Studying and Improving the Soundness of Input-Based Feature-Oriented Debloating

Jiahao Yuan, Weinuo Leng, Xuan Wei, Qi Xin, Xiaoyuan Xie, and Jifeng Xuan

***Abstract*—The paper presents (1) a systematic study of the soundness of feature-oriented debloating techniques that use inputs for feature specification and (2) BLOCKAUG, a new blocking method we propose for soundness improvement.**

Feature-oriented debloating aims to eliminate code bloat corresponding to unneeded program features. Many of these techniques rely on a usage profile—typically a set of inputs—to specify the features that should be preserved. They tend to produce programs overfitted to the provided inputs, introducing soundness issues in the form of bugs and vulnerabilities that threaten program correctness and security. However, no prior work has systematically studied the soundness of existing input-based debloating techniques and analyzed the types and causes of the soundness issues they introduce. To fill this gap, we applied 7 input-based techniques to 18 programs from two existing benchmarks for debloating and used three fuzzers with multiple sanitizers to detect soundness issues. Our results show that current techniques are highly unsound, as they can introduce a number of issues leading to program crashes. A key cause of the issues is the inappropriate deletion of soundness-related code, such as conditional checks for invalid cases, whose removal can lead to unexpected program states and unconditioned execution.

To improve the soundness of such input-based debloating, we propose BLOCKAUG, a blocking method applicable to coverage-based techniques. The core idea is to identify each branch deleted by coverage-based code pruning and, instead of leaving the branch empty, augment it to block any execution from passing it through and causing problems. To assess the effectiveness of BLOCKAUG, we used it to augment the debloated programs generated by four coverage-based techniques and evaluated the program soundness and generality. We found that BLOCKAUG can significantly improve soundness, at the cost of slightly increasing the program size. Although BLOCKAUG can alter program semantics, it does not significantly reduce generality; empirically, it largely preserves the program's ability to handle feature-related inputs not seen during debloating. Moreover, BLOCKAUG can forbid unexpected execution of any inputs the program should not have processed, thereby improving the program trustworthiness.

***Index Terms*—Input-based debloating, soundness study, blocking-based augmentation.**

I. INTRODUCTION

Programs are increasingly large and complex. To support a wide range of users, complex programs often provide an abundance of features. Unfortunately, most users exercise only a small portion of the features [1], leaving the remainder rarely used and mostly unneeded. This observation is supported by Quach et al. [2], who found that more than 60% of the code and their corresponding features are not exercised in typical use across a variety of complex programs including utilities, compilers, web browsers, and OS kernels. The large amount of unneeded features gives rise to code bloat [3]–[5], which can lead to severe performance degradation, resource waste, and security compromises [5]–[9].

To mitigate the negative effects of code bloat, feature-oriented debloating techniques have emerged [10]–[21] to identify and eliminate unneeded features and their code. These techniques differ in how features are specified and how debloating is performed. For example, OCCAM [22], [23] and TRIMMER [17] adopt configuration-based specifications to capture the program's deployment context. They perform debloating via compiler-based optimizations, leveraging constant configuration values to generate a reduced program tailored to a specific context.

Among existing techniques, a prominent group [10], [11], [13], [14], [16], [19], [20] specifies features using a profile of concrete inputs. These inputs characterize the features that must be preserved after debloating. We refer to such techniques as the *input-based* debloating techniques and to the specification inputs as *debloating* inputs. Because input-based specifications are relatively easy to construct, and these approaches are application-agnostic, they are well-suited for general-purpose feature-oriented debloating [19].

Guided by the input specification, current input-based techniques adopt dynamic methods based on delta debugging [24], [25] and execution-derived code coverage. These methods eliminate code that does not affect program behavior with respect to the debloating inputs. They produce a reduced program that is guaranteed to behave correctly (i.e., as the original program does) for the debloating inputs so as to "preserve" the needed features. Unfortunately, because a set of inputs often serves as an incomplete specification, current techniques tend to produce debloated programs that are overfitted to the inputs, introducing soundness issues in the form of bugs and vulnerabilities that threaten program correctness and security. For example, a technique may remove a null check for a variable x if x never evaluates to null for any debloating input, creating a soundness issue: the reduced program can crash when x is null in executions not covered by those inputs.

No prior study has systematically investigated the extent to which feature-oriented debloating techniques introduce soundness issues or analyzed the types, distribution, and causes of the issues. Such an investigation is crucial, as it provides key insights into how the soundness of feature-oriented debloating can be improved.

