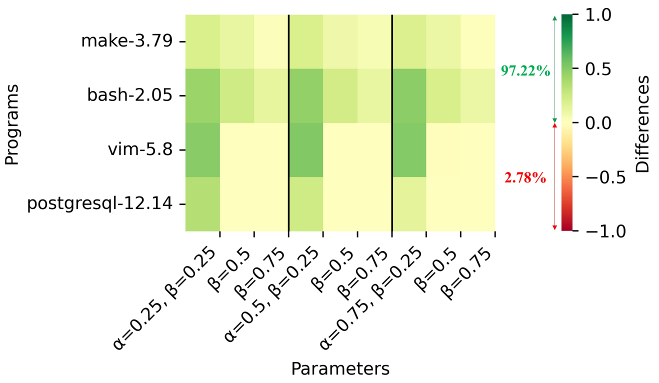


Fig. 8. The O-Score$_{top}$ of debloated programs generated by Mop minus the O-Score$_{top}$ of debloated programs generated by Debop. Green: Mop is better than Debop in terms of the O-Score$_{top}$; Red: Mop is worse than Debop in terms of the O-Score$_{top}$; Yellow: Mop and Debop have the same O-Score$_{top}$ values.

for debloating is huge. We found that, for Medium-sample and Large-sample programs, about 80.9% of the reduction that Debop-M achieved while debloating was contributed by the preprocessing step but not by its stochastic optimization.

5.4.3 Statistical Testing of O-Score Differences Between Mop and Debop (and Debop-M). Table 9 presents the statistical test results for the O-Scores, categorized by β values, based on the 26 programs without must-handle inputs and the 8 programs with must-handle inputs. We used two statistical metrics: the p-value and the effect size r. The p-value is related to statistical significance, quantifying how likely it is to obtain data as extreme or more extreme than the one obtained, if the null hypothesis were true. A p-value smaller than 0.05 indicates a significant improvement of Mop over Debop. The effect size r quantifies the standardized magnitude of the difference between Mop and Debop (Debop-M). The larger the r value, the stronger the improvement of Mop over Debop. $r < 0.3$ typically indicates a small effect; $0.3 \leq r < 0.5$ shows a medium effect; and $r \geq 0.5$ reflects a large effect [23].

Table 9. Statistical testing results comparing Mop's and Debop's (Debop-M's) O-Scores without and with the must-handle inputs used. **Bold p-values** indicate that Mop significantly outperforms Debop (and Debop-M) ($p < 0.05$), and **bold r values** indicate large effects ($r > 0.5$).

Experiment	O-Score	β	p-value	Is significant?	Effect size r
Without must-handle inputs	O-Score$_{top}$	All β	$\mathbf{1.49 \times 10^{-8}}$	Yes	**0.87**
		$\beta < 0.5$	$\mathbf{1.49 \times 10^{-8}}$	Yes	**0.87**
		$\beta \geq 0.5$	$\mathbf{7.65 \times 10^{-5}}$	Yes	**0.77**
	O-Score$_{ori}$	All β	$\mathbf{2.98 \times 10^{-8}}$	Yes	**0.86**
		$\beta < 0.5$	$\mathbf{1.49 \times 10^{-8}}$	Yes	**0.87**
		$\beta \geq 0.5$	0.06	No	0.32
With must-handle inputs	O-Score$_{top}$	All β	**0.01**	Yes	**0.90**
		$\beta < 0.5$	**0.01**	Yes	**0.90**
		$\beta \geq 0.5$	**0.02**	Yes	**0.77**
	O-Score$_{ori}$	All β	**0.03**	Yes	**0.70**
		$\beta < 0.5$	**0.03**	Yes	**0.70**
		$\beta \geq 0.5$	0.16	No	0.41

Our results show that Mop significantly outperforms Debop (and Debop-M) across most configurations ($p < 0.05$). In cases where significance is confirmed, the p-values are all not over 0.03 and in many cases smaller than 0.01, indicating strong evidence of Mop's improvement over Debop (Debop-M). The only cases where the differences are not statistically significant correspond to the O-Score$_{ori}$ results under $\beta \geq 0.5$. When β is not less than 0.5, implying that program generality is of higher weight, both Mop and Debop (and Debop-M) tend to preserve code. This means even if Debop (Debop-M) has a weaker ability in reducing code for optimization, it may still end up finding a close-to-optimal solution, which preserves most of the code, as Mop does. Moreover, computing the reduction score R-Score$_ori$ and the trade-off O-Score$_ori$ based on the original program (rather than the top program) captures the effect of the pre-processing step (which simply removes code not exercised by any given input) and thus weakens the effect of Mop 's reduction by optimization. The weakened reduction effect also contributes in part to the non-significant difference between Mop and Debop (and Debop-M). Nevertheless, as mentioned in Section 5.3.1 (the paragraph starting with "The use of p_{top}"), O-Score$_{top}$ (computed based on R-Score$_{top}$) can better reflect the optimization abilities of techniques.

In cases where Mop's improvement is significant, the r values are all over 0.5 and often greater than 0.7. Even in the two non-significant cases, the corresponding r values are still over 0.3 (effect size is medium). These effect size results suggest that Mop's superiority over Debop is often substantial.

Taken together, our statistical testing results support the conclusion that Mop significantly outperforms Debop (and Debop-M) in terms of the tradeoff scores and hence the debloating effectiveness.

5.4.4 Runtime Performance Analysis of Mop and Debop. We conducted experiments with varying time budgets to evaluate the impact of time constraints on the debloating results and convergence and to assess the runtime performance differences between Mop and Debop.

Table 10 reports the O-Score$_{ori}$ and O-Score$_{top}$ achieved by Mop and Debop under a combination of three α values and night β values and different time budgets ($0.5T$, T, and $1.5T$), where T is the baseline time budget (1 hour for Small and Medium programs and 12 hours for Large programs). Mop outperforms Debop across all time budgets in all settings, with its debloated programs achieving, on average, 15.8% higher O-Score$_{ori}$ and 28.4% higher O-Score$_{top}$ than Debop's.

Table 10. Average O-Scores of debloated programs generated by Mop and Debop under varying time budgets ($0.5T$, T, and $1.5T$), without must-handle inputs, using a combination of nine β values (between 0.1 and 0.9) and three α values (0.25, 0.5, and 0.75). Small-sample: *totinfo*. Medium-sample: *mkdir-5.2.1*, *date-8.21*, and *tar-1.14*. Large-sample: *gzip-1.3*, *sed-4.1.5*, *vim-5.8*, and *postgresql-12.14*. **Bold** result indicates a higher score. Note that higher O-Score (O-Score$_{ori}$ or O-Score$_{top}$) is better.

Type	Time Budget	O-Score$_{ori}$		O-Score$_{top}$	
		Mop	Debop	Mop	Debop
Small-sample	$0.5T$	**0.69**	0.54	**0.67**	0.51
	T	**0.70**	0.56	**0.68**	0.54
	$1.5T$	**0.70**	0.59	**0.68**	0.57
Medium-sample	$0.5T$	**0.80**	0.75	**0.73**	0.56
	T	**0.81**	0.77	**0.74**	0.60
	$1.5T$	**0.81**	0.77	**0.74**	0.60
Large-sample	$0.5T$	**0.78**	0.67	**0.68**	0.52
	T	**0.79**	0.67	**0.70**	0.52
	$1.5T$	**0.80**	0.67	**0.71**	0.52

For Small-sample programs, Mop achieves an O-Score$_{ori}$ of 0.69 and O-Score$_{top}$ of 0.67 at $0.5T$, significantly outperforming Debop's 0.54 and 0.51. As the time budget increases to T and $1.5T$, Mop's O-Score$_{ori}$ and O-Score$_{top}$ improve slightly to 0.70 and 0.68, which suggests that it can find the debloated program with a good trade-off early on. For Medium-sample programs, Mop achieves an O-Score$_{ori}$ of 0.80 and O-Score$_{top}$ of 0.73 at $0.5T$, exceeding Debop's 0.75 and 0.56. The scores of Mop improve slightly to 0.81 and 0.74 at T and $1.5T$, suggesting that Mop is already highly effective with limited time (i.e., $0.5T$). For Large-sample programs, Mop starts at 0.78 (O-Score$_{ori}$) and 0.68 (O-Score$_{top}$) at $0.5T$, surpassing Debop's 0.67 and 0.52. By the $1.5T$ time budget, Mop improves the O-Score$_{ori}$ to 0.80 and O-Score$_{top}$ to 0.71. This result indicates that Mop can quickly generate well-balanced debloated programs at $0.5T$ for Large-sample programs, and further achieve better trade-offs under longer time budgets (e.g., $1.5T$).

The above results highlight that Mop produces effective outcomes faster than Debop across all program scales. For Small-sample and Medium-sample programs, Mop's performance at T is almost the same as at $1.5T$, showing that most of the gains are achieved at $0.5T$ with only slight improvements thereafter. This underscores Mop's ability to quickly find the debloated programs with good tradeoffs. For Large-sample programs, Mop can quickly find the debloated programs with good tradeoffs, and it can still be improved over a longer time. Although Mop continues to make further refinements on Large-sample programs with additional time, it has already generated debloated programs with significantly better tradeoffs than Debop under a shorter time budget, implying its practicality in real-world applications. In contrast, Debop explores the search space more slowly. For Small-sample and Medium-sample programs, the results of Debop at $1.5T$ are noticeably better than at $0.5T$. For Large-sample programs, Debop shows no differences across different time budgets, as its limited exploration capability makes it unable to debloat Large-sample programs.

Mop achieved a better tradeoff score than Debop in up to 93% of the debloating experiments. On average, Mop's tradeoff score can be about 30% higher than Debop's score, which implies that Mop has a stronger optimization ability for debloating. The statistical testing result shows that Mop's superiority over Debop is significant in most cases. Using the must-handle inputs for debloating, Mop can still achieve a better tradeoff score than Debop. However, due to the smaller search space induced by the use of must-handle inputs, the difference of the tradeoff scores achieved by the two techniques is less significant.

