Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating

Qi Xin
Wuhan University
China
qxin@whu.edu.cn

Qirun Zhang
Georgia Institute of Technology
USA
qrzhang@gatech.edu

Alessandro Orso
Georgia Institute of Technology
USA
orso@cc.gatech.edu

ABSTRACT

Existing approaches for program debloating often use a usage profile, typically provided as a set of inputs, for identifying the features of a program to be preserved. Specifically, given a program and a set of inputs, these techniques produce a reduced program that behaves correctly for these inputs. Focusing only on *reduction*, however, would typically result in programs that are overfitted to the inputs used for debloating. For this reason, another important factor to consider in the context of debloating is *generality*, which measures the extent to which a debloated program behaves correctly also for inputs that were not in the initial usage profile. Unfortunately, most evaluations of existing debloating approaches only consider reduction, thus providing partial information on the effectiveness of these approaches. To address this limitation, we perform an empirical evaluation of the reduction and generality of 4 debloating techniques, 3 state-of-the-art ones, and a baseline, on a set of 25 programs and different sets of inputs for these programs. Our results show that these approaches can indeed produce programs that are overfitted to the inputs used and have low generality. Based on these results, we also propose two new augmentation approaches and evaluate their effectiveness. The results of this additional evaluation show that these two approaches can help improve program generality without significantly affecting size reduction. Finally, because different approaches have different strengths and weaknesses, we also provide guidelines to help users choose the most suitable approach based on their specific needs and context.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**;

KEYWORDS

Software debloating, program reduction, generality

ACM Reference Format:
Qi Xin, Qirun Zhang, and Alessandro Orso. 2022. Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556970

1 INTRODUCTION

Programs in today's world are increasingly complex. In particular, they contain an abundance of features that aim to provide service to a wide range of users. In practice, however, only a small fraction of the implemented features are typically accessed, leaving a significant amount of them unneeded [34], which leads to code bloat [12, 46, 76]. Code bloat is pervasive [24, 38, 51] and can severely harm program performance [13, 74, 76] and security [9, 63]. To mitigate this problem, many program debloating techniques have emerged that aim to identify and eliminate a program's unneeded features and produce a reduced version of it (e.g., [33, 49, 50, 64, 69, 70]).

Specifying the desired features of a program is in general a hard problem. As with the mainstream approaches from other domains, such as program repair [41], existing program debloating approaches often use a usage profile, typically provided in the form of a set of inputs, as a proxy for a specification [15, 33, 49, 67, 70]. We refer to these approaches as input-based approaches. Given a program and a set of inputs, an *input-based* approach performs code pruning to produce a reduced program that behaves correctly for the given inputs. Unfortunately, because a set of inputs can typically provide only an under-approximation of a program's expected behaviors, an input-based approach is likely to produce a program that is overfitted to the inputs used and easily fails for other, unobserved inputs. For example, with two inputs "*foo*" and "*-p foo/bar*", the input-based technique Chisel produces an overly reduced version of *mkdir* (a utility for creating directories) with the loop that identifies parent directories (Figure 1, line 1) eliminated. Chisel deemed the loop to be unnecessary, as its body was exercised only once. Consider now an additional input "*-p foo/bar/baz*". Although similar in nature to "*-p foo/bar*", the debloated program would not handle this input correctly, as the removal of the loop prevents it from processing the second slash ("/") and creating the needed directories.

The issue is that this type of input-based approaches are usually not really meant to produce a debloated program that merely handles the provided inputs I, but rather (and intuitively) a program that can handle the features characterized by I. For example, a negative integer input in I may indicate the need to handle all negative integer inputs, or at least all the negative integer inputs within a given range. If that were not the case, and I could fully characterize the required functionality, one could simply synthesize a program

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556970

that uses a map to return, for any observed input, the expected output—a program that would be useless for any unobserved inputs.

In general, a debloating approach must therefore account for, in addition to program reduction, program *generality*, that is the ability to correctly handle unobserved inputs. Based on what we discussed above, this is especially true for those inputs that are possibly related to I, in the sense that they exercise similar features. Unfortunately, there is an inherent tension between reduction and generality [70], which means that debloating approaches must try to achieve a good tradeoff between these two conflicting factors [70, 71].

Early debloating approaches [33, 58, 68] tended to pursue aggressive size reduction without considering generality. Two more recent approaches, Debop [70] and Razor [49], began to account for generality but did not properly measure it: Debop evaluates program generality based on the provided inputs, rather than unobserved ones, whereas Razor uses a weak (crash-based) oracle to determine a program's behavioral correctness and to then evaluate generality. Also, none of these approaches measured the tradeoff between reduction and generality. For these reasons, it remains largely unknown how the existing input-based approaches perform in terms of addressing both reduction and generality for debloating.

To fill this gap, we conducted a study to evaluate how input-based debloating approaches perform in terms of reduction, generality, and their tradeoff. We investigated three state-of-the-art approaches designed for C programs (Chisel, Debop, and Razor), along with a baseline that we developed (Cov), which performs debloating based on code coverage. We applied these approaches to two benchmarks containing a total of 25 programs, using different sets of inputs for debloating (debloating inputs) and for generality evaluation (testing inputs). To evaluate a debloating approach, we used two types of reductions, based on size and attack surface, and two types of generality, based on correctness (*c-generality*) and robustness (*r-generality*). We quantified the tradeoffs between different types of reduction and generality in terms of F-score measures.

The results of our experiment show that Chisel and Debop, which do not account for a program's ability to handle unobserved inputs, may produce debloated programs with low c-generality. For one of the benchmarks, for instance, the debloated program behaved correctly for less than 44% of all the testing inputs and less than 56% of the testing inputs "related" to the debloating inputs. Conversely, Razor, by performing coverage-based reduction and heuristic-based augmentation (which infers and preserves unexercised-but-related code), can produce programs with increased c-generality (behaving correctly for 51% of the testing inputs and 69% of the "related" testing inputs) without affecting reduction in a significant way. Unfortunately, however, the debloated programs produced by Razor are also ultimately not robust—they exhibit crashes or non-termination behaviors for many testing inputs.

Based on our findings, we developed two augmentation-based approaches, CovF and CovA, that aim to improve the r-generality and c-generality of debloated programs while achieving good reduction-generality tradeoffs. The two approaches are built upon Cov, which, according to our results, is considerably more efficient than the other approaches that perform source code reduction and can often achieve similar tradeoffs. CovF and CovA perform fuzz-based and analysis-based code augmentation, respectively, to identify complementary, related code and preserve such code in the debloated program.

Specifically, given a program p and a set of inputs I, both CovF and CovA first invoke Cov to remove from p code that is not exercised by I, which results in program p_{cov}. Then, for augmentation, CovF leverages fuzzing to generate inputs that can expose p_{cov}'s robustness issues and adds back code from p that is exercised by these inputs to eliminate the identified issues. CovA, conversely, does not rely on specific inputs for augmentation. Rather, it augments p_{cov} by leveraging information computed using static and dynamic analyses, namely, static program dependencies, execution frequency, and *coverage flexibility*, which measures differences in code coverage to identify parts of p that are related to p_{cov}. Intuitively, by focusing on either (1) handling inputs leading to p_{cov}'s crashes or non-termination (CovF), or (2) keeping code related to p_{cov} (CovA), these approaches are more likely to produce programs with enhanced robustness and correctness.

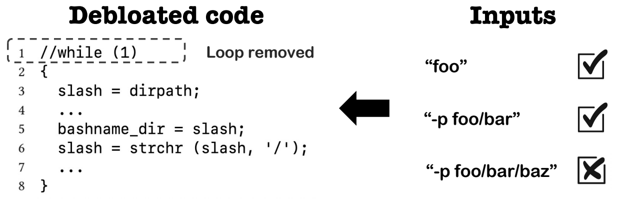
We evaluated CovF and CovA on the same benchmarks used for the state-of-the-art approaches and found that they can indeed produce programs with enhanced generality. Most importantly, they can achieve these improvements without significantly increasing program size, and thus obtain better tradeoffs between generality and size reduction than Cov and other approaches performing source code reduction.

As noted by Xin et al. [70], debloating in a realistic scenario often involves multiple, conflicting goals. In our study, in particular, we considered four specific goals and is expected to produce programs with high size reduction, attack surface reduction, c-generality, and r-generality. We found that each of the approaches we considered has its own strengths and weaknesses, and no approach is the winner in all cases. Based on the analysis of these approaches, we therefore also provide guidelines that can help users choose the most suitable approach based on their specific needs.