The authors are with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: yuanjiahao@whu.edu.cn; weinuo.leng@whu.edu.cn; isabel1015@whu.edu.cn; qxin@whu.edu.cn; xxie@whu.edu.cn; jxuan@whu.edu.cn). Corresponding author: Qi Xin. Jiahao Yuan, Weinuo Leng, and Qi Xin are also with Hubei Luojia Laboratory and State Key Lab for Novel Software Technology, Nanjing University, China.

To fill this gap, we conducted a systematic soundness study focusing on the input-based debloating techniques, given their practicality and suitability for general-purpose feature-oriented debloating. We identify soundness issues as crashes exposed by existing fuzzers and sanitizers, rather than as potential warnings flagged by static analyzers, since static analyzers are prone to producing false positive results. We empirically assessed the soundness of 7 techniques, including 6 state-of-the-art techniques—CHISEL [10], BLADE [20], COV [19], COVF [19], COVA [19], and RAZOR [11]—alongside COVEH, a technique that we implemented based on existing methods. COVEH is designed to enhance soundness by identifying and preserving error-handling and pointer validation code; it uses an existing static method [26] and patterns to detect relevant code not exercised by any debloating input and retain it together with the input-exercising code in the debloated program. We applied the 7 techniques to two existing benchmarks containing 18 programs [19], [27], and obtained the debloated programs for subsequent evaluation.

To detect soundness issues, we conducted fuzzing experiments using three representative fuzzers of different types—AFL++ [28] (gray-box), RADAMSA [29] (black-box), and SYMCC [30] (white-box)—combined with five sanitizers (ASan [31], UBSan [32], MSan [33], TSan [34], and LSan [35]) and a configuration without sanitizers (NoSan). For each issue-triggering input generated by the fuzzers, we compared the execution outputs of the original and debloated programs to confirm that the soundness issues were introduced by debloating rather than present in the original program. We also performed issue de-duplication based on the stack traces to reduce redundancy, and further identified issues triggered by fuzzed inputs that are related to the debloating inputs in terms of the features they exercise. To better understand the observed issues, we categorized them based on their symptoms, analyzed the functions in which they arise and their frequencies, and summarized their underlying causes.

Our main result is that current techniques, while reducing 37–71% LoC of the program, can introduce 49–93 soundness issues absent in the original program. Notably, about 35 are exposed by inputs that are related to the debloating inputs in terms of the features they represent. These results clearly demonstrate the unsoundness of current debloating techniques. A key cause of the soundness issues is that existing techniques are designed to pursue aggressive code reduction, which can unexpectedly remove soundness-related statements, including assignments, return statements, and "defensive" if-statements checking invalid inputs. Removing assignments or returns can lead to indeterminate variable values and undefined function behavior, whereas deleting defensive if-statements can cause unconditioned execution and crashes. Although techniques such as COVF, COVA, and COVEH attempt to retain additional uncovered code using dynamic or static analyses, these approaches remain insufficient: COVF's dynamic analysis is inherently incomplete, COVEH targets only a subset of soundness-related code, and COVA's static analysis is imprecise and improves soundness only by retaining excessive code.

The root cause of the soundness issues is inappropriate code deletion during debloating. However, there is no straightforward strategy to resolve these issues by restoring deleted code without considerably increasing program size. As an alternative, we proposed a new *blocking* method, BLOCKAUG, which can be easily applied to coverage-based debloating techniques to significantly reduce bugs and vulnerabilities in their debloated programs and improve soundness. The core idea is to (1) identify each branch deleted by a coverage-based technique and (2) instead of leaving the branch empty, augment it (by adding print and exit statements) to prevent any execution from passing through it. Essentially, the blocking method forbids the execution for any input that the debloated program cannot process in the same way as the original program, thereby resolving soundness issues.

To assess the effectiveness of BLOCKAUG, we applied it to the debloated programs generated by the four coverage-based techniques—COV, COVF, COVA, and COVEH—and ran the same fuzzing experiments to evaluate soundness. Because blocking augmentation can change program semantics thus affect generality—which reflects a program's ability to handle inputs not observed during debloating, especially those related to the debloating inputs in terms of the same features—we also measured the generality of these block-augmented programs using the same inputs from a previous study [19].

Our results are encouraging, as they show that (1) BLOCKAUG reduces more than 98% of soundness issues; (2) BLOCKAUG incurs only a slight increase in program size (about 6%); (3) BLOCKAUG does not significantly weaken the program's ability to handle feature-related inputs not observed during debloating, causing only a 6% drop in generality; and (4) BLOCKAUG improves the trustworthiness of debloating by successfully warning about and preventing the execution of inputs that the debloated program should not process, rather than silently producing incorrect outputs.