5.5 Results for RQ3: How does Mop compare with Chisel and Razor in terms of size and attack-surface reductions?

We also compared Mop with Chisel and Razor to understand Mop's reduction ability. Chisel and Razor are reduction-oriented techniques. To have a fair comparison, as we noted in Section 5.2.3, we followed the practice of [94] and applied Chisel and Razor to a program for debloating based on not all the given inputs in the usage profile but the inputs that the debloated programs generated by Mop can correctly handle in the experiments (without the must-handle inputs). In this way, we made sure that the debloated programs generated by all techniques behave correctly for the same inputs (having the same generality), and we focused on comparing the code reduction achieved by all techniques.

Table 11 presents the average R-Score$_{ori}$ (code reduction calculated based on the original program) scores achieved in debloating experiments where a combination of three α values and different ranges of β values are considered: $\beta < 0.5$, $\beta \geq 0.5$, and all β values (between 0.1 and 0.9). The left part shows the scores for Mop and Chisel. The right part compares Mop-bin (a variant of Mop) and Razor in four versions (Razor-zCode, Razor-zCall, Razor-zLib, and Razor-zFunc) using different heuristics for code augmentation. Since Razor, unlike Mop, is a binary-based technique, we created Mop-bin only for a comparison with Razor. Mop-bin compiles the debloated program generated by Mop with the -g option (as Razor requires a binary containing the debugging information for debloating). We computed the binary-based size reduction (SR-Score) for Mop-bin and the Razor-based tools. The size reduction scores for Mop and Chisel are source-based. Table 12 shows the SR-Score$_{ori}$ (size reduction calculated based on the original program) and AR-Score$_{ori}$ (attack surface reduction calculated based on the original program) scores achieved by different techniques.

Mop outperforms Chisel in achieving higher size reduction and attack-surface reduction scores (Table 12) and higher reduction scores for all β values (Table 11). Overall Mop's R-Score$_{ori}$ is about 16.7% higher than Chisel's score. This result shows that Mop has a stronger reduction ability than Chisel. We found that Chisel, which performs delta-debugging-based debloating, is empirically inefficient. Chisel relies on repeated program execution against the given inputs to validate each debloated program it generates. Because a program can have hundreds to thousands of inputs associated with it (e.g., the *space* program has more than 13K inputs), program validation is expensive. Mop also needs repeated program execution for optimization assessment. However, the expensive repeated evaluation of debloated programs is only performed for two stages. Its first stage can quickly identify a debloated program based on code coverage for the subsequent less aggressive optimization.

The right part of Table 11 shows that the average R-Score$_{ori}$ of Mop-bin is 0.60. The score is higher than that of Razor-zCall (0.53), Razor-zLib (0.36), and Razor-zFunc (0.35), and is only slightly lower than Razor-zCode (0.62). This implies that Mop's reduction ability is at least comparable to

Table 11. The R-Score$_{ori}$ of the debloated programs generated by Mop, Mop-bin, Chisel, and the four versions of Razor (i.e., Razor-zCode, Razor-zCall, Razor-zLib, and Razor-zFunc) using different heuristics for code augmentation. Mop-bin is a variant of Mop. It compiles the debloated program generated Mop with the -g option (the same option we used to obtain an initial binary for Razor debloating) and calculates a binary-based size reduction and attack-surface reduction. We used Mop-bin instead of Mop to compare with Razor, as Razor is a binary-based debloating technique. Note that the size reduction computed for Mop and Chisel is source-based. It is computed based on the fraction of the statements removed. **Bold** result indicates the highest score.

β	Comparison between Mop and Chisel		Comparison between Mop and the four versions of Razor				
	Mop*	Chisel	Mop-bin	Razor-zCode	Razor-zCall	Razor-zLib	Razor-zFunc
$\beta < 0.5$	**0.94**	0.80	**0.85**	0.73	0.64	0.45	0.43
$\beta \geq 0.5$	**0.38**	0.34	0.40	**0.54**	0.45	0.29	0.28
All β	**0.63**	0.54	0.60	**0.62**	0.53	0.36	0.35

* Note that the result of Mop is different from that in Table 3. This is because in this experiment we excluded some programs that Chisel and Razor fail to handle, as explained in Section 5.2.4.

Table 12. The SR-Score$_{ori}$ and AR-Score$_{ori}$ scores of the debloated programs generated by Mop, Mop-bin, Chisel, and the four versions of Razor using different heuristics for code augmentation. The SR-Score$_{ori}$ scores for Mop and Chisel are computed based on the number of statements reduced in the source files. The SR-Score$_{ori}$ scores for Mop-bin and the four Razor tools are computed based on the number of bytes reduced in the binaries. **Bold** result indicates the highest score.

Techniques	Average SR-Score$_{ori}$	Average AR-Score$_{ori}$
Mop	**0.67**	**0.59**
Chisel	0.55	0.49
Mop-bin	0.58	**0.62**
Razor-zCode	**0.66**	0.56
Razor-zCall	0.62	0.42
Razor-zLib	0.48	0.22
Razor-zFunc	0.47	0.20

Razor's. Mop is actually more effective when β is smaller than 0.5 (i.e., reduction is weighted more than generality) — its R-Score$_{ori}$ is 0.85 whereas the scores of all Razor-based techniques are no higher than 0.73.

From left to right in the right part of Table 11, one can see that the reduction scores of Razor techniques get lower and lower. This is because these techniques use an increasing level of aggressiveness for code augmentation. Code augmentation is performed with different aggressiveness to retain different amounts of related code for generality enhancement. Razor-zCode does the minimal augmentation by only retaining the related edges of CFG, and thus achieves the highest reduction score. In contrast, Razor-zFunc does the most aggressive augmentation by retaining non-executed external functions, and thus has the lowest score.

Table 12 shows that Razor-zCode and Razor-zCall have higher size reduction scores than Mop-bin while Mop-bin has the highest attack-surface reduction score. The reason why some Razor-based tools achieve better size reduction than Mop-bin could be because that Razor, which targets code instructions (rather than statements) for debloating, has more code-removal opportunities (e.g., it may remove components of a statement instead of the whole statement as Mop does). The reason why Mop can achieve higher attack surface reduction than Razor is perhaps because that

Mop, by reducing single segments or statements for optimization in its last two stages, can avoid fully preserving the original complete execution paths derived from profiling, which is helpful for attack surface reduction.

Even as an optimization-oriented technique, Mop achieves a reduction score that is 16.7% higher than Chisel. Mop can do better attack surface reduction than both Chisel and Razor. Its size reduction ability is at least comparable to Razor's.

5.6 Results for RQ4: How effective is each of the Mop's optimization stages?

To answer RQ4, we (1) analyzed the contribution of each of Mop's optimization stages in terms of the tradeoff improvement (measured by the increase of the O-Score value) and (2) did an ablation experiment to compare Mop with its variants, each performing only two stages for optimization.

Table 13. The average increase (in fraction) of O-Score$_{ori}$ and O-Score$_{top}$ contributed by each of the Mop's optimization stages in the debloating experiments where different α and β values are used. Mop-S_1, Mop-S_2, and Mop-S_3 are Mop's first, second, and third optimization stages respectively.

β	Increase of O-Score$_{ori}$			Increase of O-Score$_{top}$		
	Mop-S$_1$	Mop-S$_2$	Mop-S$_3$	Mop-S$_1$	Mop-S$_2$	Mop-S$_3$
$\beta < 0.5$	60.71%	20.80%	18.49%	42.30%	33.10%	24.59%
$\beta \geq 0.5$	59.33%	36.59%	4.08%	47.84%	48.32%	3.84%
All β	60.35%	24.97%	14.68%	44.55%	39.29%	16.16%

5.6.1 Contribution Analysis of the Mop's Optimization Stages. Table 13 presents the increase (in fraction) of O-Score (either O-Score$_{ori}$ or O-Score$_{top}$) contributed by each stage. As we explained in Section 5.2.4 (in Component analysis of Mop), the contribution of stage-k (or sk) is computed as $(o_{sk\_end} - o_{sk\_start})/(o_{end} - o_{start})$, where o_{sk_start} and o_{sk_end} are the O-Score values achieved at the start and end of the stage-k and o_{start} and o_{end} are O-Score values before and after the whole optimization process. To ensure a valid fraction, while making Table 13, we excluded the experiment results where the optimization process does not improve the O-Score (i.e., $o_{end} - o_{start}$ is 0).

Our results show that the first two stages make significant contributions in terms of increasing the O-Score values. The increase of O-Score$_{ori}$ caused by Mop-S_1 is 60.35% and about 2.5 times of that of O-Score$_{ori}$ by Mop-S_2. When the top program is used as the reference for reduction calculation and as a result the part of reduction caused by the preprocessing step is canceled, the increase by Mop-S_1 (44.55%) gets closer to that by Mop-S_2 (39.29%).

Comparatively, the increase of O-Score caused by Mop-S_3 is much less significant and is no more than 17%. The slow increase is a reflection of the nature of the third stage, which uses a single-statement-based mutation model and aims for fine-grained exploration. One can see that, when β is less than 0.5, Mop-S_3's result is much better than what it is when β is no less than 0.5. Upon analyzing the experiment logs, we found that when β is less than 0.5, by adding back single statements for mutation (which often occurs), Mop has a good chance of generating good samples, which it accepts for further exploration, by achieving a significant improvement of generality (possibly due to the add-back of a key statement or in the presence of weak oracle for output comparison) or attack surface reduction. Take *space* for example, by adding back the string copy statement `strcpy((char * __restrict)(ErrorMessages[80]), (char const * __restrict )"** ERROR 80: Incorrect PHASE excitation definition.");`, which possibly causes the corruption of several attack chains, Mop obtained a 17.8% increase of attack surface reduction and chose to accept the sample for further optimization. When β is no smaller than 0.5, by reducing a single statement at a time to proceed, Mop tends to generate samples that are funtionality-broken to reject. This explains why the increase of O-Score is less significant.