In summary, using a set of representative inputs that reflects which features of a program should be preserved is a practical way of achieving program debloating, as long as the right techniques are used and relevant tradeoffs are considered. We believe that our study, along with the approaches we developed, can provide lessons learnt and resources for both current users of debloating techniques as well as for future research on input-based debloating.

The main contributions of this paper are:

- A study that systematically assesses and compares six input-based debloating approaches based on different types of code reduction, generality, and their tradeoffs. Our result shows that existing input-based debloating approaches tend to produce programs that are overfitted to the inputs used to guide debloating and have low generality.
- Two novel augmentation-based approaches, CovF and CovA, that show how augmentation can effectively mitigate the overfitting problem in input-based debloating techniques and can be used to achieve good tradeoffs between reduction and generality.
- A discussion on the different strengths and weaknesses of input-based debloating approaches that provides guidance on how to choose a debloating approach that best suits specific goals and can guide future research in this area.



```
//while (1)
{
  slash = dirpath;
  ...
  bashname_dir = slash;
  slash = strchr (slash, '/');
  ...
}
```

Figure 1: An example of debloated code produced by Chisel.

- An artifact that contains our experimental data and results, together with the prototype implementation of the two approaches we developed (CovF and CovA), and is available at https://github.com/qixin5/debloating_study.

2 BACKGROUND

In this section, we provide the definitions of input-based debloating. We also define different debloating measures in terms of reduction, generality, and their tradeoffs.

2.1 Input-Based Debloating

Let p be a deterministic program and I be a set of inputs for p. We use $p(i)$ to denote the result of running p with input i. Given p and I, input-based debloating is the process of removing code from p to produce a reduced program p' such that, for each $i \in I$, we have $p(i) = p'(i)$.

2.2 Debloating Measures

2.2.1 Reduction. Given a program p and its debloated version p', we use *reduction* to indicate the amount of code that has been removed. As with previous work [33, 49, 70], we consider two types of reduction: (1) *size reduction* and (2) *attack surface reduction.* Given p and p', size reduction *sred* is computed as

$$sred(p, p') = \frac{size(p) - size(p')}{size(p)},$$

where $size(\cdot)$ measures the program's size. We consider two types of sizes: (a) number of statements and (b) number of bytes of the executable memory region. These size measurements were used in the evaluation of Chisel [33] and Razor [49]. In our study, we use these tools' size measuring utilities to compute (a) and (b).

Similarly, given p and p', attack surface reduction *ared* is computed as

$$ared(p, p') = \frac{asurf(p) - asurf(p')}{asurf(p)},$$

where $asurf(\cdot)$ measures a program's attack surface in terms of number of *Return-Oriented-Programming* (ROP) gadgets it contains. An ROP gadget [63] is a sequence of machine instructions that (typically) ends with a return instruction. An attacker can take advantage of a vulnerability (e.g., stack overflow) to overwrite a gadget's return address to divert the control flow and execute malicious code [63]. ROP gadgets are typically used for measuring attack surface [33, 49, 64, 70]. As in [33], we use the ROPgadget tool [59] to count ROP gadgets.

2.2.2 Generality. Given a program p and its debloated version p' generated based on a set of inputs I, we define two types of *generality*, correctness-based (*c-generality*) and robustness-based (*r-generality*), to measure the extent to which p' could behave correctly and robustly, respectively, for inputs not in I. To quantify c-generality, we gather a different set of inputs I' ($I' \cap I = \emptyset$) and measure the fraction of inputs of I' for which p' behaves correctly. Formally, for c-generality *cgen*, we have

$$cgen(p', I') = \frac{\sum_{i' \in I'} p'(i') = p(i')}{|I'|},$$

where $p(\cdot)$ and $p'(\cdot)$ denote the outputs produced by p and p', and $|I'|$ denotes the number of inputs in I'. We investigate two types of c-generality evaluated based on different types of I'. When I' represents a universal set of inputs, we refer to the computed generality as *all-input-based* c-generality. When I' is a set comprising only inputs that are related to I (e.g., in exercising similar features), we refer to the generality as *related-input-based* c-generality. Note we do not use the generality measure from [70], as that measure is computed based on a single set of inputs; it is designed to guide the optimization process and is not suitable for evaluating debloated programs.

In addition to c-generality, we also investigated r-generality, which measures a debloated program's resilience to unobserved inputs and reflects the program's reliability. To quantify r-generality, we gather a different set of inputs I' ($I' \cap I = \emptyset$) representing the usage of p in invalid cases and measure the fraction of inputs of I' for which p' does not exhibit a crash and terminates. Formally, for r-generality *rgen*, we have

$$rgen(p', I') = \frac{\sum_{i' \in I'} robust(p'(i'))}{|I'|},$$

where $robust(\cdot)$ indicates whether the program execution terminates with no crash (e.g., no segmentation fault). In our study, we obtained I' by producing a fuzzed version of I.

2.2.3 Reduction-generality tradeoff. To quantify the tradeoff between reduction *red* (either size-based or attack-surface-based) and generality *gen* (either correctness-based or robustness-based), we use the harmonic mean, or F-score, f computed as

$$f(red, gen) = \frac{2 \cdot red \cdot gen}{red + gen}.$$

A high F-score indicates both high reduction and high generality. When an approach pursues aggressive reduction or generality at the cost of the other, it will produce a program with a low F-score. In the extreme case, where none or all of the program's code is pruned, reduction or generality (but not both) is 0, and the F-score is 0.

3 DEBLOATING APPROACHES CONSIDERED

In this section, we discuss the debloating approaches we considered. All these approaches perform input-based debloating. Among these approaches, Razor is a *binary-based* approach that modifies a program's binary. The other approaches reduce program's source code and are *source-based.*

3.1 Previous Input-Based Approaches: Chisel, Debop, and Razor

Chisel is a reduction-oriented approach based on a reinforcement-learning-guided delta-debugging-based algorithm for debloating. Debop is a multi-objective approach that takes three factors (size reduction, attack surface reduction, and generality) into consideration, and performs stochastic optimization to produce a debloated program, aiming to achieve an optimal tradeoff between these factors. Razor performs binary-level debloating based on tracing, heuristic-based augmentation, and binary-rewriting. Specifically, it uses four heuristics with increasing aggressiveness in identifying feature-related code. More details of these approaches can be found in [33], [70], and [49].

3.2 Cov: Coverage-Based Debloating

Cov is an approach we developed that performs coverage-based debloating. Given a program p and a set of inputs I, Cov instruments p and executes it against I to identify all the statements exercised by I. To produce a debloated program, it removes statements that are not exercised together with those that are exercised but are actually unneeded (e.g., a variable declaration that is never used). A prototype of Cov we implemented uses *llvm-cov* [42] and *gcov* [27] for instrumentation. It relies on Clang [21] to build the program's abstract syntax tree (AST), and traverses the AST to identify all the statements. By analyzing the coverage report generated by the instrumentation tools, Cov finds statements not exercised in the execution and deletes them. It additionally deletes, through dependency analysis, unnecessary statements, including (1) "dangling" instructions with empty bodies and side-effect-free conditions (e.g., *if (x==0) {}*), (2) unused variable declarations, and (3) unused label statements.

3.3 CovF: Coverage-Based Debloating with Fuzz-Based Augmentation

A coverage-based approach, by eliminating the unexercised code, is likely to produce a program that is overly reduced. Such a program is not robust and can easily crash or hang (i.e., cannot terminate) for an unobserved input. To address this problem, we developed CovF, an approach that performs fuzz-based augmentation to produce debloated programs with enhanced r-generality (robustness). Given a program p and a set of inputs I, CovF first invokes Cov to produce a reduced program p_{cov}. Next, it performs fuzzing to produce, based on I, a set of inputs I' that are "robustness-related" and can make p_{cov} crash or hang. In its current implementation, CovF produces I' by first applying the blackbox fuzzer Radamsa [53] to I, to obtain an initial set of fuzzed inputs I'_{init}, and then running p_{cov} against I'_{init}, to identify $I' \in I'_{init}$ whose inputs make p_{cov} crash or hang. The reason why p_{cov} crashes or hangs for an unobserved input i' could be the lack of needed code in p for correctly processing i'. Therefore, in its final step, CovF produces a debloated program by preserving statements in p that are exercised by I' and I and removing all others. Note that it is possible that the original p exhibits robustness issues for an unobserved input i'. In that case, the robustness issues were not caused by debloating, and CovF, by performing augmentation, would not help resolve the problem.