Overall, BLOCKAUG represents an important step toward improving soundness and can be naturally combined with coverage-based code pruning to enable more reliable feature-oriented debloating.

This paper makes the following main contributions.

- A systematic study of 7 input-based debloating techniques that investigates the soundness issues they introduce, as detected via fuzzing, and analyzes the categories, distribution, and causes of these issues.
- A new blocking method, BLOCKAUG, for improving the soundness of feature-oriented debloating, along with a comprehensive evaluation demonstrating its effectiveness.
- An artifact containing the study data and results, test scripts, tools, and experimental outputs, made publicly available at [36].

II. INPUT-BASED DEBLOATING APPROACHES

A. Input-Based Debloating

Feature-oriented debloating techniques remove code corresponding to unneeded features. A prominent class of such techniques is input-based. These techniques use a set of user-provided debloating inputs to specify the features that must be preserved. Given a deterministic program p and a set of

debloating inputs I, which constitutes a usage profile for p, input-based debloating is the process of systematically removing code from p to produce a debloated program p' such that, for each input $i \in I$, p' should produce the same output as the original program p, that is, $p'(i) = p(i)$, where $p(i)$ and $p'(i)$ denote the outputs of p and p' on input i, respectively.

B. Previous Input-Based Debloating Approaches

CHISEL [10] is a debloating technique that performs reinforcement-learning-guided delta debugging for code pruning, aiming to identify a minimal program (with the guarantee of *1-minimality* [37]) that can handle the given inputs correctly.

BLADE [20], while also using delta debugging, achieves faster and more scalable debloating by employing a structure-aware, hierarchical approach that supports simultaneous code reduction. The debloating process runs in $\mathcal{O}(n)$ time complexity (n is the number of statements in the program), enabling it to deal with larger codebases.

COV [19] performs coverage-based debloating by tracking and retaining statements exercised by the given inputs and deleting others. It additionally performs dead code elimination (DCE) to remove code that is exercised by the given inputs but does not affect the program's behavior, including the "dangling" statements (e.g., an if-statement with an empty then-branch) and the unused variables and labels.

COVF and **COVA** [19] are built upon COV. Both approaches first perform coverage-based debloating using COV and then augment the debloated program by restoring some of the deleted code. COVF aims to improve the soundness of the debloated program. The idea is to use fuzzing to generate additional inputs covering the program's edge use cases and further use such inputs to identify and preserve code to enhance program soundness. More specifically, COVF performs fuzzing to generate crash-inducing inputs, adds these inputs to the set of debloating inputs, and then re-debloats the program, producing an augmented program that can handle both the user-specified debloating inputs and the crash-inducing inputs. COVA aims to improve the generality of the debloated program. It performs static and dynamic analyses to infer feature-related code based on a set of heuristics and adds back the inferred code to the debloated program.

RAZOR [11], unlike the above source-based techniques, directly operates on program binaries post-deployment. It utilizes program tracing to log all executed code and uses four heuristics to infer more complementary code that it deems necessary to support user-specified features. It then rewrites the program binary to only retain the executed code and the inferred code.

Summary. Most of these approaches are not soundness-aware. CHISEL and BLADE perform aggressive code pruning to generate a minimal program that adheres to the input specification. COV does not preserve any code that is not covered in the execution of the debloating inputs. These techniques can severely suffer from the input-overfitting problem. COVA, RAZOR, and COVF perform code augmentation to alleviate the problem. However, COVA and RAZOR are generality-oriented, as their augmentation methods aim to identify missing code that implements the user-specified features for add-back, and are not specifically designed to address soundness issues. While these augmentations can sometimes improve soundness as a side effect, because they are based on heuristics, they can either retain too little code to tackle the soundness issues or too much code, undermining the debloating goal. There seems no easy adaption to address this problem. Fundamentally, heuristics-based methods are not accurate. COVF is the only soundness-aware technique that leverages fuzzing for soundness improvement. Unfortunately, fuzzing as a dynamic approach is inherently insufficient to exercise all relevant code.