5.6.2 Result of the Ablation Experiments. We did an ablation experiment by creating three variants of Mop: Mop-$S_{1,2}$, Mop-$S_{1,3}$, and Mop-$S_{2,3}$, each performing optimization with one stage left out, and then we applied these techniques to the same programs for debloating.

Our results show that the average O-Score values (either O-Score$_{ori}$ or the O-Score$_{top}$) achieved by Mop and its variants are very similar. The difference is often less than 0.1. This result is a bit surprising, as it shows that Mop's three optimization stages are equally ineffective. After delving into the log details, we realized that the selection of the initial sample (the starting point for optimization) affects our investigation of the effectiveness of Mop's stages. A good initial sample is somewhere close to the optimal result, leaving not much room for the subsequent optimization to improve upon it. In this way, a good initial sample can make the comparison between Mop and its variants less significant for an effective component analysis, as there can be little optimization opportunity for these techniques. Although we cannot directly assess the quality of the initial sample by quantifying how close it is to the optimal result (because we do not know the optimal result), we think it is possible that in many cases Mop starts with a good sample for debloating – in 42.1% of the experiments, Mop's optimization did not find a debloated program whose O-Score is higher than the initial sample.

In order to maximize the optimization opportunity for the Mop's three stages as a means to gain a more in-depth investigation of their effectiveness, we did an additional experiment. In this experiment, as we explained in Section 5.2.4, we created two extreme settings: Setting-I and Setting-II for debloating. In Setting-I, we set β as 0.1, the smallest β value used in the previous experiments, to focus on code reduction. However, instead of using p_{base} as the initial sample, as Mop did in the previous experiments, Mop is forced to start with p_{top} for a code-reduction-intensive optimization. Likewise, in Setting-II, we set β as 0.9, the largest β value, to focus on generality preservation. And here Mop is configured to start with p_{base} for a code-recovery-intensive optimization. In both settings, we set α as 0.5, given that the influence of debloating with difference α values is minimal (Section 5.3.3).

Table 14 presents the debloating result, showing that the O-Score values of Mop-$S_{1,3}$ and Mop-$S_{2,3}$ are lower than those of Mop This result implies that, without performing the first- and second-stage optimization, Mop can achieve worse O-Score values, demonstrating the effectiveness and usefulness of the two stages.

Table 14. The average O-Score$_{ori}$ and O-Score$_{top}$ of the debloated programs generated by Mop, Mop-$S_{1,2}$, Mop-$S_{1,3}$, and Mop-$S_{2,3}$ in two extreme settings for debloating where β is 0.1, α is 0.5, and the top program is the initial sample for optimization (Setting-I) and β is 0.9, α is 0.5, and the base program is the initial sample (Setting-II). **Bold** result indicates the highest O-Score.

Settings	**Average O-Score$_{ori}$**				**Average O-Score$_{top}$**			
	Mop	**Mop-S$_{1,2}$**	**Mop-S$_{1,3}$**	**Mop-S$_{2,3}$**	**Mop**	**Mop-S$_{1,2}$**	**Mop-S$_{1,3}$**	**Mop-S$_{2,3}$**
Setting-I	0.61	**0.63**	0.55	0.62	0.47	**0.51**	0.32	0.49
Setting-II	**0.93**	**0.93**	**0.93**	0.20	**0.89**	**0.89**	**0.89**	0.20
Average	0.77	**0.78**	0.72	0.41	0.68	**0.70**	0.60	0.35

In particular, the first stage plays a key role in Setting-II where the debloating task involves an intensive code add-back to the base program. By performing optimization without the first stage, Mop-$S_{2,3}$ achieved much lower O-Score values (0.2 for both O-Score$_{ori}$ and O-Score$_{top}$) than Mop (0.93 for O-Score$_{ori}$ and 0.89 for O-Score$_{top}$). The second stage is more important for Setting-I where intensive code reduction is performed. In this scenario, Mop's first stage is less effective, as the optimization starts with the top program, and due to the significant overlap of coverages induced from the execution with different inputs, a coverage-based model may not easily achieve a significant reduction by choosing a coverage and removing part of it that has no overlap with others. Comparatively, the second stage is less important for add-back-intensive debloating, as the model that adds back a single segment to a base program (or something close to it) can easily generate samples with no generality improvement and thus be rejected.

According to Table 14, the score of Mop-$S_{1,2}$ is close to and even slightly higher than that of Mop. This result implies that the third stage of Mop is not as effective as the other two. We nevertheless believe that there are scenarios where a fine-grained exploration provided by the third stage can provide unique strengths for debloating and provide an example in Section 5.6.1. We leave to future work a more extensive experiment to investigate the unique effectiveness of the third stage. Finally, it is worth noting that Mop supports adjustable exploration budgets for the three stages, and a user can reduce the budget of the third stage (or even cut it off) for debloating, if needed.

Through a contribution analysis and the ablation experiments, we found that the first two stages of Mop contribute significantly for optimization-based debloating. The third stage of Mop is less effective but has unique strengths.

5.7 Results for RQ5: How effective is Mop's post-processing step in improving the robustness of the debloated program?

To investigate Mop's effectiveness at enhancing program robustness, we applied Mop's post-processing method to the 8 debloated programs derived from an experiment where α and β are both set as 0.5 for debloating (see Section 5.2.4, the "Robustness Improvement" part). The results are presented in Table 15. It shows that only three programs (*date-8.21*, *sed-4.1.5*, and *tar-1.14*) exhibited crash or hang behaviors when tested with the fuzzed inputs. After the enhancement, an average of 2.4% additional statements (relative to the debloated program size) were reintroduced into the programs. This enhancement reduced the crashes and hangs triggered by 70% of the testing inputs, demonstrating significant robustness improvement for the debloated programs.

The absence of crash and hang behaviors for the remaining five programs (*totinfo*, *gzip-1.3*, *mkdir-5.2.1*, *vim-5.8*, and *postgresql-12.14*) is attributed to the high generality (G-Score) of their three-stage optimized debloated programs. The high G-Score indicates that Mop preserved sufficient robustness maintaining code (e.g., error-handling code) during optimization, as ensuring generality implicitly maintains code critical for handling edge cases. Consequently, these programs exhibited robust behavior even before the explicit robustness enhancement step.

The reintroduced code in the robustness enhancement phase primarily consists of error-handling statements. For instance, in *sed-4.1.5*, the added code includes `bad_prog((((errors + sizeof ("multiple ‘!ś"))+sizeof("unexpected ‘,̋")) + sizeof("invalid usage of +N or N as first address")) + sizeof("unmatched ‘{̋")))`, which checks for invalid command syntax and unmatched braces in the input. The modest size increase (averaging 2.4%) is due to the benchmark inputs already including cases designed to test error-handling capabilities. Thus, Mop's three-stage optimization implicitly accounts for error-handling as part of generality, ensuring that only a small amount of additional code is needed to address the identified crash and hang behaviors.

Table 15. Robustness evaluation results for the 8 benchmark programs. **Fuzzed Inputs** refers to the inputs generated to detect crash and hang behaviors. **Crash/Hang Inputs** indicates the number of inputs that actually cause crashes and hangs of the debloated program. **Size Increase** represents the percentage of statements added to the debloated program because of the robustness enhancement. **Robustness**∗ denotes the number of validation inputs that trigger crash or hang behaviors (see Section 5.2.4).

Program	Fuzzed Inputs	Crash/Hang Inputs	Size Increase	Robustness∗
TOTINFO	100	0	-	-
DATE-8.21	1750	9	2.8%	4/10
GZIP-1.3	100	0	-	-
MKDIR-5.2.1	390	0	-	-
SED-4.1.5	100	3	0.2%	1/2
TAR-1.14	830	406	4.1%	0/594
VIM-5.8	100	0	-	-
POSTGRESQL-12.14	1930	0	-	-

∗ The results of **Robustness** column are in x/y format. x and y are the numbers of inputs triggering crash/hang behaviors of the debloated programs generated **after** and **before** robustness enhancement.

MOP's post-processing step significantly improves the debloated programs' robustness by reducing an average of 70% robustness issues manifested as the crash/hang behaviors of the programs, while only reintroducing 2.4% of the statements.

6 Discussion

In this section, we revisit the key insights behind MOP's three-stage mutation strategy and analyze the consistency of scores. We also assess MOP's computational complexity, runtime overhead, and configurability. In addition, we explain the limitations of preserving critical dependencies during debloating and how MOP behaves on programs that require no debloating. Finally, we reflect on the inherent limitations of input-based debloating techniques.

Effectiveness of the Three-Stage Search. The design of MOP is inspired by the Hierarchical Delta Debugging (HDD) approach [60], which advocates starting with coarse-grained exploration and progressing to finer levels for efficient analysis. This coarse-to-fine principle guides MOP's three-stage process, where each stage targets a specific code granularity — coverage, segments, or statements. By beginning with broader mutations, MOP quickly narrows the search space before applying more precise optimizations, enhancing debloating efficiency.

The three-stage process leverages distinct mutation strategies to optimize program size while preserving functionality. In the first stage, coverage mutation uses test execution data to balance functionality against code size, eliminating or preserving large code regions. The second stage refines this by targeting coverage segments and partitioning statements into unique sets for optimization. Finally, the third stage performs statement-level mutations, enabling fine-grained exploration to remove individual statements with minimal impact on program generality.

To further evaluate the effectiveness of this staged approach, we implemented a "three-in-one" variant of MOP, which applies all three mutation models—coverage, coverage segments, and statements—simultaneously within a single cycle. Each mutation operator is used with equal frequency, flipping code by removing or restoring elements. This three-in-one design, intended for quick assessment, contrasts with MOP's sequential strategy, as simultaneous mutations risk producing syntactically or semantically invalid programs due to conflicting granularities.