3.4 CovA: Coverage-Based Debloating with Analysis-Based Augmentation

A fuzz-based approach focusing on robustness enhancement is prone to producing many irregular inputs that exercise a program's non-core logic and fail to exercise the different features of the program. Therefore, this kind of augmentation cannot easily lead to a significant improvement of c-generality. For example, it would not be easy to obtain, by fuzzing input i: *"-m 777 testdir"* for *mkdir*, another input i': *"-m a=rwx testdir"*, which is related to i because it exercise a related feature f—making a directory with permission expressed symbolically instead of numerically. Without finding valid inputs such as i', a fuzz-based approach cannot augment a debloated program to preserve f.

In order to achieve better improvements of c-generality, we developed another analysis-based approach: CovA. CovA performs static and dynamic analyses to infer related code for augmentation based on dependency relationship among functions and the program's execution traces obtained with the debloating inputs. Unlike CovF, which relies on finding specific inputs for augmentation, CovA directly identifies functions that may be feature-related and preserves them entirely in the final debloated program. By targeting functions for augmentation, CovA reduces the likelihood of producing programs that are syntactically or semantically invalid (e.g., programs that contain a loop but do not contain a critical `break` statement). CovA computes, for each function, an augmentation score approximating how much related it is to the desired features and how relevant it is for augmentation based on the functions' dependency relationship, their execution frequency, and the variance of coverage obtained from its analyses. Intuitively, a function that is higher-level (in terms of dependencies) and is exercised by more inputs in I is considered to be more feature-related. Moreover, if the coverage of the function varies when exercised by more inputs, it is considered to need additional augmentation to handle unobserved inputs that may exercise new code. Based on the computed scores, CovA ranks the functions and selects the top-K for augmentation.

More formally, given a program p, a set of debloating inputs I, and an augmentation threshold *topk*, CovA first obtains all the functions defined in p. It then performs analyses to compute, for each function f, three scores: specificity *spec*, frequency $freq$, and flexibility $flex$, and saves these in a map. CovA normalizes these scores based on all the functions. Given these scores, CovA computes the average of the normalized scores as the augmentation score $aug(f)$:

$$aug(f) = \frac{spec_n(f) + freq_n(f) + flex_n(f)}{3},$$

where $spec_n$, $freq_n$, and $flex_n$ are the normalized results of $spec$, $freq$, and $flex$, respectively. Finally, CovA sorts the functions based on their augmentation score and produces a debloated program by preserving (1) statements of p exercised by I plus (2) all the statements of the top-K functions. It also removes unneeded code using an approach similar to the one described in Section 3.2. We next discuss how to compute *specificity*, *frequency*, and *flexibility* in turn.

Specificity. CovA performs static analysis, more specifically dependency analysis, to compute *specificity*, which quantifies the extent to which a function is related to program-specific features. Since CovA’s augmentation targets feature-related code, a function with higher specificity is of more interest. Conversely, a function that has low specificity is likely to behave more like a utility (library) and serve a more general purpose, so CovA considers it low priority for augmentation. To compute specificity, CovA builds a function call graph and computes, for each function f, a utilityhood based on the fan-in and fan-out of f, that is, on the number of functions on which f depends and the number of functions that depend on f. Specifically, CovA uses the metric defined in [32] to compute the utilityhood of a function f and computes specificity $spec$ as $1 - util$:

$$spec(f) = 1 - util(f) = 1 - \frac{fanin(f)}{N} \times \frac{log(\frac{N}{fanout(f)+1})}{log(N)},$$

where N is the total number of functions and $fanin(f)$ and $fanout(f)$ are the fan-in and fan-out values for f. $fanin(f)$ is the number of functions that call f in the graph, and $fanout(f)$ is the number of functions that f calls. Because, as explained in [32], the values of fan-in and fan-out vary between 0 to $|N| - 1$ (self-dependencies are ignored), the value of $util(f)$ is between 0 and 1. In the definition of $util(f)$, a function f that has a larger number of callers (larger value of $fanin(f)$) and a smaller number of callees (smaller value of $fanout(f)$) is considered more self-contained and thus has a higher utilityhood. Because such a function f behaves more like a utility, it has a lower specificity.

Frequency. CovA executes the program with each input in I and computes, for each function f, the *frequency* $freq$ as

$$freq(f, I) = \frac{N_f}{N_I},$$

where N_f is the number of inputs that exercise f, and N_I is the total number of inputs in I. Intuitively, a function that is exercised by more inputs in I is more likely to be related to the features exercised by I.

Flexibility. CovA also analyzes the execution of a function f to compute its *flexibility*. For each f that is exercised by I, CovA tracks, for each input $i \in I$, the set of all the statements i exercises, $S(f, i)$. If f is not exercised by i, $S(f, i) = \emptyset$. Next, it counts the unique sets of statements in f exercised by the inputs in I, $c(f, I)$. As an example, assume that I contains three inputs, i_0, i_1, and i_2, and the sets of statements in f exercised by these inputs are $S(f, i_0) = \{s_0, s_1\}$, $S(f, i_1) = \{s_0\}$, and $S(f, i_2) = \{s_0, s_1\}$, where s_0, s_1, and s_2 are three statements. In this case, $c(f, I) = 2$, as there are two unique sets of statements. Using $c(f, I)$, CovA computes the flexibility $flex$ of f as

$$flex(f, I) = \frac{c(f, I)}{N_I},$$

where N_I is the number of inputs in I. The fact that a function has lower flexibility implies that its execution is less likely to change for unobserved inputs, so CovA considers it low priority for augmentation.

It is worth noting that, although CovF and CovA are developed based on Cov, their augmentation methods can be adapted and used by other approaches.

Table 1: Benchmark programs and inputs.

Bench	Program	LOC	#Func	#Stmt	#TotalIn	#MinIn	#MaxIn	AvgIn
Util	bzip2-1.0.5	11,782	97	6,154	59	2	13	5.9
	chown-8.2	7,081	122	3,765	111	3	20	11.1
	date-8.21	9,695	78	4,228	174	5	45	17.4
	grep-2.19	22,706	315	10,977	145	4	27	14.5
	gzip-1.2.4	8,694	91	4,049	81	3	12	8.1
	mkdir-5.2.1	5,056	43	1,804	50	2	12	5.0
	rm-8.4	7,200	135	3,835	84	3	16	8.4
	sort-8.16	14,264	233	7,805	117	5	32	11.7
	tar-1.14	30,477	473	13,995	84	3	16	8.4
	uniq-8.16	7,020	65	2,086	72	2	12	7.2
	Total	123,975	1,652	58,698	977	32	205	97.7
LSIR	bash-2.05	58,319	1,003	27,646	1,061	1,061	1,061	1,061
	flex-2.5.4	15,518	162	6,704	670	670	670	670
	grep-2.4.2	16,203	131	8,437	806	806	806	806
	gzip-1.3	8,882	97	4,287	213	213	213	213
	make-3.79	26,118	248	12,901	1,832	1,832	1,832	1,832
	sed-4.1.5	18,866	247	9,179	370	370	370	370
	space	8,215	136	4,376	13,549	13,549	13,549	13,549
	vim-5.8	136,531	1,699	66,080	975	975	975	975
	Total	288,652	3,723	139,610	19,476	19,476	19,476	19,476
SSIR	printtokens2	824	19	341	4,058	4,058	4,058	4,058
	printtokens	1,069	18	396	4,073	4,073	4,073	4,073
	replace	938	21	416	5,542	5,542	5,542	5,542
	schedule2	604	16	238	2,710	2,710	2,710	2,710
	schedule	537	18	211	2,650	2,650	2,650	2,650
	tcas	382	9	162	1,608	1,608	1,608	1,608
	totinfo	586	7	265	1,052	1,052	1,052	1,052
	Total	4,940	108	2,029	21,693	21,693	21,693	21,693

4 EVALUATION

We investigated the following four research questions:

- **RQ1**: How do the approaches considered compare in terms of reduction, c-generality, and their tradeoff?
- **RQ2**: How do the approaches considered compare in terms of reduction, r-generality, and their tradeoff?
- **RQ3**: How do the approaches considered perform when an increasing amount of inputs is used for debloating?
- **RQ4**: How efficient are the approaches?