Current input-based techniques rely on an extensive set of debloating inputs representing various feature-exercising scenarios and use result validation mechanisms (based on for example static analyzers such as sanitizers) during debloating to provide confidence on the quality of the debloated programs generated. However, there is no guardrail providing guarantees on the debloated programs' behavioral correctness for inputs not seen while debloating. Delta-debugging-based techniques have no warrants on the correctness of the debloated programs handling any unseen inputs. Coverage-based techniques can ensure program correctness in handling unseen inputs whose execution footprint (code coverage) is the same (or a subset) of those of the given inputs (assuming deterministic execution), but provide no guarantee beyond.

C. CovEH: Coverage-Based Debloating with Static Augmentation for Soundness Improvement

To complement COVF's dynamic approach for soundness improvement, we also explored and implemented a static approach, COVEH, which identifies and retains the error-handling and pointer validation code for augmentation. Error-handling and pointer validation code are common for soundness assurance. The lack of error-handling code allows programs to continue the execution even when errors are encountered, which is problematic, and the absence of pointer validation code allows programs to access invalid memory locations, which directly results in crashes. COVEH relies on an existing static method [26] to detect the error-handling code and retain them, and it uses heuristics for pointer validation code detection and preservation.

Note that we do not claim COVEH as a novel technique contribution, as its main components, the error-handling and pointer validation code detectors, are built upon a previous method and simple heuristics. Rather, we implemented and used COVEH for an exploratory purpose, to investigate whether, and to what extent, static approaches could alleviate the soundness issues, in the context of input-based debloating.

Like COVA and COVF, COVEH performs coverage-based code pruning using COV to generate a reduced program p'. It then augments p' with additional code from the original program pertaining to error handling and pointer validation. The augmentation of COVEH consists of three steps: (1) identifying branches related to error handling; (2) identifying branches related to pointer validation; and (3) restoring the identified branches and their dependencies.

Identifying Error-Handling Branches. To detect code related to error handling, CovEH follows a previous approach [26]. At a high level, the approach is as follows. It first identifies an initial set of error-handling branches B_{init} based on keyword matching. B_{init} serves two purposes: (1) identifying the potential error-handling functions as those explicitly called in each initial branch of B_{init} (no nested calls included) and (2) approximating the likelihood of each potential function being related to error handling. With the likelihoods of the potential error-handling functions, the approach examines each branch b of the program and computes a score measuring the error-handling likelihood of b. The score is the sum of the likelihoods of the potential error-handling functions explicitly called in b. Lastly, any branch with a score above a certain threshold is considered as a final error-handling branch.

Specifically, CovEH collects the initial set of branches that contain any of the error-handling-related keywords defined in a user-specified keyword list. In our implementation, we use a set of general keywords including "error", "fail", "fatal", and "invalid". Users can add more keywords to the list. Note that identifying branches based on keywords is not a simple string matching with the entire branch code, as this would lead to many false positives. Instead, CovEH only analyzes the top-level statements of the branch code, (i.e., it ignores any statements inside nested branches such as the inner if and while blocks) and matches the keywords with the names of the called functions and the string literals (displaying for example error messages) appearing in the top level. Note that there is no cutoff threshold used for initial branch identification. As long as the top-level statements of a branch uses any of the keywords, the branch is determined as intial.

After identifying all the initial branches containing the keywords, CovEH considers the functions explicitly called in these branches as the potential error-handling functions. For each such function f, it calculates a score to quantify its likelihood of being error-related. The score is calculated as a ratio n_{init}/n_{total} where n_{init} is the number of calls to f in the initial set of branches and n_{total} is the total number of calls to f in the entire program. The idea behind this is that, if a function is invoked more frequently in the initial error-handling branches than in the entire program, it is more likely to be related to error handling. CovEH then collects all branches that explicitly call the potential error-handling functions, calculates the score for each branch by summing the scores of all functions called in the branch, and considers the branch as a final error-handling branch if its score is above a certain threshold. (The threshold is chosen as 0.5 based on our experimental observation. We found that CovEH's soundness is not very sensitive to the threshold selection [38].) Note that in this way CovEH can detect branches containing no keywords as also error handling. For example, some branch b calling the function *lerrsf* in program *flex-2.5.4* does not contain any identifiable keywords. However, because a large fraction of all branches calling *lerrsf* (7 out of 13) contain the keyword "error" (appearing in the error messages generated as warnings), *lerrsf* has a score 0.54 (7/13). As a result, all branches calling f, including b, are considered as error-handling branches by CovEH.

```
1 // tmp_22 = settime((struct timespec const *)(&when));
2 // if (tmp_22 != 0) {
3 //     tmp_20 = gettext(``cannot set date'');
4 //     tmp_21 = __errno_location();
5 //     error(0, *tmp_21, (char const *)tmp_20);
6 //     ok = (_Bool)0;
7 // }
```

Fig. 1. An example of if-condition deletion.