We tested both Mop and its "three-in-one" variant on the *tar-1.14* program, following the experimental setup in Section 3, with $\alpha = 0.5$, $\beta = 0.5$, and a 1-hour time budget, excluding must-handle inputs to focus on the optimization ability. Our result shows that Mop achieved an O-Score$_{top}$ of 0.58 and removed 36% of the code, while the variant scored 0.54 and removed only 20%. A key reason for the variant's weaker performance is its lower sample acceptance rate — 42.6% compared to Mop's 61.3%. This means the "three-in-one" variant's mixed-granularity mutations led to premature code fragmentation, generating many invalid samples and reducing generality, whereas Mop's staged approach ensured more effective debloating.

Computational and Space Complexities and Configurability. Mop's three-stage optimization, utilizing Markov Chain Monte Carlo (MCMC) search with models targeting coverages, segments, and statements, introduces additional complexity compared to Debop's single-stage, statement-only approach. The added complexity mainly stems from the first two stages, which require tracking coverages and segments for MCMC sampling. Overall, however, the additional overheads are minimal. To understand this, we provide analytical Big-O analyses of the computational and space complexities of Mop and Debop for a comparison.

Since Debop operates in a top-down manner by reducing code from the original (top) program for debloating and does not account for must-handle inputs, we analyze the complexities of Mop and Debop using the top program p_{top} as the initial sample without accounting for the influence of any must-handle inputs. The complexities of Mop's optimization with the base program as the initial sample are similar. To focus on the core optimization abilities, we also exclude Mop's additional pre- and post-processing steps, which are only one-time efforts and are much less expensive.

Before delving into the analyses, we give the following definitions. Let N be the number of statements in p_{top} (code covered by any must-handle inputs is filtered), M the number of inputs in the usage profile, C the number of coverages (with code covered by the must-handle inputs filtered), and S the number of coverage segments (also with code covered by the must-handle inputs filtered).

Computational complexity. Here we analyze the computational costs of Mop's and Debop's optimizations. Recall that Mop operates in three stages. In each stage, Mop proceeds iteratively to create a series of samples. In each iteration, Mop selects an element (coverage for Stage-I, coverage segment for Stage-II, and statement for Stage-III) from the current sample to mutate (either remove or preserve) and generate a new sample to evaluate. The search space of Stage-I is 2^C (as each element can be preserved or removed, resulting in a total of 2^C samples). Likewise, the search spaces of Stage-II and Stage-III are 2^S and 2^N respectively. Unlike Mop, because Debop only mutates statements for optimization, its search space is 2^N.

Let Φ_i be the per-iteration cost of Mop in stage i (including sample construction, compilation, and possibly also validation) and τ_i be the number of MCMC iterations required for a sufficient exploration (the mixing time). Intuitively, τ_i is the number of iterations needed for the search to converge. Let Φ and τ denote the per-iteration cost and the mixing time for Debop. Now we have the computational complexities of Mop and Debop as follows.

$$O_{\text{compute}}(\text{Mop}) = O(\Phi_1\tau_1 + \Phi_2\tau_2 + \Phi_3\tau_3), \quad O_{\text{compute}}(\text{Debop}) = O(\Phi\tau). \tag{18}$$

We note that $O_{\text{compute}}(\text{Mop})$ is approximately the same as $O_{\text{compute}}(\text{Debop})$. To understand this, let us first analyze the per-iteration costs. We have $\Phi_2 = \Phi_3 = \Phi$, each involving creating, compiling, and validating a sample (running it against the inputs for generality evaluation). Φ_1 is smaller than Φ_2 (Φ_3 or Φ), and we have $\Phi_1 < \Phi_2 = \Phi_3 = \Phi$, since Φ_1 only involves sample creation and compilation. The cost of sample validation is saved, as the coverage is derived from the execution of each input, and by removing or retaining a coverage, Mop can quickly decide whether the sample (program) correctly handles the corresponding input without having to execute the program.

Second, for the τ part, the mixing time, which is the number of iterations needed for a sufficient exploration, is positively correlated with the search space. Intuitively, the larger the search space, the greater the number of iterations for the exploration to converge. So we have

$$\tau_1 \propto 2^C, \quad \tau_2 \propto 2^S, \quad \tau_3 \propto 2^N, \quad \tau \propto 2^N. \tag{19}$$

Here, we note that for real programs, N is considerably greater than either C or S, as C and S are related to the number of inputs and are much smaller. For example, we have $C = 84$, $S = 191$, and $N = 3644$ for *tar-1.14*. As a result, we have $2^C \leq 2^S \ll 2^N$, which means $\tau_1 \leq \tau_2 \ll \tau_3 \approx \tau$. Finally, given that $\Phi_1 < \Phi_2 = \Phi_3 = \Phi$ and $\tau_1 \leq \tau_2 \ll \tau_3 \approx \tau$, we have $\Phi_1\tau_1 < \Phi_2\tau_2 \ll \Phi_3\tau_3 \approx \Phi\tau$, or $O_{\text{compute}}(\textsc{Mop}) \approx O_{\text{compute}}(\textsc{Debop})$.

Space complexity. The space complexities of Mop and Debop are also approximately the same. To see this, let us first show Mop's complexity. Mop maintains an $C \times N$ coverage matrix that records, for each input, the set of statements covered. This matrix is used not only in Stage-I to determine whether a sample correctly handles an input by checking if the statements covered by the input are retained in the sample, but also in Stage-II and -III to collect the coverage segments and compute the transition probabilities. The coverage matrix requires $O(CN)$ space. During optimization, Mop additionally maintains bit vectors to record coverages ($O(C)$), segments ($O(S)$), and statements ($O(N)$), along with an array for the validation results ($O(M)$) and a segment-to-statement mapping ($O(S + N)$). Taken together, the space complexity of Mop is as follows.

$$O_{\text{space}}(\textsc{Mop}) = O(CN + C + S + M + N) \tag{20}$$

Next, let us show Debop's space complexity. Debop adopts a simple statement mutation model for optimization, and it only maintains a statement index vector ($O(N)$) and an array for saving the results of input execution ($O(M)$). Thus, Debop's space complexity is as follows.

$$O_{\text{space}}(\textsc{Debop}) = O(M + N) \tag{21}$$

As we have previously explained, $C \leq S \ll N$. Also, the number of inputs M is often much smaller than N. These make both $O_{\text{space}}(\textsc{Mop})$ and $O_{\text{space}}(\textsc{Debop})$ approximately $O(N)$.

It is worth noting that although Mop consumes more memory than Debop because of maintaining additional data structures, the extra overhead is practically limited and affordable. For example, in the case of debloating *tar-1.14*, Mop's peak memory usage was approximately 0.53 GB, and Debop's usage was 0.37 GB. The average CPU utilization was about 10% for both.

Configurability. For large and complex programs, Mop's full debloating process can be slow and thus not ideal in real-time or high-throughput scenarios. However, Mop's configurability allows adjustment of search budgets across its stages via the time-allocation parameters (t_1, t_2, t_3). Mop's first stage is fast, whereas the last two are more expensive, as they require repeated dynamic evaluation of any reduced program generated along the search by executing the program against all inputs. Analytically, Mop's first stage optimization has a computational complexity of $O_{\text{compute}}(\textsc{Mop}\text{-}S_1) = O(\Phi_1\tau_1)$, while the first two stages incur a complexity $O_{\text{compute}}(\textsc{Mop}\text{-}S_{1,2}) \approx O(\Phi_2\tau_2)$, both of which are significantly lower than Debop's $O_{\text{compute}}(\textsc{Debop}) = O(\Phi\tau)$. In time-sensitive scenarios or when dealing with large-scale software systems, more time budget can be allocated to the coarse-grained stage, while the fine-grained stages can be scaled back as needed. Results from Section 5.4.4 showed that using less time for optimization may yield similar results. Moreover, there can be ways to improve the efficiency of Mop's last two stages, for example, by executing the program against the inputs in parallel to speed up program evaluation or by using a small representative sample of inputs for initial evaluation and only performing full evaluation if the initial evaluation is satisfactory.

Dependency Preservation for Debloating. Mop does not enforce the preservation of code dependencies while debloating. In fact, debloating inevitably breaks dependencies, and enforcing various types of dependencies to preserve would render debloating less effective or even infeasible. As evidence to support this, we created a dependency-preserving variant of Mop and evaluated it using the experiment from Section 3 (tar-1.14, $\alpha = 0.5$, $\beta = 0.5$, and a 1-hour budget). The variant uses the DG tool [59] to capture various types of dependencies, including the control and data flow dependencies, pointer-analysis-based dependency, and value-relation dependency. During the optimization process, the variant rejects any sample derived from code removal that breaks any of these dependencies. Our results show the high rejection rates incurred by dependency-preserving debloating—92% in the first stage, 85.8% in the second, and 62.6% in the third—resulting in a small (7%) code reduction and virtually no attack surface reduction. This result emphasizes how enforcing full dependency preservation damages debloating efficiency and weakens attack surface removal.

From Mop's optimization perspective, it would be beneficial to remove unneeded/unfavorable features and break their code dependencies. Attempting to preserve all dependencies, regardless of their relevance to the target use case, introduces unnecessary code. That said, if the user of Mop values certain types of dependencies, she could define a function measuring the degree to which certain types of dependencies were compromised due to code reduction. That function can be incorporated into the objective function to guide the search.