4.1 Tool Implementation

We used the implementation of Chisel [19], Debop [22], and Razor [56] provided by their authors and implemented the other approaches based on their description in Section 3.

4.2 Benchmarks

4.2.1 Benchmark programs. In our evaluations, we used two sets of programs: ten utility programs (*Util*) and 15 programs from the SIR benchmark [2] (*SIR*). We used the ten utilities because they were used in the evaluation of previous debloating approaches [33, 49, 70]. For these programs, we used their all-in-one-file versions provided in Chisel’s benchmark [20]. An all-in-one-file program contains a single C file that combines all of the original C source files in the project. In addition to the utilities, we also used a set of 15 programs, which are all the C programs provided in the SIR benchmark. They represent a range of programs of different types including a lexical analyzer (*printtokens*), a utility (e.g., *make*), a Unix shell (*bash*), and a text editor (*vim*). We also chose these benchmarks because they have a large number (from hundreds to thousands) of tests associated. Having a large number of tests allows us to effectively evaluate a debloated program’s generality and to perform a thorough investigation of the effectiveness of debloating with an increasing amount of inputs. For the SIR programs, we used the

CIL merger [1] to obtain all-in-one-file versions for debloating for them as well.

Table 1 presents the 25 programs considered in terms of the specific benchmark to which they belong (*Bench*), name (*Program*), size in lines of code (*LOC*), and number of functions (*#Func*) and statements (*#Stmt*). The SIR benchmark contains seven programs (*SSIR*) whose sizes are considerably smaller than those of the other eight ones (*LSIR*). Although we compared all approaches on all programs, we focus on analyzing and presenting the results for the utilities and the large SIR programs (LSIR), as debloating is typically not applied to small programs for reduction. We will nevertheless provide a summary of the results for SSIR in Sections 4.4.4 and 4.5.

Because current implementations of the source-based approaches (i.e., Chisel, Debop, and Cov) only work as intended on programs whose source code is merged (by CIL), we are currently unable to investigate their effectiveness on very large programs without investing non-trivial effort in either improving their implementation or making CIL (currently not maintained) work for such programs. We therefore leave this kind of evaluation for future work.

4.2.2 Inputs. For each program, we used different sets of inputs for debloating and for generality evaluation. Because the latter needs a large number of inputs, we did not use for the utilities the original set of inputs associated with the programs, which is small in size and was artificially created. Instead, we searched for user inputs online from different websites. Specifically, we collected ten sets of inputs for each utility via Google Search [29], using a program's name plus the word "usage" as the search query. Each set of inputs is from a specific website demonstrating the usage of the program from a real user. A list of all the websites from which we extracted inputs is available at [3]. In cases in which a file was needed for the user input, and one was not provided, we generated a file with random content. Table 1 presents the statistics of the inputs in terms of *#TotalIn*, *#MinIn*, *#MaxIn*, and *#AvgIn*: *#TotalIn* is the total number of inputs from all input sets; *#MinIn* is the number of inputs in the set with the smallest number of inputs among all the sets of inputs collected, S_{min}; *#MaxIn* is the number of inputs in the set with the largest number of inputs, S_{max}; and *#AvgIn* is the average number of inputs among all the input sets. For the SIR programs, we used the inputs associated with the programs provided in [2]. Table 1 presents the total number of inputs (*#TotalIn*) for these programs. Because each SIR program has a single set of inputs associated with it, the four measures have the same value.

For the utilities, we applied each debloating approach considered using each of the ten input sets, while we used the other input sets for c-generality evaluation. For each SIR program, we divided the set of inputs associated with a program, I_u, into two subsets, I_d and I_e, which we used for debloating and for evaluating c-generality, respectively. We created I_d by randomly selecting ten inputs from I_u, so as to make its size approximately equal to the number of inputs (97.7/10 = 9.7) used for debloating Util programs and mimic real usage. We also created three additional I_d sets comprising 10%, 20%, and 30% of the inputs from I_u, and obtained the corresponding I_e sets, to investigate the relationship between input size and debloating effectiveness.

To evaluate related-input-based c-generality, we used different approaches to identify related inputs for different programs. For programs accepting command-line inputs with options (e.g., *bzip2*), we considered two inputs to be related if they used the same set of options, following an approach similar to the one performed in the evaluation of Razor [49]. For *bash* and *vim*, the provided inputs were already tagged with a label indicating the functionality they exercised (e.g., "arith" for the arithmetic functionality). For these inputs, therefore, we used the tags to determine relatedness. For all other programs, which consisted of the SSIR programs, we considered all inputs to be related.

To evaluate r-generality, we used Radamsa [53] and AFL [4] to produce, based on the debloating inputs, a set of fuzzed inputs I'. (Note that we did not use the fuzzed inputs used in the evaluation of r-generality in CovF's debloating process.) We produced I' with Radamsa by selecting at most ten inputs within I and generating 100 fuzzed versions for each selected input. To generate I' with AFL, we ran it for 30 minutes based on the debloated program and each of the previously selected inputs. The time threshold we used for fuzzing a debloated program was five hours. Because AFL's fuzzing process is expensive, we selected ten programs from the Util and LSIR benchmarks (five each) for this experiment. We also excluded Razor, as we discovered that AFL did not correctly fuzz Razor's debloated binaries in its QEMU mode [4]. Finally, we performed input sampling, as input fuzzing and testing for all programs over all inputs would have been too time consuming. We leave as future work a more extensive evaluation of program robustness.

4.2.3 Oracles. We need oracles to evaluate a program's behavioral correctness and its robustness. Correctness-based oracles check whether a debloated program behaves correctly for a test input by comparing the output of the debloated program against that of the original program. For the utilities, we considered as output a program's exit value, any printed message, and any file generated. For files, the oracles check the file content (and ownership and permission when needed). It is worth noting that we designed the oracles based specifically on the programs and inputs considered, and that *the oracles are only needed for the evaluation.* For the SIR programs, we used the oracles provided in the original tests to identify program outputs for comparison. In addition, for robustness evaluation, we developed an oracle that checks whether a program crashes or does not terminate. To determine whether a program crashed, the oracle checks for a list of exit codes (131–136 and 139) that represent abnormal termination, including segmentation fault, floating point exception, and bus error. To determine non-termination, we ran a debloated program against a fuzzed input for at most ten seconds, which was long enough for all the correct executions we considered to terminate, and the oracle checked whether there was a timeout.

4.3 Setup

4.3.1 Evaluation Method. For each utility p, we obtained ten sets of inputs and used debloating approach T to produce a debloated program p' for p based on each set of inputs. In this way, we obtained ten debloated programs for each T. We measured the reduction, generality, and tradeoff scores for these programs, and computed their average as the scores achieved by T. To measure c-generality, we executed p' against each set of inputs not used for debloating (for a total of nine sets) and computed the average ratio of inputs for which p' behaved correctly. For each SIR program p, we used a debloating approach T to produce a debloated program p' based on I_d, the debloating inputs, and evaluated the c-generality of p' based on I_e, the testing inputs. We used the fuzzed inputs and oracles described in Sections 4.2.2 and 4.2.3 to evaluate the r-generality (robustness) of p'.

4.3.2 Parameters. We used their default parameter values to run Chisel, Debop, and Razor. Because Chisel and Debop are computationally expensive, we ran both approaches for a maximum of six hours to produce a program based on each set of inputs. Razor performs augmentation in four levels, and we experimented with all these levels. For CovF, we selected a maximum of 100 inputs within the input set and used Radamsa to produce 10 fuzzed inputs for each selected input for debloating. For CovA, which selects the top-K functions for augmentation, we experimented with 15 values of K (from 1% to 5% and from 10% to 50%) and compared the different results. To measure generality, because a debloated program's execution against an input may not terminate, we used a timeout of ten seconds.

4.3.3 Platform. We performed our evaluation on a machine running Ubuntu-18.04, with 260GB of RAM and 32 AMD-Opteron 1.4GHz processors. The machine time necessary to obtain all the debloated programs for all experiments is over 284 hours (11 days). It then took many extra hours for evaluating the generality of the debloated programs.