Identifying Pointer Validation Branches. CovEH detects branches corresponding to the common null pointer check and boundary check for pointer validation via pattern matching. To detect null pointer checks, CovEH looks for if-conditions that check a pointer variable's nullity as either unary negation (e.g., `!ptr`) or equality check (e.g., `ptr == NULL`). To detect boundary checks, CovEH looks for if-conditions that check a variable's value being out of bounds, such as `i >= len` and `i < 0`. The variable currently being checked is restricted to an array access index (e.g., `arr[i]`) or an increment operation (e.g., `i++`) inside a loop. No restriction of variables can incur high false positives for validation code identification.

Restoring the Identified Branches. After collecting all the branches related to error handling and pointer validation, CovEH examines each such branch b and compares the debloated program and the original program to see (a) if b has been deleted and (b) the parent statement has not been fully deleted (e.g., the parent statement is a "dangling" if-statement with an empty body). CovEH restores the branch only if (a) and (b) are both met. If (b) is not met, CovEH does not restore b, as b in this case is an inner-level branch of another deleted branch and is often not feature-related. To understand this, consider the example of Figure II-C. Although the branch of lines 3–6 is error-handling, it is used to handle the error result of `settime` (line 1). However, setting time is not a needed feature and the call to `settime` and its located branch has been deleted by Cov (which CovEH invokes for coverage code pruning). If `settime` is unwanted, there is no need to restore any error-handling part for it.

For any branch b CovEH decides to restore, CovEH not only adds back the code of b itself but recursively handles the dependencies by restoring the definitions of invoked functions and the labeled bodies for goto statements. For example, if a call to a function is added back but the function's definition is deleted, CovEH also adds back the function's definition and all its dependencies recursively.

III. Study of the Soundness of Input-Based Debloating Techniques

We conducted a study to assess and analyze the soundness issues introduced by current input-based debloating techniques. We seek to answer the following research questions.

- **RQ1:** How many soundness issues are introduced by current input-based debloating techniques?
- **RQ2:** What are the categories and frequencies of the soundness issues?
- **RQ3:** How are the soundness issues distributed in the debloated program? What functions contain the most issues?

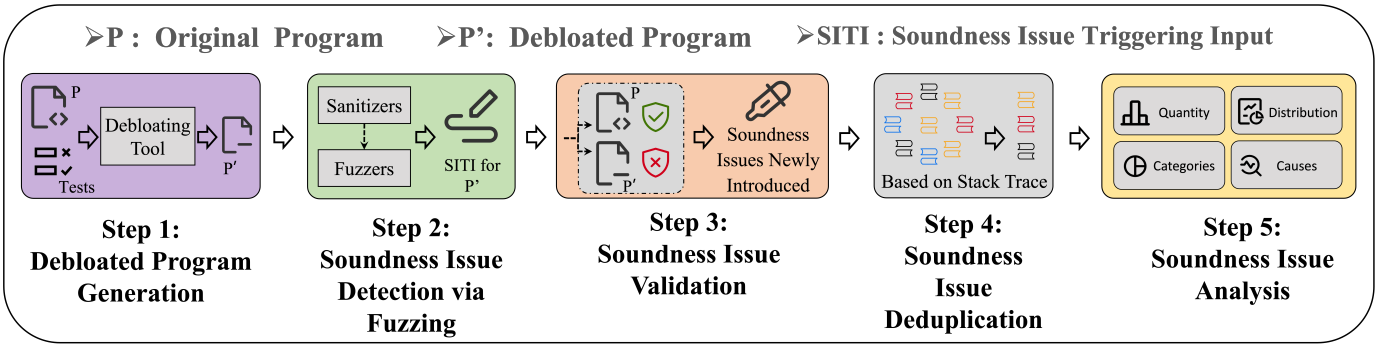

Fig. 2. An overview of our method for soundness issue study.

- **RQ4:** How are the soundness issues introduced by debloating?

Answers to RQ1 and RQ2 allow us to assess the extent to which existing techniques introduce soundness issues and analyze the issue categories and frequencies to understand the symptoms. The answer to RQ3 helps us identify characteristics of functions that are the most "problematic". Debloating for such functions can be done with more and better static and dynamic analyses to improve soundness. Finally, RQ4's answer can shed light on how to avoid the issues.

We first outline the study methodology, then describe its steps and experimental setup in detail, and finally present the results and analysis.