How Would Mop Handle Programs that Do Not Need Debloating? When Mop is applied to a program that does not need debloating, it will still perform its optimization process but can end up returning the original program unchanged. This can happen because using an objective function accounting for weighted reduction and generality, Mop can decide that any sample (i.e., reduced program) examined during its search is unfavorable. For example, it is possible that reducing any code can lead to the generation of a program whose generality is significantly compromised (especially true for small programs), in which case, Mop reduces no code. In our experiments, for example, Mop reduced no code for *tcas*, a small program from SSIR, in the experiment where a high β is specified — it rejected any sample during the search process, as it decided that no sample examined is better than the original program in terms of the reduction-generality trade-off. For complex programs, however, previous studies [74, 96] as well as our observation both show that no opportunities for debloating are rare.

Limitations of Input-Based Debloating Techniques. The design of Mop relies on a usage profile derived from concrete inputs to specify features for debloating, a practice shared by many feature-based debloating techniques (e.g., [7, 35, 71]). While not optimal, input-based profiles are widely adopted due to their general applicability across diverse programs. Alternative feature specification methods, such as manual code annotation or application-specific features (e.g., activities in Android apps [52]), are less prevalent owing to their labor-intensive nature or limited scope. To our knowledge, no other specification approach matches the ubiquity of concrete inputs for general-purpose program debloating.

A key limitation of debloating techniques using input-based specifications is the tendency to produce debloated programs that are overfitted to the given inputs, failing to generalize beyond the provided inputs. This challenge is inherent to any technique relying on input profiles for debloating. To mitigate overfitting, we introduced a post-processing step in Mop to enhance program robustness. Preliminary results suggest that this step improves robustness, but further strategies could be explored. For instance, incorporating diverse input sets or leveraging dynamic analysis to validate feature coverage may reduce overfitting. We consider such enhancements as promising directions for future research to advance the reliability and generality of debloated programs.

7 Threats to Validity

In this section, we discuss the potential internal threats, external threats, and construct threats that may affect the validity of our experimental results.

7.1 Internal Threats

The first threat is any implementation error of Mop. To mitigate this threat, during the development of Mop, we conducted meticulous code review and thorough testing. We also made Mop and our results available for public review and extensions. The second threat relates to the validity of other tools. We note that we used most of the tools provided by the authors (as explained in Section 5.2.3). Debop-M is created by us and is built upon Debop. The difference between Debop-M and Debop is that Debop-M allows the optional, must-handle inputs that specify the features that must be preserved in the debloated program. We carefully tested the must-handle functionality of Debop-M before using Debop-M in the experiment. The third threat concerns the evaluation of a debloated program in terms of the G-Score. The evaluation involves running the program against each of the inputs with a timeout value specified to avoid non-termination. Because the inputs are many, a large timeout value can make the evaluation of any non-terminating debloated program unnecessarily long. Yet, a small value can occasionally cause a normal program execution to break in our environment where experiments were run in parallel. To mitigate such a threat to the validity of the evaluation, we ran each program against each input multiple times in our environment (where parallel experiments were enabled), recorded the longest execution time observed t, and set the timeout value to $1.5t$.

While our results show that Mop's staged optimization quickly generated high-quality debloated programs, the very nature of its sequential search introduces a risk of missing a globally optimal solution. Because each stage picks the locally best sample before passing it on to the next, the early suboptimal choice can steer the later stages into a less promising region of the search space. Nevertheless, Mop is empirically more effective than its variants performing an exhaustive one-stage search using each of the three mutation models (Section 5.6.2) and a three-in-one model (Section 6).

7.2 External Threats

First, our evaluation is based on 26 programs and their associated inputs from the benchmarks. The results and conclusions may not generalize to other programs and inputs. We note however that most (or 25) of the programs and their inputs are from existing benchmarks [35, 96], and we additionally used *postgresql-12.14* to investigate how existing tools perform for a large application. The inputs of *postgresql-12.14* are also from the application's official repository. Albeit limited, the set of subject programs are diverse in terms of sizes — they range from small ones with just a few hundred LoC (e.g., *tcas*) to medium programs with up to 100K LoC (*vim-5.8*) and to the large application having over 930K LoC.

Second, our results might be subject to the experiment parameters we used and may not generalize to other parameters. The threat however is minimal, as we experimented with different α and β values and used the same k value from the previous evaluation [94], which has demonstrated the proper selection of the value for density function evaluation.

7.3 Construct Threats

Apart from the reduction and tradeoff scores computed based on a comparison of the debloated program and the original program, we introduced new scores evaluated against the top program. The new scores are not traditional, and yet, they reveal the effectiveness of optimization, the core part of Mop. Our implementation and evaluation of Mop only involve three factors: size reduction, attack surface reduction, and generality for optimization. They could be extended with more factors such as performance and energy consumption.

8 Related Work

To deal with the negative influence of code bloat, many program debloating techniques have been proposed. They can be classified as feature-based, static analysis-based, library-based, Android apps-oriented, web applications-oriented, and other specific tasks-oriented approaches (targeting for example OS kernels, containers, and Java projects).

Feature-based Debloating. Feature-based debloating techniques aim to identify and eliminate unnecessary features of programs and produce reduced versions of these programs. They adopt various strategies to achieve debloating. Many use a set of inputs to represent the expected features to preserve and prune the undesired features.

Mop and Debop take into account both generality and reduction. They use stochastic optimization to seek the optimal tradeoff between code reduction and generality. The key difference between Mop and Debop is the way performed for code reduction. Debop uses a simple mutation model that removes one statement each time for sample generation, which leads to its weak reduction capability. Mop achieves effective code reduction through coarse-grained code exploration while supporting fine-grained code exploration. In addition, compared to Debop, Mop eliminates dead code to remove unneeded code and allows users to define features they believe must be retained. Mop is also different in that it supports the use of must-handle inputs to specify the must-preserve features and that it uses fuzzing-guided code augmentation for program robustness enhancement. DomGad [95] specializes in reducing the size of programs by focusing on the specific subdomains they operate in. It identifies targeted subdomains of functionalities and performs the stochastic search to debloat the program.

Apart from these three techniques, there are optimization-based techniques do not focus on generality. MoMS [8] is designed to optimize software applications by minimizing their size while balancing multiple objectives such as performance, energy consumption, and resource usage. It employs a learning-driven approach to seek tradeoffs, ensuring the optimized software meets the desired criteria without significantly compromising functionalities or user experience. PolyDroid [34] aims for mobile application customization, leveraging learning-driven techniques to optimize applications based on user behavior and preferences, enhancing user experience and application performance.

Many debloating techniques are reduction-oriented. They focus on code reduction and do not account for generality. Chisel [35] leverages delta debugging [60, 100] to remove code, and it uses reinforcement learning to optimize the search process. It aims to generate a minimal program that can correctly handle these inputs. Because Chisel needs to execute inputs frequently to validate each program it generates, the search process is time-consuming when the input set contains many inputs. Since Chisel only cares about generating a debloated program that behaves correctly for the given inputs, the program can overfit the inputs and has low generality. To address the low efficiency of delta debugging, BLADE [7] makes use of the syntactic structure of the target program and does a reverse traversal of the syntax tree for code removal. To speed up the reduction process, it leverages a heuristic method to identify nodes that can be removed in parallel. However, it only traverses each node once, which makes it more likely to obtain a locally optimal solution rather than a globally optimal solution.

Razor [71] is also an input-based technique. Unlike Chisel, it uses a trace-based method with control-flow-based heuristic methods to infer feature-related code. Razor starts with instrumenting the original program, executing the program with given inputs to obtain coverage information, and building a control flow graph. Razor can quickly eliminate all the code that is not covered by traces to produce a reduced program. Then, to handle unseen inputs, Razor uses a set of heuristics to identify and preserve code that is not exercised by the given inputs but is feature-related. Razor operates on binary files, so it cannot utilize source code information.

JDBL [86] is also a coverage-based technique. It keeps track of the code of the program and the dependencies that are covered by the given inputs and generates the reduced program. However, different from Razor, JDBL does not perform code inference to identify feature-related code. Similar to JDBL, Cov [94], used for C programs, preserves the code that is covered by given inputs. To improve the generality of reduced programs, CovF and CovA [96] use fuzz-based and analysis-based methods to identify and preserve feature-related code. ANCILE [16] also uses fuzzing to perform debloating. It uses fuzzing to generate extra inputs, which cover more feature-related code. In this way, ANCILE can generate reduced programs that are more likely to behave as the users expect. Additionally, ANCILE performs CFI [1] to enhance the security of the debloated program. LEADER [50] leverages an LLM to perform feature-based debloating by understanding program documentation and desired features expressed as test cases. It generates augmented tests to improve feature coverage and debloats the program accordingly, while a security advisor uses static analysis to restore missing security checks. Similar to Mop, these techniques also use coverage information, and some of them use methods to expand the code contained in the reduced program, with the goal of identifying more feature-related code. However, unlike Mop, they do not perform optimization for debloating.

OCCAM [57], OCCAM-v2 [62], DeepOCCAM [47], Trimmer [4, 83], C2C [31], and LMCAS [6] are debloating techniques that leverage compiler optimization for code removal. OCCAM performs program specialization via partial evaluation based on a user-specified configuration (e.g., a command-line option). OCCAM-v2 is an extension of OCCAM, and it uses more advanced static analysis for code removal. DeepOCCAM improves OCCAM by using reinforcement learning to train a debloating policy that identifies functions where optimization should be applied. Trimmer applies more precise constant propagation and more aggressive loop unrolling strategies for program specialization. C2C is a generic configuration-based attack surface reduction technique that utilizes static code analysis and instrumentation to map configuration options to application code automatically. LMCAS performs partial interpretation to capture a precise program state of the configuration logic based on the given inputs. It then applies partial-evaluation optimizations to generate a specialized program by propagating constants, eliminating unwanted code, and preserving desired functionalities. These techniques are based on configuration and optimization, and they are effective in program debloating. Unlike Mop, these techniques target specialization and do not perform stochastic optimization for debloating.