4.4 RQ1: Comparison in terms of reduction, correctness-based generality, and tradeoff

Table 2 shows the (average) c-generality (*CGen*), reduction (*Red*), and tradeoff (*Tradeoffs*) scores for all the debloating approaches considered. The columns in the table show the benchmark id (*Benchmark*), the approach (*Appr*), related-input-based c-generality (*RelCGen*), all-input-based c-generality (*AllCGen*), statement-based size reduction (*SRed*), memory-based size reduction (*MRed*), attack surface reduction (*ARed*), and six types of tradeoff F-scores computed based on all combinations of the c-generality and reduction scores. For example, *RelSF* is computed based on *RelCGen* and *SRed*.

The table shows results for three sets of programs: the utilities (*Util*), the large SIR programs (*LSIR*), and a reduced set of large SIR programs (*R-LSIR*). The reason to have R-LSIR is that, for three LSIR programs, *bash*, *make*, and *vim*, we detected errors in the tracing phase of Razor's debloating process. We therefore compared Razor with other approaches on R-LSIR, which does not contain the three programs for which Razor behaved incorrectly. For the R-LSIR programs, we show the scores in parentheses.

For Razor and CovA, we experimented with different augmentation thresholds to produce a range of debloated programs with different reduction-generality tradeoffs. In the table, we show the best performance of these tools by reporting, for each program, thresholds that lead to the best tradeoffs, measured by the highest F-scores (*AllMF* for Razor and *AllSF* for CovA). Note, however, that Razor and CovA are both capable of producing debloated programs with higher and lower c-generality and reduction scores than what is presented in Table 2. We leave as future work the automatic identification of the best threshold. Below we discuss the comparison of the approaches.

4.4.1 ***Comparison of Chisel, Debop, and Cov.*** We first compare Chisel and Debop, the two existing input-based approaches, and the baseline Cov. According to Table 2, Chisel's and Debop's c-generality scores are relatively low and are less than 0.6. These results confirm that the two approaches are prone to producing overfitting programs, which do not correctly handle many (related) unseen inputs. As shown in the *Tradeoffs* columns of the table, their size-based F-scores (*RelSF*, *RelMF*, *AllSF*, and *AllMF*) are not higher than those for Cov. This shows that Chisel and Debop are not better than Cov in obtaining good tradeoffs between size reduction and c-generality.

As we discussed in Section 3.1, Chisel is based on delta debugging. In principle, delta debugging is more aggressive than a coverage-based approach for code pruning. In practice, however, Chisel pruned less code than Cov. This is mainly because its delta debugging process is computationally expensive. Within the six-hour time limit that we considered, Chisel was unable to prune much unneeded code for large programs. Debop is based on stochastic optimization and relies on a coverage-based approach to produce a reduced program p_c. However, because Debop does not augment p_c to increase c-generality; it rather only reduces p_c with the goal of achieving a better tradeoff by increasing size reduction. This explains why Debop can almost always produce a debloated program with a higher size reduction and lower c-generality than Cov. Its size-based F-scores are often lower than, but very close to, those of Cov.

For the utilities, Chisel and Debop removed more ROP gadgets than Cov, which results in a slightly higher *RelAF* and *AllAF* scores. This result implies that Cov, which completely preserves executed statements, can produce programs retaining more gadgets that can potentially be exploited for attacks. Approaches such as Chisel and Debop, which delete single statements in the execution path, are more effective at reducing gadgets. Another reason why Debop eliminated more gadgets is that it explicitly uses gadget reduction as one of its optimization objectives. For LSIR programs, Debop also removed more ROP gadgets than Cov and obtained higher F-scores. For these larger programs, Chisel's gadget reduction ability is considerably weaker, for the reasons explained above, which leads to lower reduction and, ultimately, lower F-scores.

Existing approaches Chisel and Debop tend to produce debloated programs with low c-generality. They also do not outperform Cov in finding a better tradeoff between size reduction and c-generality. Finally, Debop is more effective at reducing attack surface.

4.4.2 ***Comparison of CovF, CovA, and others.*** In this section, we focus on comparing CovF and CovA, the two augmentation-based approaches we developed, with other source-based approaches to investigate their effectiveness. As described in Section 3, CovF and CovA essentially augment p_{cov}, the debloated program produced by Cov, by adding to it code from the original program. According to Table 2, CovF's and CovA's c-generality scores are higher than those for Cov, which shows that their augmentation approaches can result in better c-generality. In particular, CovA's

Table 2: Reduction, correctness-based generality, and tradeoff scores (▲: highest score for source-based approaches; ⋆: highest score for all approaches).

Benchmark	Appr	CGen		Red			Tradeoffs					
		RelCGen	AllCGen	SRed	MRed	ARed	RelSF	RelMF	RelAF	AllSF	AllMF	AllAF
Util	Chisel	0.55	0.43	0.71	0.65	0.36	0.59	0.57	0.41	0.51	0.49	0.36
	Debop	0.54	0.40	▲0.76	▲0.67	▲0.39	0.62	0.59	▲0.43	0.51	0.49	▲0.37
	Cov	0.57	0.43	0.74	0.65	0.33	0.63	0.59	0.39	0.53	0.50	0.34
	CovF	0.61	0.46	0.71	0.62	0.31	0.65	0.61	0.39	0.54	0.51	0.34
	CovA	▲⋆0.73	▲⋆0.60	0.65	0.57	0.26	▲0.68	▲0.63	0.38	▲0.62	▲⋆0.58	0.36
	Razor	0.69	0.51	-	⋆0.70	⋆0.51	-	⋆0.68	⋆0.57	-	⋆0.58	⋆0.49
LSIR (R-LSIR)	Chisel	0.59 (0.44)	0.57 (0.33)	0.26 (0.39)	0.28 (0.42)	0.13 (0.19)	0.26 (0.38)	0.27 (0.38)	0.09 (0.10)	0.23 (0.32)	0.24 (0.32)	0.07 (0.07)
	Debop	0.49 (0.49)	0.40 (0.37)	▲0.65 (▲0.60)	▲0.56 (▲⋆0.54)	▲0.30 (▲0.26)	0.51 (0.48)	0.46 (0.44)	▲0.32 (▲0.29)	0.46 (0.42)	0.42 (0.39)	▲0.30 (▲0.27)
	Cov	0.50 (0.50)	0.41 (0.38)	0.64 (▲0.60)	0.55 (0.53)	0.27 (0.23)	0.52 (0.49)	0.47 (0.46)	0.30 (0.28)	0.47 (0.43)	0.43 (0.40)	0.29 (0.26)
	CovF	0.53 (0.54)	0.43 (0.41)	0.63 (0.58)	0.54 (0.52)	0.25 (0.19)	0.53 (0.51)	0.48 (0.47)	0.29 (0.25)	0.48 (0.44)	0.43 (0.41)	0.27 (0.23)
	CovA	▲0.69 (▲0.73)	▲0.60 (▲0.63)	0.53 (0.44)	0.45 (0.39)	0.17 (0.10)	▲0.58 (▲0.54)	▲0.52 (▲0.50)	0.25 (0.18)	▲0.55 (▲0.51)	▲0.49 (▲0.47)	0.24 (0.17)
	Razor	- (⋆0.79)	- (⋆0.73)	- (-)	- (0.52)	- (⋆0.32)	- (-)	- (⋆0.59)	- (⋆0.43)	- (-)	- (⋆0.56)	- (⋆0.41)

c-generality scores are much higher than Cov's scores—CovA produced programs that can correctly process 16-19% more testing inputs. This confirms the effectiveness of CovA in generating debloated programs that behave correctly for a larger set of unseen inputs. CovA is also the approach achieving the highest c-generality among all source-based approaches. CovF's c-generality scores are not as high as CovA's, which is expected because CovF focuses on robustness-related code for augmentation. Its strength lies in improving r-generality, as we will discuss in Section 4.5.

By performing augmentation, CovF and CovA pruned less code than Cov. Overall, their size-based F-scores are higher than those of all the other source-based approaches, including Cov. This, along with the results we will show in Section 4.5, indicates that CovA and CovF can considerably improve program generality without significantly affecting size reduction, thus achieving good debloating tradeoffs. Such tradeoffs are relevant for all the scenarios in which debloated programs are expected to be robust and be able to correctly handle inputs (especially feature-related inputs) other than those used for debloating.