A. Methodology Overview

Figure 2 is an overview of our method for studying soundness of current techniques. It has five steps: (1) debloated program generation, (2) soundness issue detection via fuzzing, (3) soundness issue validation, (4) soundness issue deduplication, and (5) soundness issue analysis.

In the first step, we applied the 7 input-based debloating techniques to 18 programs from two established benchmarks to generate debloated programs based on the debloating inputs. To investigate the soundness of these techniques, in the second step, for each debloated program generated, we performed fuzzing using three types of fuzzers with five sanitizers and NoSan (no sanitizer) to detect soundness issues reported as crashes. A soundness issue may exist both in the debloated program and in the original program, and may not be related to debloating. To identify issues introduced by debloating, in the third step, we perform soundness issue validation to filter out issues that also exist in the original programs. In the fourth step, we perform deduplication to remove redundant issues based on the stack traces. Finally, we analyzed the deduplicated issues to answer the research questions.

B. Debloated Program Generation

Program benchmarks. We chose as the benchmarks CHISELBENCH [27], which has been widely used to evaluate existing debloating techniques [11], [14], [19], [20], [39], [40], and LSIR [19], [41], which has been used by a previous study [19] investigating the generality and reduction of debloated programs. We excluded the SSIR benchmark from the study [19] because its programs are too small — the average size is below 1 KLOC. In contrast, the average sizes of the programs in CHISELBENCH and LSIR are considerably larger and are 12.56 and 30.14 KLOC respectively.

In total, 18 programs are used in our study. Most programs are command line utilities. The others are the text editor *vim* and the command interpreter *bash* from LSIR. Each program is associated with a set of debloating inputs (provided in the benchmark) for which the debloated programs should behave correctly (i.e., as the original program does). The average numbers of debloating inputs for programs in CHISELBENCH and LSIR are 32 and 10 respectively.

Another reason why we chose these benchmarks is that they provide the all-in-one-file (merged) versions of the original programs. Current tools (due to implementation limitations) can only run with these merged programs but not the original multi-file programs for debloating. In particular, BLADE, COV, COVF, and COVA in their current implementations do not support debloating for multi-file programs. CHISEL, due to its failure of handling the macro syntax, can produce invalid result for multi-file programs.

Debloating tools. We studied the 7 debloating tools, BLADE, CHISEL, COV, COVA, COVF, COVEH, and RAZOR, all of which require a usage profile in the form of a set of debloating inputs as the feature specification. We implemented COVEH on top of COV and used the other tools from the repositories [11], [39], [42], [43] prepared by the authors and only slightly adapted the tools to ensure compatibility with recent libraries (e.g., Clang 15.0.7) set up in our experiment environment. We excluded tools that do not use inputs or require additional information (e.g., manual annotation) for feature specification such as Trimmer [17], [44], LMCAS [45], and Carve [12], as these techniques, due to their reliance on special forms of specification, are not highly applicable for general feature-oriented debloating. Given that the specifications are different, it is also unfair to compare them with the 7 techniques considered. We did not include optimization-based techniques Debop [14] and DomGad [40], as they adhere to a different criterion and do not guarantee to generate programs that behave correctly for all the debloating inputs. We also

excluded others that are not implemented for C program debloating (such as JShrink [16] and MiniMon [21]), those for which the implemented tools are not available or not usable (such as Hecate [13]), and those that are not for feature-oriented debloating (e.g., BlankIt [46]).

Debloating Process. We applied the 7 tools to each program to generate the debloated programs. COVF and COVA require extra parameters to control augmentation, and we followed the previous study [19] to configure the tools. More specifically, we configured COVF to generate 10 fuzzed inputs for each debloating input. For COVA, considering that the previous study experimented with a range of thresholds from 1 to 50, we set the parameter value as 25 to select the top-25 functions for augmentation. RAZOR can use four different heuristics for augmentation. We experimented with five different types of debloating using each heuristic and no heuristic to generate five debloated programs and computed the average result across these programs. We did not include CHISEL's and BLADE's debloating results for *vim*, as *vim* is large and the delta-debugging processes of CHISEL and BLADE are slow, and as a result, CHISEL and BLADE do not converge (i.e., finish the delta-debugging processes) within a reasonable amount of time. In 24 hours, CHISEL and BLADE can only reduce less than 5% of the code, which is far from convergence. Using such partially debloated programs to investigate soundness is problematic. We also excluded RAZOR's results for *vim*, *bash*, and *make* due to errors detected during its tracing phase.