In addition to the above techniques that use a set of tests or a user-provided configuration to specify the expected features, CARVE [17] requires source code annotations to mine the features of the program and the code that is mapped to each feature. The feature mapping established by the annotation facilitates fast and effective debloating. Unfortunately, annotating source code involves expensive human efforts and is not user-friendly. PRAT [93] is designed to remove features from IoT (Internet of Things) third-party software components. It automatically identifies available features by analyzing the build system and removes unneeded features based on code coverage analysis. XDebloat [88] also does not require users to describe the desired features. Instead, it generates a file containing all identified features, and users just need to specify the features to be retained. It uses manual annotations in the source code or automatic identification to identify features at all granularities, providing options for users to select. IOSPReD [63] uses user-provided inputs to identify and track necessary data chunks, automatically packaging only these essential chunks along with the application in a container. Mansouri et al. [58] presented a system that detects and disables features activated by common input to reduce software vulnerability risks. MiniMon [52] collects executed methods during monitoring and utilizes program analysis, graph clustering, and graph embedding techniques to identify additional methods related to desired features. Finally, it generates reduced apps by removing the unrelated methods.

Static Analysis-based Debloating. Static analysis is commonly used in program debloating because it can discriminate between necessary and unnecessary code. It involves analyzing the source code or compiled code of a program to identify dead code, unreachable code, and unused libraries or dependencies. Once these unnecessary components are identified, they are removed or optimized, resulting in a more efficient program.

JRed [42] converts Java code into intermediate code Jimple [90], builds a call graph based on static analysis, and then removes methods and classes that are not included in the call graph to achieve program reduction. Decker [66] is a technique that does not require inputs. It uses static analysis to determine key function sets that should be disabled at runtime and only loads the code that needs to be executed on demand to reduce the attack surface. Mininode [45] is a static analysis tool that measures and removes unused code in Node.js applications. It first parses the program and generates AST, then builds the file-level dependency graph and the call graph, and finally removes the unused module identified in the two graphs. BinTrimmer [77] is a static program debloating technique built on the abstract domain. It utilizes abstract interpretation techniques to reliably remove redundant code from binaries, improving efficiency and reducing file size. Pashakhanloo et al. presented PacJam [64], a framework that uses static analysis to remove all unreachable packages and unimplemented packages. It provides a fallback mechanism that seamlessly installs the original package to guarantee robustness when calling the removed package.

Debloating for Libraries. Developers frequently import numerous features from program libraries to streamline software development, preventing redundancy and enabling the creation of new applications with ease. However, libraries typically encompass lots of features, of which only a fraction may be necessary for a specific program. To address the bloat of libraries, lots of debloating techniques have been proposed.

Piece-Wise [75] analyzes call dependencies during library compilation to load only necessary code during application startup, thus improving software efficiency. Nibbler [3] is a system that identifies and erases unused functions within dynamic shared libraries. It processes the shared libraries and reconstructs the function call graph (FCG) of each library. Then, the functions required by applications and the already-extracted library FCGs are composed to determine functions that are never called. BlankIt [67] uses a predictor to selectively load library functions. The predictor is a decision tree-based learner that acts as an input-sensitive predictor and provides a list of required functions. However, an attacker may be able to revive the attack by fabricating the context. Wang et al. address this by proposing Picup [91]. It employs the Convolutional Neural Network (CNN) with an attention mechanism to extract key bytes from the input and map them to library functions, predicting the necessary library functions as soon as the input is obtained, thus removing unused code before attackers tamper with data. μTrimmer [102] is a tool that removes unused basic blocks from shared libraries without extra runtime or memory overhead. It introduces a method to identify address-taken blocks/functions, maintains an inter-procedural control flow graph to include potentially useful library code, and finally removes unused code. JSLIM [99] obtains the mapping relationship between the vulnerability and the NPM package and determines which function in the package causes a specific vulnerability. Then, based on the generated function call graph, JSLIM removes unused code and uses sandbox isolation for code that contains vulnerabilities but cannot be removed. Slimming [85] performs static analysis on Java reflections, which are commonly used by popular frameworks, and parses configuration files to identify reflection targets. It then resolves the string arguments of reflection APIs relevant to users, identifying all necessary dependencies.

This helps developers evaluate debloating strategies by weighing the benefits of removing bloat against the costs of managing dependencies.

Debloating for Android Apps. Existing programming practices for building Android apps follow a one-size-fits-all strategy to meet the needs of different users and adapt to different mobile platforms. Running a large amount of user-irrelevant code will further strain the already limited resources of mobile platforms (e.g., battery power and memory usage). Jiang et al. propose RedDroid [41], a static analysis-based approach to identify and eliminate unnecessary code in Android applications. It mainly removes dead code and redundant ABI (Application Binary Interface). Huang et al. [39] proposed a UI-driven approach to debloat Android applications, which removes user-selected UI elements and their associated UI elements and functionality, such as removing a button and its corresponding implementation. Liu et al. proposed AutoDebloater [51], which provides a web page that allows users to view the activity transformation graph (generated by StoryDistiller [20]), select unwanted activities, and delete them. ACVCut [65] can achieve 100% code coverage for third-party Android applications by debloating binary files without source code access permissions. LegoDroid [53] automatically decomposes an Android application for flexible loading and installation while preserving the expected functionality with a fast and instant application load. Madusa [48] constructs a graph representing dependencies among methods and resources and identifies a sub-part of the graph using integer linear programming to generate a reduced version that behaves as similarly as possible to the original application.

Debloating for Web Applications. Modern browsers and web applications are constantly evolving to meet the needs of diverse users. Given their vast user base and extensive codebases, these programs inevitably contain numerous attack surfaces, removing the attack surface is worth researching. Slimium [72] successfully make efforts to debloat browsers. It employs a hybrid approach (static, dynamic, and heuristic) to remove unwanted features from Chromium binaries at a functional granularity based on a predefined feature-code map (via semi-automatic analysis) and user-specified websites. For web applications, Azad et al. [11] proposed a dynamic analysis-based approach to eliminate bloat in PHP applications. It uses a set of automated tools and scripts to simulate interactions between the client and server, recording the files and lines executed on the server side. This process helps identify necessary code and debloat the program at both the file and function levels. Afterward, Azad and Nikiforakis conducted a user study to investigate how developers interact with web applications. Then, they combined this result with their proposed tool DBLTR [9] to collect coverage traces of users, forming clusters of users with similar usage behaviors, and customizing different reduced applications based on each user's needs. Stubbifier [89] identifies dead code by building static or dynamic call graphs from the test cases. It then replaces unreachable code with either file-level or function-level stubs capable of fetching and executing the original code dynamically. AnimateDead [10] is a PHP emulator that employs a hybrid debloating approach based on concolic execution, improving web application security by reducing code size and vulnerabilities without the runtime overhead of traditional debloating techniques. Minimalist [40] generates a call-graph for a given PHP web application and performs a reachability analysis for the features required by users and removes unreachable functions in the analyzed web application.

Debloating for Other Specific Tasks. There are also many studies that focus on other specific tasks. Hacksaw [38] specializes the OS kernel to the target machine based on additional hardware components. Cimplifier [76] uses dynamic analysis to partition a large container into many simple and isolated containers, which contain only the resources needed to implement permission separation. Additionally, a recent study [33] also evaluated three container debloaters [30, 49, 84] based on metrics such as category-based system call reduction, mitigated CVEs, size reduction, and execution correctness. It found that all three techniques were effective at debloating containers, but only two of them [30, 49] could produce correct debloated containers. MMLB [101] is a framework that focuses on machine learning containers. It measures the amount of bloat at both the container and package levels, quantifying the sources of bloat. Gharachorlu et al. [29] proposed a type batched debloating technique, which uses machine learning to suggest portions of a program, or batches, that are most likely to be advantageous to reduce in the reduction. SCARF [82] is a product deprecation system that identifies unused code and data assets and safely removes them. It operates automatically and gathers developer inputs where it cannot take automated actions, leading to further removals. Soto et al. [87] proposed a technique to specialize third-party dependencies of Java projects, removing unused dependencies and unused classes from the remaining dependencies. Zhang et al. [103] presented a debloating tool for address sanitizers to reduce its runtime overhead. Kalhauge [44] used the propositional Boolean logic to specify dependencies, which effectively searches for valid sub-inputs while avoiding invalid sub-inputs and then debloated Java byte code. JShrink [18] is also a technique used for Java bytecode debloating. It combines static analysis with dynamic analysis and type dependency analysis to identify unused methods and fields for removal. DynaCut [56] is a dynamic software customization tool that can enable or disable unused code features at runtime without needing access to the source code. It achieves this through dynamic process rewriting and execution trace-based differential analysis.

Different from these techniques, which are tailored for a particular type of code or application for debloating, Mop is a general input-based, generality-aware technique that treats debloating as an optimization problem and aims to generate a debloated program with the best tradeoff between code reduction and program generality.

9 Conclusion and Future Work

Mop is a novel technique that performs stochastic optimization for feature-based debloating. Taking as input a (bloated) program and a usage profile manifested as a set of inputs exercising the program's various features, Mop performs a preprocessing step and then goes through three stages for stochastic optimization. The goal is to generate a debloated program achieving the best tradeoff between reduction and generality, the two key factors that affect the quality of debloating. In the preprocessing step, Mop identifies an initial sample. Starting with the sample, Mop proceeds with a three-stage optimization process to create new samples using an MCMC-based stochastic algorithm guided by an objective function that quantifies the reduction-generality tradeoff. Mop uses different mutation models in different stages to allow both aggressive and fine-grained exploration for optimization. It selects the best sample with the highest objective value as the output of each stage. Finally, in the postprocessing step, Mop performs fuzzing-guided augmentation to improve the robustness of the last stage's output and reports the augmented program as the debloating result.