Although CovF's and CovA's size-based F-scores are higher than those achieved by Cov, their attack-surface-based F-scores are often lower. This is because Cov, on which the two approaches are built, is not effective at reducing attack surface: its *ARed* scores are 0.33 for utilities and 0.27 for LSIR programs. When Cov's reduction score is low, that causes the reduction score for the augmentation approaches based on Cov to be even lower, which ultimately results in a low overall F-score. One possible way to address this issue is to further improve the augmentation approach by adjusting its aggressiveness based on Cov's reduction ability, which is another research direction we plan to investigate in future work.

> CovF's and CovA's augmentation strategies can lead to good debloating tradeoffs, improving program generality without greatly increasing program size. These strategies are, however, not amenable to situations in which attack surface reduction is the main goal.

4.4.3 *Comparison of Razor and source-based approaches.*

Razor is the only binary-based approach among the ones we considered (i.e., the only approach that operates on binary, rather than source code). As shown in Table 2, Razor obtained both high reduction and high c-generality scores—its size-based and attack-surface-based F-scores are the highest for both benchmarks. The main reasons for this result are that Razor (1) operates at the instruction level and (2) performs augmentation to infer complementary code not exercised by the given inputs. Because of (1), Razor's search space is larger than that of source-based approaches, which enables Razor to find more and also better tradeoffs. This also makes a comparison between Razor and source-based approaches somehow unfair because Razor, by operating at the instruction level, can remove additional, compiler-generated code. To better understand this aspect, we used Razor's coverage-based approach, which we call RazorCov, to produce debloated programs with no augmentation. We found that, by pruning instructions rather than statements, RazorCov produced debloated programs with smaller memory size and attack surface than Cov, thus making more room for augmentation. RazorCov's memory-size-based reduction scores for *Util* and *R-LSIR* programs are 0.77 and 0.69, and its attack-surface-based reduction scores for these two benchmarks are 0.56 and 0.48. Although the debloated programs have low c-generality (e.g., the *AllGen* score is 0.25 for utilities), by inferring code for augmentation, Razor effectively improves such c-generality (from 0.25 to 0.51) while still achieving high reduction (0.7 for *MRed*). This again confirms that code augmentation is beneficial for input-based debloating. Despite its effectiveness in finding a good tradeoff between reduction and c-generality, however, Razor is prone to producing programs with robustness issues, as we will discuss in Section 4.5.

> The binary-based approach Razor outperforms source-based approaches in obtaining a better tradeoff between reduction and correctness-based generality for debloating.

4.4.4 *Comparison on small programs.*

The results for the small SSIR programs show that Cov did not achieve high reduction scores (e.g., 0.13 for *SRed*) and high F-scores (e.g., 0.22 for *AllSF*). CovF and CovA, which further augment the program produced by Cov, achieved less reduction and no higher F-scores. This seems to imply that source-based augmentation is not beneficial on small programs. For small programs, Chisel and Debop pruned more code and achieved higher F-scores, due to the small search space for these programs. As before, Razor outperformed source-based approaches in reducing memory size and attack surface and obtained higher F-scores. Its *AllMF* is almost twice as high as Chisel's score (the highest one for source-based approaches).

> For small programs, source-based augmentation is not beneficial in improving the tradeoff between reduction and correctness-based generality.

4.5 RQ2: Comparison in terms of reduction, robustness-based generality, and tradeoff

Table 3 presents, in the second column, the average r-generality scores of all approaches. These scores are computed based on fuzzed inputs produced by Radamsa. The table also shows three types of reduction scores and the corresponding tradeoff scores. According to our results, CovF achieves the highest r-generality. Its augmentation is very effective and helps improve Cov's r-generality from 0.57 to 0.98. Most importantly, CovF does not achieve a significant r-generality improvement at the cost of considerably increasing program size, which is supported by the fact that CovF's reduction scores are only slightly lower than those for Cov. Among all approaches, CovF achieves the highest size-based tradeoff scores (0.8 for *SF* and 0.74 for *MF*). Its *AF* score is not the highest because a coverage-based code pruning method is not effective at reducing attack surfaces, as explained in Section 4.4.2.

The results also show that input-based approaches, which do not account for the program behavior on unobserved inputs, produced programs with low r-generality. This is because the inputs they used for debloating only reflect the typical usage of the program and do not contain invalid inputs used in less common cases. Without considering such cases, a debloating approach may remove "defensive" code and produce a less robust program. As an example, an approach may delete an if-statement that checks whether a function argument is a valid integer. Without this check, the program may crash for a non-integer argument when parsing the argument with *atoi*. Although CovA's analysis-based augmentation may also help improve r-generality, its augmentation is not targeted at finding robustness-related code. CovA may therefore identify a non-trivial amount of code for augmentation while still omitting the critical code needed to avoid robustness issues.

According to Table 3, Razor produced debloated programs with the lowest r-generality (0.39). We found that, by targeting binary instructions when pruning, RazorCov (i.e., Razor with no augmentation) produced a program with low r-generality (0.15), which indicates considerable overfit to the debloating inputs. Although Razor's augmentation can help improve r-generality, this improvement is limited because Razor, like CovA, does not target robustness-related code for augmentation. In fact, for most (80%) programs, Razor's augmentation excluded paths involving library calls (in pursuit of good reduction-generality tradeoffs). This resulted in excluding defensive checks, which often invoke library calls (e.g., `printf` and `exit`), for handling invalid cases.

We also used AFL [4] to test and evaluate the r-generality of the debloated programs based on each debloating input previously selected for fuzzing (see Section 4.2.2). Because AFL's fuzzing process is feedback-based, it generates different fuzzed inputs for different programs. Therefore, we did not evaluate r-generality based on the frequency of crashes or non-termination, as we did for Radamsa, but rather (1) checked, for each debloated program, whether a crash or non-termination ever occurred and (2) computed the ratio of tested programs that terminated correctly. The results show that CovF has the highest r-generality (0.87), followed by Chisel, whose r-generality is 0.75, and the other approaches, whose scores are all 0.67. CovF is also the approach achieving the highest size-based tradeoff scores: 0.76 for *SF* and 0.71 for *MF*. These results further confirm CovF's effectiveness.

Table 3: Reduction, robustness-based generality, and tradeoff scores (▲: highest score for source-based approaches; ★: highest score for all approaches).

Appr	RGen	Red			Tradeoffs		
		SRed	MRed	ARed	SF	MF	AF
Chisel	0.61	0.60	0.57	0.30	0.61	0.59	0.41
Debop	0.60	▲0.71	▲0.63	▲0.34	0.65	0.61	▲★0.44
Cov	0.57	0.70	0.61	0.29	0.63	0.59	0.39
CovF	▲★0.98	0.66	0.59	0.27	▲0.80	▲★0.74	0.43
CovA	0.71	0.58	0.51	0.21	0.64	0.59	0.33
Razor	0.39	-	★0.64	★0.44	-	0.49	0.42

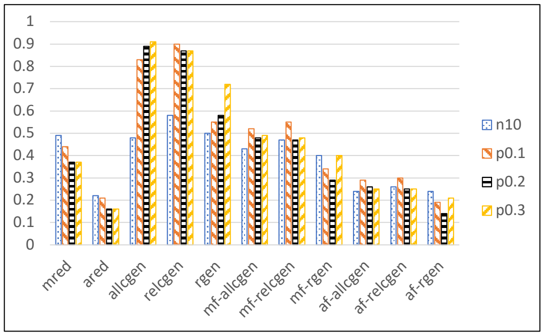


Figure 2: The average reduction, generality, and F-scores achieved with different ratios of debloating inputs (n10: ten inputs; p0.1-p0.3: 10%-30% inputs).

We also tested all the small programs in SSIR with fuzzed inputs produced by Radamsa and found that, for these programs, all approaches achieve high r-generality, with CovF's score being the highest (0.96) and Razor's score being the lowest (0.88). Similar to what we have shown in Section 4.4.4, the source-based augmentation strategies used by CovF and CovA tend to be ineffective for small programs, for which only limited code reduction is possible.

By performing robustness-oriented augmentation, CovF produced debloated programs with the highest r-generality. Conversely, Razor is more prone to producing debloated programs with robustness issues than other approaches.