C. Fuzzing

In this study, we used fuzzing both with and without sanitizers to expose soundness issues of the debloated programs generated by various techniques.

Fuzzers. We used three fuzzers, AFL++, RADAMSA, and SYMCC, which are state-of-the-art gray-box, black-box, and white-box techniques whose tools are available and relatively easy to use.

Seed inputs. The three fuzzers require seed inputs (provided as seed files) for mutation. For the programs of the study, a seed input can be in one of the two forms containing: (1) command line options and their values or (2) file contents. Consider the command line input "*uniq -f 2 f1.txt*". A fuzzer can mutate the command line itself or the file content of *f1.txt*. For any input containing both the command line options and files, we created two types of seed inputs: (1) *argv-fuzz*, which includes the command line options and their values (e.g., "*-f 2 f1.txt*") and (2) *file-fuzz*, which includes the file content (e.g., the content of *f1.txt*). The use of both types of seed inputs gives the fuzzer the opportunity of mutating both the command line and the file content. Note that the actual seed inputs for *file-fuzz* can be multiple, as there can be multiple files. For any input containing either the command line options or files, the corresponding type of seed inputs is created.

We created the seed inputs based on each debloating input. Since feeding a fuzzer with multiple seed inputs for one fuzzing run is often not viable (for example, AFL++ does not support fuzzing both command line options and file contents in one run, and for file-fuzz, it cannot use multiple command line options such as *"uniq -f 2 f1.txt"* and *"uniq -f 3 f1.txt"* to fuzz in one run), we chose to run the fuzzers each time with one seed input provided. We refer to a fuzzing process based on one seed input as a fuzzing *trial*. Because the number of seed inputs can be large and AFL++'s and SYMCC's fuzzing processes are expensive, we did not run AFL++ and SYMCC using all the seed inputs, but instead randomly chose three *argv-fuzz* inputs and three *file-fuzz* inputs (at most, since not all debloating inputs use files) and ran the two fuzzers with each selected seed input. Since RADAMSA is much more lightweight, we ran it with every seed input.

Sanitizers. We used five sanitizers, the AddressSanitizer (ASan) [31], MemorySanitizer (MSan) [33], UndefinedBehaviorSanitizer (UBSan) [32], ThreadSanitizer (TSan) [34], and LeakSanitizer (LSan) [35], implemented in the Clang framework [47] together with no sanitizer (NoSan) for soundness issue detection. To allow soundness issues to be reported by a specific sanitizer t, for AFL++ fuzzing, we set the environment variable *AFL_USE_X=1* (where X is the sanitizer name) to activate the sanitizer t when using the tool. For RADAMSA fuzzing and the issue validation in the next step, we used Clang to compile the debloated program with the sanitizer option turned on for t (e.g., using *-fsanitize=address* to turn on ASan). We also set the option *abort_on_error* to 1 for all the sanitizers to ensure that the program aborts when a soundness issue is detected. A soundness issue is exposed as a crash of the debloated program determined by the exit code of the execution (between 130 and 139 inclusive). A sanitizer with the option *abort_on_error* set as 1 can always trigger a crash when it finds an issue. Given that SYMCC directly analyzes the source code of a program for fuzzing and it is not sanitizer-guided (as explained in the paper [30]), we bundled SYMCC with NoSan for soundness issue detection.

Fuzzing process. For each debloated program p', we prepared the seed inputs (all and selected) using the above method and then ran all fuzzers to detect soundness issues reported by different sanitizers and NoSan. For *argv-fuzz*, to allow fuzzers to read mutated command line from files, we slightly modified p' by adding an inclusion of *argv-fuzz-inl.h* and a statement invoking *AFL_INIT_SET0* in the main function, according to the AFL++'s guidelines [48]. The number of fuzzing trials performed by AFL++ for a debloated program is up to 36 (using up to 6 selected seed inputs with 5 sanitizers plus NoSan for each). Each fuzzing trial takes 24 hours, a common time budget for fuzzing [49]–[53]. SYMCC is not sanitizer-guided. The number of fuzzing trials it performs for a debloated program is up to 6 (using up to 6 selected seed inputs with NoSan for each). Here a fuzzing trial also takes 24 hours. RADAMSA is a black-box fuzzing tool. We configured it to generate 500 fuzzed inputs for each seed input to detect soundness issues for a debloated program compiled with and without sanitizer instrumentation. A fuzzing trial consists of the generation of the 500 fuzzed inputs and the execution of these inputs, each having a timeout of half a second.