The result of our evaluation demonstrates the effectiveness of Mop. It shows that Mop can prune a significant amount (nearly 70%) of code to produce a debloated program with high generality and thus achieve a good reduction-generality tradeoff (0.76); that using different weights, Mop can generate debloated programs with different tradeoffs; that Mop outperforms Debop in achieving a better tradeoff in most of the experiments and the improvement of the tradeoff score is over 15%; that even as an optimization-oriented technique, Mop can achieve a reduction score that is higher than Chisel and its reduction ability is at least comparable to Razor's; and that Mop's fuzzing-guided method can effectively improve the debloated program's robustness.

In future work, we plan to explore advanced convergence conditions that are not purely time-based to improve Mop's optimization effectiveness. We will also investigate a more flexible comparison of the reduction and generality changes (via for example introducing an adaptive weight in the objective function) to allow the acceptance of promising samples in extreme cases. Because the last two stages of Mop require repeated program execution for generality evaluation and are costly, we also plan to improve their efficiency. One way to achieve this is to leverage static analysis to identify mutation-affecting inputs and only use those inputs for generality calculation. Finally, we hope to extend the experiments by including more and larger programs of diverse types for evaluation and have a better understanding of the unique strengths of Mop's last optimization stage.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China under the grant numbers 62202344, 62572363, and 62472326, and CCF-Zhipu Large Model Innovation Fund (No.CCF-Zhipu202408).

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009), 40 pages. doi:10.1145/1609956.1609960

[2] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association, 2435–2452. https://www.usenix.org/conference/usenixsecurity21/presentation/abubakar

[3] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-scale Debloating of Binary Shared Libraries. *Digital Threats* 1, 4, Article 19 (Dec. 2020), 28 pages. doi:10.1145/3414997

[4] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Fareed Zaffar, and Junaid Haroon Siddiqui. 2022. Trimmer: An Automated System for Configuration-Based Software Debloating. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3485–3505. doi:10.1109/TSE.2021.3095716

[5] akihe. 2024. Radamsa. https://gitlab.com/akihe/radamsa

[6] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. 2022. Lightweight, Multi-Stage, Compiler-Assisted Application Specialization. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. 251–269. doi:10.1109/EuroSP53844.2022.00024

[7] Muaz Ali, Rumaisa Habib, Ashish Gehani, Sazzadur Rahaman, and Zartash Uzmi. 2023. BLADE: Scalable Source Code Debloating Framework. In *IEEE Secure Development Conference (SecDev)*. IEEE, 75–87. doi:10.1109/SecDev56634.2023.00022

[8] Nasir Ali, Wei Wu, Giuliano Antoniol, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Jane Huffman Hayes. 2011. MoMS: Multi-objective miniaturization of software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 153–162. doi:10.1109/ICSM.2011.6080782

[9] Babak Amin Azad and Nick Nikiforakis. 2023. Role Models: Role-based Debloating for Web Applications. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy* (Charlotte, NC, USA) *(CODASPY '23)*. Association for Computing Machinery, New York, NY, USA, 251–262. doi:10.1145/3577923.3583647

[10] Babak Amin Azad, Rasoul Jahanshahi, Chris Tsoukaladelis, Manuel Egele, and Nick Nikiforakis. 2023. AnimateDead: Debloating Web Applications Using Concolic Execution. In *32nd USENIX Security Symposium (USENIX Security '23)*. USENIX Association, Anaheim, CA, 5575–5591. https://www.usenix.org/conference/usenixsecurity23/presentation/azad

[11] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security '19)*. USENIX Association, Santa Clara, CA, 1697–1714. https://www.usenix.org/conference/usenixsecurity19/presentation/azad

[12] BaiGeiQiShi. 2024. Mop. https://github.com/BaiGeiQiShi/Mop

[13] BaiGeiQiShi. 2024. Mop Benchmark. https://github.com/BaiGeiQiShi/MopBenchmark

[14] BaiGeiQiShi. 2024. Mop-RQ1. https://github.com/BaiGeiQiShi/Mop/tree/main/Results/RQ1-Curves

[15] Suparna Bhattacharya, Karthick Rajamani, K. Gopinath, and Manish Gupta. 2011. The interplay of software bloat, hardware energy proportionality and system bottlenecks. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems* (Cascais, Portugal) *(HotPower '11)*. Association for Computing Machinery, New York, NY, USA, Article 1, 5 pages. doi:10.1145/2039252.2039253

[16] Priyam Biswas, Nathan Burow, and Mathias Payer. 2021. Code Specialization through Dynamic Feature Observation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy* (Virtual Event, USA) *(CODASPY '21)*. Association for Computing Machinery, New York, NY, USA, 257–268. doi:10.1145/3422337.3447844

[17] Michael D. Brown and Santosh Pande. 2019. CARVE: Practical Security-Focused Software Debloating Using Simple Feature Set Mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (London, United Kingdom) *(FEAST'19)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3338502.3359764

[18] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: in-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE '20)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/3368089.3409738

[19] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/1455770.1455776

[20] Sen Chen, Lingling Fan, Chunyang Chen, and Yang Liu. 2023. Automatically Distilling Storyboard With Rich Features for Android Apps. *IEEE Transactions on Software Engineering* 49, 2 (2023), 667–683. doi:10.1109/TSE.2022.3159548

[21] CIL Project. 2024. CIL. https://github.com/cil-project/cil

[22] CIL Project. 2024. CIL Merger. https://people.eecs.berkeley.edu/~necula/cil/merger.html

[23] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Routledge, New York, NY. doi:10.4324/9780203771587

[24] Inc Docker. 2024. Docker. https://www.docker.com/

[25] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. 2012. How much does unused code matter for maintenance?. In *2012 34th International Conference on Software Engineering (ICSE)*. 1102–1111. doi:10.1109/ICSE.2012.6227109

[26] Catherine O. Fritz, Peter E. Morris, and Jennifer J. Richler. 2012. Effect Size Estimates: Current Use, Calculations, and Interpretation. *Journal of Experimental Psychology: General* 141, 1 (2012), 2–18. doi:10.1037/a0024338

[27] Andrew Gelman and Donald B. Rubin. 1992. Inference from Iterative Simulation Using Multiple Sequences. *Statist. Sci.* 7, 4 (1992), 457 – 472. doi:10.1214/ss/1177011136

[28] John Geweke. 1991. *Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments.* Staff Report 148. Federal Reserve Bank of Minneapolis. https://ideas.repec.org/p/fip/fedmsr/148.html

[29] Golnaz Gharachorlu and Nick Sumner. 2023. Type Batched Program Reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '23)*. Association for Computing Machinery, New York, NY, USA, 398–410. doi:10.1145/3597926.3598065

[30] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458. https://www.usenix.org/conference/raid2020/presentation/ghavanmnia

[31] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-grained Configuration-driven System Call Filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1243–1257. doi:10.1145/3548606.3559366

[32] W.R. Gilks, S. Richardson, and D. Spiegelhalter. 1995. *Markov Chain Monte Carlo in Practice.* CRC Press. https://books.google.co.kr/books?id=T2G1DwAAQBAJ

[33] Muhammad Hassan, Talha Tahir, Muhammad Farrukh, Abdullah Naveed, Anas Naeem, Fareed Zaffar, Fahad Shaon, Ashish Gehani, and Sazzadur Rahaman. 2023. Evaluating Container Debloaters. In *2023 IEEE Secure Development Conference (SecDev)*. 88–98. doi:10.1109/SecDev56634.2023.00023

[34] Brian Heath, Neelay Velingker, Osbert Bastani, and Mayur Naik. 2019. PolyDroid: Learning-Driven Specialization of Mobile Applications. https://arxiv.org/abs/1902.09589

[35] 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* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. doi:10.1145/3243734.3243838

[36] Curt Hibbs, Steve Jewett, and Mike Sullivan. 2009. *The Art of Lean Software Development: A Practical and Incremental Approach* (1st ed.). O'Reilly Media, Inc.

[37] Gerard J. Holzmann. 2015. Code Inflation . *IEEE Software* 32, 02 (March 2015), 10–13. doi:10.1109/MS.2015.40

[38] Zhenghao Hu, Sangho Lee, and Marcus Peinado. 2023. Hacksaw: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1994–2008. doi:10.1145/3576915.3623208

[39] Jianjun Huang, Yousra Aafer, David Perry, Xiangyu Zhang, and Chen Tian. 2017. UI driven Android application reduction. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 286–296. doi:10.1109/ASE.2017.8115642

[40] Rasoul Jahanshahi, Babak Amin Azad, Nick Nikiforakis, and Manuel Egele. 2023. Minimalist: Semi-automated Debloating of PHP Web Applications through Static Analysis. In *32nd USENIX Security Symposium (USENIX Security '23)*. USENIX Association, Anaheim, CA, 5557–5573. https://www.usenix.org/conference/usenixsecurity23/presentation/jahanshahi

[41] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 189–199. doi:10.1109/ISSRE.2018.00029

[42] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 12–21. doi:10.1109/COMPSAC.2016.146

[43] Elmar Juergens, Martin Feilkas, Markus Herrmannsdoerfer, Florian Deissenboeck, Rudolf Vaas, and Karl-Heinz Prommer. 2011. Feature Profiling for Evolving Systems. In *2011 IEEE 19th International Conference on Program Comprehension*. 171–180. doi:10.1109/ICPC.2011.12

[44] Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI '21)*. Association for Computing Machinery, New York, NY, USA, 1003–1016. doi:10.1145/3453483.3454091

[45] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 121–134. https://www.usenix.org/conference/raid2020/presentation/koishybayev

[46] James LARUS. 2009. Spending Moore's Dividend. *Commun. ACM* (2009). doi:10.1145/1506409.1506425

[47] Nham Le Van, Ashish Gehani, Arie Gurfinkel, Susmit Jha, and Jorge A Navas. 2019. Reinforcement Learning Guided Software Debloating. https://www.csl.sri.com/users/gehani/papers/MLSys-2019.DeepOCCAM.pdf

[48] Jaehyung Lee, Hangyeol Cho, and Woosuk Lee. 2023. Madusa: Mobile Application Demo Generation Based on Usage Scenarios. *Automated Software Engineering* 30, 1 (2023), 8. doi:10.1007/s10515-022-00372-8

[49] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-Phase Execution of Application Containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Michalis Polychronakis and Michael Meier (Eds.). Springer International Publishing, Cham, 230–251.