4.6 RQ3: Debloating with different input amounts

In this RQ, we investigated how the debloating approaches perform using different amounts of inputs. We used the SIR benchmark for this experiment, as we have a large number of inputs for these programs. For each program p, we created three sets of inputs, I_1, I_2, and I_3, consisting of 10%, 20%, and 30% of the inputs in I, randomly selected and such that $I_0 \subset I_1 \subset I_2 \subset I_3$. We also considered I_0, the set of 10 inputs from I that we used to investigate the previous RQs. We applied each approach to debloat p based on the four input sets and evaluated the c-generality of the resulting programs based on inputs not used for debloating. We measured r-generality using the set of fuzzed inputs previously created by Radamsa.

Figure 2 presents the reduction (*mred* and *ared*), generality (*allcgen*, *relcgen*, and *rgen*), and tradeoff (from *mf-allcgen* to *af-rgen*) as average scores for all the approaches considered. These results show that, when the input size increases, the reduction scores tend to drop, and the generality scores tend to raise. When the input size increases from 10 inputs to 10% of the inputs in *I*, the debloating approaches produced programs with high c-generality scores: 0.83 for *allcgen* and 0.9 for *relcgen*. Because c-generality increases by a very large margin, the F-scores *mf-allcgen* and *mf-relcgen* also increase. However, when the input size further increases from 10% to 30%, the improvement of *allcgen* is more limited, and the *relcgen* score even slightly drops (which is possible, as the number of related testing inputs used for evaluating *relcgen* increases). In these cases, because the reduction scores drop by a non-trivial amount (e.g., from 0.44 to 0.37 for *mred*), the overall c-generality-based F-scores slightly decrease.

Unlike c-generality, r-generality (*rgen*) does not considerably increase when the input size increases from 10 to 10%. This shows that the inputs do not cover less common cases that would help increase the robustness of the debloated programs. Because the increase of r-generality does not outweigh the decrease of code reduction, the r-generality-based F-scores drop. However, when inputs increase from 10% to 30%, the *rgen* score increases by a large margin, from 0.55 to 0.72, which helps improve the corresponding F-scores. Note that the findings of this section also hold for the statement-based size reduction and the F-scores, which we do not present for all source-based approaches due to space constraints.

> When the number of inputs used for debloating increases, all approaches tend to achieve higher generality. The reduction-generality tradeoff, however, tends to decrease unless the set of inputs before the increase is small.

4.7 RQ4: Efficiency of the approaches

We compared all the approaches considered in terms of debloating time, that is, the time an approach takes to produce a debloated program based on a set of inputs. Among all approaches, Chisel and Debop are the most time consuming and need 4.7 and 5.4 hours on average to produce a debloated program. Razor, Cov, and CovA are significantly more efficient and need only 0.2, 0.4, and 0.6 minutes for debloating, on average. By performing coverage-based reduction, these approaches do not need to re-execute the same inputs again and again, as Chisel and Debop do, and their augmentation is also efficient. CovF's augmentation is slightly more expensive, as it must perform fuzzing. As a consequence, its debloating time is 3.3 minutes on average.

> Cov, Razor, CovF, and CovA take at most a few minutes to produce a debloated program. Chisel and Debop are considerably more expensive, in the order of a few hours.

4.8 Discussion

One of our main findings is that existing input-based approaches tend to produce debloated programs that are overfitted to the inputs used for debloating and thus have low generality. The two approaches we proposed provide source-based solutions that help improve program generality—both correctness- and robustness-based—without significantly increasing the size of the debloated programs, thus achieving good tradeoffs. The strength of these approaches lies in the augmentation strategies used for inferring which complementary code (i.e., code not exercised by the given inputs) to preserve for generality improvement. Although useful, however, code augmentation should be performed with caution. As we have previously discussed, when the program being debloated is small, or the number of inputs used for debloating is large, augmentation may negatively affect the reduction-generality tradeoff. Based on these observations, we believe it would be interesting to investigate how to dynamically adjust the aggressiveness of the augmentation phase based on different factors, including the program size and the inputs used (e.g., their number and coverage).

Our results show that Razor, by performing binary-based debloating, can achieve a better tradeoff between reduction and c-generality than the source-based approaches. However, Razor's instruction-based code pruning can also result in debloated programs with low r-generality that require augmentation. Therefore, despite its focus on finding a good tradeoff between reduction and c-generality, a binary-based approach like Razor is not a straightforward replacement for source-based approaches. Furthermore, it may be possible to leverage source-level information (e.g., variable names and types) to improve debloating.

Our results, which highlight the strengths and weaknesses of the various approaches, can provide guidelines to help users suitably choose a debloating approach. Razor is effective in achieving a good tradeoff between reduction and c-generality but can produce debloated programs with robustness issues, which exhibit crash or do not terminate when executed on unobserved inputs. Therefore, if robustness is important, and program source code is available, users should consider using source-based approaches instead. Among those, Cov is a simple approach that produces reasonable results. CovF should be preferred in cases in which robustness is a priority. Conversely, if the main objective is high correctness-based generality, users should consider the use of CovA. Chisel is a good choice if size reduction is of particular interest and efficiency is not a concern, as its debloating process can take several hours to finish, especially for large programs. Finally, if the main reason for debloating is attack surface reduction, Debop should be used.

In general, it is challenging to design a one-size-fits-all input-based debloating approach. Our research demonstrates the importance of accounting for both reduction and generality when debloating and provides solutions for improving generality without significantly decreasing reduction. One challenge, in this context, is that it is difficult to measure actual program generality and identify ways to improve it at debloating time, when the inputs that will be provided to the debloated program are unknown. Another interesting venue for future work would be to investigate techniques for estimating generality that can guide input-based debloating.

4.9 Threats to Validity

Like all evaluations, there can be internal and external threats to the validity of our results. To account for internal threats to validity, we used the implementation of Chisel, Debop, and Razor provided by the authors and carefully tested the approaches we implemented. As for external threats to validity, our evaluation is based on a set of 10 Unix utilities and 15 SIR programs, for which we gathered different sets of inputs. Additional empirical studies are needed to assess whether our results generalize to other programs and inputs.

5 RELATED WORK

Debloating. Existing debloating techniques that tackle the problem of eliminating a program's unneeded features often use a set of inputs (as a usage profile) for feature specification [30, 33, 49, 58, 67, 68, 70]. These techniques tend to produce programs that are overfitted to the inputs used and are not properly evaluated in terms of the generality of such programs and the reduction-generality tradeoff involved. To fill this gap, we conducted a study aiming to investigate the effectiveness of existing input-based approaches, along with a baseline and two augmentation-based approaches we developed, in terms of reduction, generality, and their tradeoff. Specifically, we selected three state-of-the-art approaches that work for C programs: Chisel, Debop, and Razor.

In addition to input-based approaches, there are approaches that use feature specifications based on manual annotations [14], human-developed domain sampling [71], configuration data [64], and surveys [6]. Yet other approaches target specific types of applications (e.g., the Chromium web browsers [50], Android apps [69]) and leverage the unique characteristics of these applications for feature identification. Compared to the input-based approaches, these approaches have more limited usability, as they require non-trivial program understanding for feature identification and/or are only applicable to specific types of programs.

Another class of related approaches are those that, instead of trying to remove a program's unneeded features, rely on static analysis [37, 39, 57] to eliminate dead/unused code, trim binaries and libraries [5, 28, 48, 52], reduce the size of applications in a given domain (e.g., containers [54], web applications [9], server systems [18], Android apps [35, 36], Java applications [15, 45, 62], and OS kernels [40]), or focus on specific tasks (e.g., safety checking [26] and API specialization [47]).

Cimplifier [54, 55], in particular, which targets container debloating, uses a technique that is similar to the one used by CovF to obtain additional inputs. Unlike CovF, however, which generates additional inputs through fuzzing to improve program robustness, Cimplifier performs symbolic execution to obtain additional inputs for improving the identification of required system resources.

Our study is also broadly related to other studies that focus on bloat analysis [13, 51], bloat detection [11, 74, 75], and identification of unnecessary code [31] and dependencies [66].

Branch/Path prediction. The augmentations performed by CovF and CovA aim to predict the "right" code to be preserved in a debloated program. In this sense, our work is related to static branch/path prediction, which aims to identify "hot" branches and paths based on heuristics [10] and machine learning [16, 17]. Unlike these approaches, the predictions performed by CovF and CovA are based on fuzzing and a combination of static and dynamic analyses. Moreover, because the branch/path prediction approaches are not developed in a debloating context, they do not need to consider the tradeoff between reduction and generality.