[50] Bo Lin, Shangwen Wang, Yihao Qin, Liqian Chen, and Xiaoguang Mao. 2025. Large Language Models-Aided Program Debloating. *IEEE Transactions on Software Engineering* 51, 9 (2025), 2651–2670.

[51] Jiakun Liu, Xing Hu, Ferdian Thung, Shahar Maoz, Eran Toch, Debin Gao, and David Lo. 2023. AutoDebloater: Automated Android App Debloating. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2090–2093. doi:10.1109/ASE56229.2023.00017

[52] Jiakun Liu, Zicheng Zhang, Xing Hu, Ferdian Thung, Shahar Maoz, Debin Gao, Eran Toch, Zhipeng Zhao, and David Lo. 2024. MiniMon: Minimizing Android Applications with Intelligent Monitoring-Based Debloating. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 206, 13 pages. doi:10.1145/3597503.3639113

[53] Yi Liu, Yun Ma, Xusheng Xiao, Tao Xie, and Xuanzhe Liu. 2023. LegoDroid: Flexible Android App Decomposition and Instant Installation. *Science China Information Sciences* 66, 4 (2023), 142103. doi:10.1007/s11432-021-3528-7

[54] LLVM Project. 2008. Clang: a C language family frontend for LLVM. https://clang.llvm.org/

[55] LLVM Project. 2024. llvm-cov. https://llvm.org/docs/CommandGuide/llvm-cov.html

[56] Abhijit Mahurkar, Xiaoguang Wang, Hang Zhang, and Binoy Ravindran. 2023. DynaCut: A Framework for Dynamic and Adaptive Program Customization. In *Proceedings of the 24th International Middleware Conference* (Bologna, Italy) *(Middleware '23)*. Association for Computing Machinery, New York, NY, USA, 275–287. doi:10.1145/3590140.3629121

[57] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15)*. Association for Computing Machinery, New York, NY, USA, 1504–1511. doi:10.1145/2695664.2695751

[58] Mohamad Mansouri, Jun Xu, and Georgios Portokalidis. 2023. Eliminating Vulnerabilities by Disabling Unwanted Functionality in Binary Programs. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia) *(ASIA CCS '23)*. Association for Computing Machinery, New York, NY, USA, 259–273. doi:10.1145/3579856.3595796

[59] mchalupa. 2024. DG. https://github.com/mchalupa/dg

[60] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 142–151. doi:10.1145/1134285.1134307

[61] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts.* Addison-Wesley Longman Publishing Co., Inc., USA.

[62] Jorge A. Navas and Ashish Gehani. 2023. OCCAM-v2: Combining Static and Dynamic Analysis for Effective and Efficient Whole-Program Specialization. *Commun. ACM* 66, 4 (2023), 40–47. doi:10.1145/3583112

[63] Chaitra Niddodi, Ashish Gehani, Tanu Malik, Sibin Mohan, and Michael Lee Rilee. 2023. IOSPReD: I/O Specialized Packaging of Reduced Datasets and Data-Intensive Applications for Efficient Reproducibility. *IEEE Access* 11 (2023), 1718–1731. doi:10.1109/ACCESS.2022.3233787

[64] Pardis Pashakhanloo, Aravind Machiry, Hyonyoung Choi, Anthony Canino, Kihong Heo, Insup Lee, and Mayur Naik. 2022. PacJam: Securing Dependencies Continuously via Package-Oriented Debloating. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 903–916. doi:10.1145/3488932.3524054

[65] Aleksandr Pilgun. 2020. Don't Trust Me, Test Me: 100% Code Coverage for a 3rd-party Android App. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. 375–384. doi:10.1109/APSEC51365.2020.00046

[66] Chris Porter, Sharjeel Khan, and Santosh Pande. 2023. Decker: Attack Surface Reduction via On-Demand Code Mapping. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 192–206. doi:10.1145/3575693.3575734

[67] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. Association for Computing Machinery, New York, NY, USA, 164–180. doi:10.1145/3385412.3386017

[68] PostgreSQL Global Development Group. 2024. PostgreSQL Application. https://github.com/postgres/postgres

[69] PostgreSQL Global Development Group. 2024. PostgreSQL: Perl-based TAP tests. https://github.com/postgres/postgres/tree/REL_12_STABLE/src/test

[70] PostgreSQL Global Development Group. 2024. PostgreSQL: Perl-based TAP tests introduction. https://github.com/postgres/postgres/blob/REL_12_STABLE/src/test/perl/README

[71] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security '19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[72] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 461–476. doi:10.1145/3372297.3417866

[73] qixin5. 2024. Benchmark of 25 programs. https://github.com/qixin5/debloating_study/tree/main/expt/debaug/benchmark

[74] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (Dallas, Texas, USA) *(FEAST '17)*. Association for Computing Machinery, New York, NY, USA, 65–70. doi:10.1145/3141235.3141242

[75] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security '18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[76] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE '17)*. Association for Computing Machinery, New York, NY, USA, 476–486. doi:10.1145/3106237.3106271

[77] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer International Publishing, Cham, 482–501.

[78] ROPgadget Project. 2024. ROPgadget. https://people.eecs.berkeley.edu/~necula/cil/merger.html

[79] Vivekananda Roy. 2020. Convergence Diagnostics for Markov Chain Monte Carlo. *Annual Review of Statistics and Its Application* 7, Volume 7, 2020 (2020), 387–412. doi:10.1146/annurev-statistics-031219-041300

[80] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 305–316. doi:10.1145/2451116.2451150

[81] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552–561. doi:10.1145/1315245.1315313

[82] Will Shackleton, Katriel Cohn-Gordon, Peter C. Rigby, Rui Abreu, James Gill, Nachiappan Nagappan, Karim Nakad, Ioannis Papagiannis, Luke Petre, Giorgi Megreli, Patrick Riggs, and James Saindon. 2023. Dead Code Removal at Meta: Automatically Deleting Millions of Lines of Code and Petabytes of Deprecated Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE '23)*. Association for Computing Machinery, New York, NY, USA, 1705–1715. doi:10.1145/3611643.3613871

[83] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 329–339. doi:10.1145/3238147.3238160

[84] slimtoolkit. 2024. DockerSlim. https://github.com/slimtoolkit/slim

[85] Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. 2024. Efficiently Trimming the Fat: Streamlining Software Dependencies with Java Reflection and Dependency Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 103, 12 pages. doi:10.1145/3597503.3639123

[86] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2023. Coverage-Based Debloating for Java Bytecode. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 38 (April 2023), 34 pages. doi:10.1145/3546948

[87] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. 2023. Automatic Specialization of Third-Party Java Dependencies. *IEEE Transactions on Software Engineering* 49, 11 (2023), 5027–5045. doi:10.1109/TSE.2023.3324950

[88] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2022. XDebloat: Towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4501–4520. doi:10.1109/TSE.2021.3120213

[89] Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. 2022. Stubbifier: Debloating Dynamic Server-Side JavaScript Applications. *Empirical Software Engineering* 27, 7 (Sept. 2022), 161. doi:10.1007/s10664-022-10195-6

[90] Raja Vallée-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations. https://api.semanticscholar.org/CorpusID:10529361

[91] Xiaoke Wang, Tao Hui, Lei Zhao, and Yueqiang Cheng. 2023. Input-Driven Dynamic Program Debloating for Code-Reuse Attack Mitigation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE '23)*. Association for Computing Machinery, New York, NY, USA, 934–946. doi:10.1145/3611643.3616274

[92] Frank Wilcoxon. 1992. Individual Comparisons by Ranking Methods. (1992), 196–202. doi:10.1007/978-1-4612-4380-9_16

[93] Ryan Williams, Tongwei Ren, Lorenzo De Carli, Long Lu, and Gillian Smith. 2021. Guided Feature Identification and Removal for Resource-constrained Firmware. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 28 (Dec. 2021), 25 pages. doi:10.1145/3487568

[94] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program debloating via stochastic optimization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 65–68. doi:10.1145/3377816.3381739

[95] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2021. Subdomain-based generality-aware debloating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 224–236. doi:10.1145/3324884.3416644

[96] Qi Xin, Qirun Zhang, and Alessandro Orso. 2023. Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 99, 13 pages. doi:10.1145/3551349.3556970

[97] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 419–430. doi:10.1145/1542476.1542523

[98] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (Santa Fe, New Mexico, USA) *(FoSER*

'10). Association for Computing Machinery, New York, NY, USA, 421–426. doi:10.1145/1882362.1882448

[99] Renjun Ye, Liang Liu, Simin Hu, Fangzhou Zhu, Jingxiu Yang, and Feng Wang. 2022. JSLIM: Reducing the Known Vulnerabilities of JavaScript Application by Debloating. In *Emerging Information Security and Applications*, Weizhi Meng and Sokratis K. Katsikas (Eds.). Springer International Publishing, Cham, 128–143.

[100] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 253–267.

[101] Huaifeng Zhang, Mohannad Alhanahnah, Fahmi Abdulqadir Ahmed, Dyako Fatih, Philipp Leitner, and Ahmed Ali-Eldin. 2024. Machine Learning Systems are Bloated and Vulnerable. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 1, Article 6 (Feb. 2024), 30 pages. doi:10.1145/3639032

[102] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. 2022. One size does not fit all: security hardening of MIPS embedded systems via static binary debloating for shared libraries. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 255–270. doi:10.1145/3503222.3507768

[103] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. 2022. Debloating Address Sanitizer. In *31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association, Boston, MA, 4345–4363. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen

Received 10 December 2024; revised 2 April 2025; accepted 8 December 2025