Feature identification and location. The debloating approaches investigated in our study rely on a set of inputs to identify desired features and locate code corresponding to these features. They are therefore related to existing approaches for feature identification (e.g., [7, 8, 25]) and feature location(e.g., [23, 60]), which are mainly designed to ease program understanding. Although intended for a different purpose, we believe that some of these approaches may be leveraged to improve debloating effectiveness.

Program repair. Similar to debloating approaches, which rely on a set of inputs for feature identification, program repair approaches [41] rely on a set of test cases to specify the correct behavior of a program. These approaches can therefore produce repaired programs that are overfitted to the test cases considered and may not correctly fix the bug at hand [65]. Techniques for mitigating this problem include generating new test cases (e.g., [72, 73]), prioritizing correct patches based on heuristics (e.g., [43]) and the use of probabilistic models (e.g., [44, 61]).

6 CONCLUSION AND FUTURE WORK

Previous research on program debloating has mostly focused on program reduction and neglected the generality of the reduced programs, that is, their ability to handle inputs that were not considered during debloating. Similarly, existing research has mostly ignored the important tradeoffs between program generality and program reduction. To fill this gap, we performed a study in which we (1) applied three state-of-the-art debloating approaches and a baseline technique to a set of 25 programs and different sets of inputs for these programs and (2) systematically assessed their performance in terms of program reduction, (different kinds of) generality, and their corresponding tradeoffs.

Motivated by our results and findings, which showed that the techniques considered could indeed produce programs that are overfitted to the inputs used and have low generality, we developed two novel augmentation-based approaches, CovF and CovA, and showed that they can improve generality without significantly affecting size reduction, and thus obtain good tradeoffs between these two important measures.

Our results also show that different debloating approaches have different strengths and weaknesses. We also provided guidelines that can help users choose the most suitable debloating approach based on their specific needs and context. Finally, our findings can also guide future research in input-based debloating.

Specifically, in future work, we will investigate ways to dynamically adjust the augmentation aggressiveness based on program size, input coverage, and other possible relevant factors. We will also extend our evaluation by including more programs, considering additional fuzzing approaches for robustness testing, and performing a user study to assess the feasibility of using debloated programs in a real-world scenario. Finally, we will investigate to what extent and how debloating is performed in industry. We also mentioned additional venues for future work throughout the paper.

ACKNOWLEDGMENTS

We thank the authors of Chisel [33] and Razor [49] for making their tools available. This research is supported in part by the United States National Science Foundation (NSF) under grants 1917924, 2114627, and CCF-0725202; the Defense Advanced Research Projects Agency (DARPA) under contract N66001-21-C-4024; the Office of Naval Research (ONR) under contracts GR10003933 and GR00004554; Amazon under an Amazon Research Award in automated reasoning; and gifts from Facebook, Google, IBM Research, and Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

REFERENCES

[1] 2022. *CIL Merger*. https://people.eecs.berkeley.edu/~necula/cil/merger.html

[2] 2022. *Software-artifact Infrastructure Repository*. https://sir.csc.ncsu.edu/portal/index.php

[3] 2022. *Websites visited for input collection*. https://drive.google.com/file/d/14FmXF2Z3YvPCVlfMKTwz-pPTn6gCztj8/view?usp=sharing

[4] AFL 2022. *AFL*. https://github.com/google/AFL

[5] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. 70–83.

[6] 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)*. IEEE, 153–162.

[7] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2018. Inferring hierarchical motifs from execution traces. In *Proceedings of IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 776–787.

[8] Fatemeh Asadi, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2010. A heuristic-based approach to identify concepts in execution traces. In *Proceedings of 14th European Conference on Software Maintenance and Reengineering (CSMR)*. 31–40.

[9] 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)*. 1697–1714.

[10] Thomas Ball and James R Larus. 1993. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI)*. ACM, 300–313.

[11] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining concern input with program analysis for bloat detection. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications (OOPSLA)*. ACM, 745–764.

[12] Suparna Bhattacharya, Kanchi Gopinath, Karthick Rajamani, and Manish Gupta. 2011. Software bloat and wasted joules: Is modularity a hurdle to green software? *Computer* (2011), 97–101.

[13] 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*. 1–5.

[14] 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*. 1–7.

[15] Bobby 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 14th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[16] Raymond PL Buse and Westley Weimer. 2009. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE, 144–154.

[17] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. 1997. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1997), 188–222.

[18] Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. 2018. Toss: Tailoring online server systems through binary feature customization. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. 1–7.

[19] Chisel 2022. *Chisel*. https://github.com/aspire-project/chisel

[20] ChiselBench 2022. *ChiselBench*. https://github.com/aspire-project/chisel-bench

[21] Clang 2022. *Clang*. https://clang.llvm.org/

[22] Debop 2022. *Debop*. https://github.com/qixin5/debop

[23] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* (2013), 53–95.

[24] 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)*. IEEE, 1102–1111.

[25] Yang Feng, Kaj Dreef, James A Jones, and Arie van Deursen. 2018. Hierarchical abstraction of execution traces for program comprehension. In *Proceedings of the 26th Conference on Program Comprehension (ICPC)*. 86–96.

[26] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. 2017. Failure-directed program trimming. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 174–185.

[27] gcov 2022. *gcov*. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[28] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1009–1022.

[29] Google Search 2022. *Google Search*. https://www.google.com/

[30] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 556–566.

[31] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2020. Is Static Analysis Able to Identify Unnecessary Source Code? *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2020), 1–23.

[32] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. 2006. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th IEEE International Conference on Program Comprehension (ICPC)*. 181–190.

[33] 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 (CCS)*. ACM, 380–394.

[34] Curt Hibbs, Steve Jewett, and Mike Sullivan. 2009. *The art of lean software development: a practical and incremental approach*. "O'Reilly Media, Inc.".

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

[36] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android application redundancy customization based on static analysis. In *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 189–199.

[37] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 12–21.

[38] 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 (ICPC)*. IEEE, 171–180.

[39] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*. ACM.

[40] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* (2020), 1–27.

[41] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Comm. of ACM* (2019).

[42] llvm-cov 2022. *llvm-cov*. https://llvm.org/docs/CommandGuide/llvm-cov.html

[43] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 166–178.

[44] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of ProgrammingLanguages (PLDI)*. ACM, 298–312.

[45] Konner Macias, Mihir Mathur, Bobby R Bruce, Tianyi Zhang, and Miryung Kim. 2020. WebJShrink: a web service for debloating Java bytecode. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 1665–1669.

[46] Joanna McGrenere and Gale Moore. 2000. Are we all in the same "bloat"?. In *Graphics interface*. 187–196.

[47] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking exploits through API specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. 1–16.

[48] 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 (PLDI)*. 164–180.

[49] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Conference on Security Symposium (USENIX Security)*. 1733–1750.

[50] 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 (CCS)*. 461–476.

[51] 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 (FEAST)*. 65–70.

[52] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. 869–886.

[53] Radamsa 2022. *Radamsa*. https://gitlab.com/akihe/radamsa

[54] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings*

of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 476–486.

[55] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. 2017. New Directions for Container Debloating. In *Proceedings of the 2017 Workshop on Forming and Ecosystem Around Software Transformation (FEAST)*. ACM, 51–56.

[56] Razor 2022. *Razor*. https://github.com/cxreet/razor

[57] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 482–501.

[58] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 335–346.

[59] ROPgadget 2022. *ROPgadget*. https://github.com/JonathanSalwan/ROPgadget

[60] Julia Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*. 29–58.

[61] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 648–659.

[62] Ulrik P Schultz, Julia L Lawall, and Charles Consel. 2003. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2003), 452–499.

[63] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *ACM conference on Computer and communications security*. ACM, 552–561.

[64] 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 (ASE)*. ACM, 329–339.

[65] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 532–543.

[66] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1021–1031.

[67] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2020. Trace-based Debloat for Java Bytecode. *arXiv preprint arXiv:2008.08401* (2020).

[68] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 361–371.

[69] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2021. XDebloat: Towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering* (2021).

[70] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program debloating via stochastic optimization. In *2020 IEEE/ACM 42st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*.

[71] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Subdomain-Based Generality-Aware Debloating. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 224–236.

[72] Qi Xin and Steven P. Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 226–236.

[73] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 789–799.

[74] 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 (PLDI)*. ACM, 419–430.

[75] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2014).

[76] 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*. ACM, 421–426.