Feature-related soundness issues. In the context of feature-oriented debloating, we not only identified soundness issues exposed by all fuzzed inputs but also distinguished those detected by fuzzed inputs that are related to the debloating

inputs in terms of the features they exercise. That is, we additionally identified the feature-related soundness issues. We reported the feature-related issues separately, as the debloated program is not supposed to handle all inputs but only those that correspond to the needed features. The result can help us assess the quality of debloating with respect to the inputs the debloated program intends to handle and show how debloating fullfills its promise.

To determine the feature-relatedness of two inputs, we followed the practice of a previous study [19]. For programs taking command-line inputs, if a fuzzed input i' uses the same command line options (only options, not values) with the original debloating input i, then any issues exposed by i' are feature-related. In this definition, the fuzzed inputs generated by *file-fuzz* are always feature-related to the original debloating inputs, since they only modify the file content and do not change the command line options. The fuzzed inputs generated by *argv-fuzz* do not necessarily represent the user-specified features, since the command line options often change. For example, the fuzzed input *"gzip -z file.txt"* (compression) and the debloating input *"gzip -d file.txt"* (decompression) exercise different features whereas *"gzip -z file.txt"* and *"gzip -z file.txt"* with different file contents of *file.txt* represent the same feature. In our study, we detected feature-related soundness issues as those found by fuzzed inputs derived from *file-fuzz* and reported the numbers of all soundness issues and the feature-related soundness issues. *bash* and *vim* are programs that do not use command line inputs. For these programs, the previous study [19] uses the functionality tags associated with the inputs to determine feature-relatedness. In our context, where the fuzzed inputs have no tags, it is not clear how to determine the input feature-relatedness. So we excluded these two programs for feature-related soundness analyses.

D. Soundness Issue Validation

A soundness issue detected in the debloated program may either pre-exist in the original program or be introduced by the debloating process. To identify debloating-related issues, in this step, we ran both the original program p and the debloated program p' against each crash-triggering input i reported by the fuzzers (with and without sanitizers) and compared the outputs. If running p' with i results in a crash, while running p with i does not, we keep i in later steps for further analysis. Otherwise, we discard i, as non-debloating-related issues are not the concern of this study.

E. Soundness Issue Deduplication

Soundness issues detected in the above steps may be redundant. For example, various inputs triggering the same out-of-bounds crash for an array access, despite with different accessing indices, can be considered as the same issue. For each program p', we performed deduplication on the soundness issues reported by the fuzzers and sanitizers to obtain a set of unique soundness issues. We considered two issues as the same if they share the same stack trace extracted from the error summary reported by the sanitizers. In cases where the error summary did not contain a stack trace but an error location (specified by error line and column numbers), we took the error location as a unique identifier for deduplication. NoSan provides no error summary, and we used the gdb *bt* command to obtain a stack trace based on the core dump for deduplication. Unless otherwise noted, soundness issues refer to *unique* soundness issues in the rest of the paper.

F. Experimental Environment

We ran the most expensive fuzzing experiments on a cluster of machines provided by our affiliated institution(s) and others on our laptops. Each machine in the cluster is equipped with a 16-core dual Intel Xeon E5-2630v3 2.4 GHz CPU and 96 GB ECC 2133 MHz DDR4 RAMs. All experiments were run in Docker containers with Ubuntu 22.04, Clang 15.0.7, and Python 3.10 installed. We note that the fuzzing experiments are very expensive, as they involve running three fuzzers—two with 24-hour time budgets—on the debloated programs generated by each tool using five sanitizers plus NoSan, with different seed inputs. The machine time added up is over 3K days. Due to the parallel use of the machines, the actual running time is much less but still more than one month.

G. Results

This section shows our results and analyses for each RQ.

TABLE I
THE SIZE REDUCTION AND SOUNDNESS RESULTS OF DIFFERENT DEBLOATING TECHNIQUES.

DebTool	Benchmark	Reduction	#All Issues	#Related Issues
Blade	ChiselBench	75.04%	57.0	31.4
	LSIR	60.09%	83.1	38.0
	Average	67.56%	70.0	34.7
Chisel	ChiselBench	71.51%	84.3	50.3
	LSIR	71.40%	88.6	72.8
	Average	71.46%	86.4	61.5
Cov	ChiselBench	54.18%	98.2	48.1
	LSIR	55.41%	88.5	47.5
	Average	54.80%	93.3	47.8
CovA	ChiselBench	36.34%	46.2	16.9
	LSIR	37.53%	52.4	13.2
	Average	36.94%	49.3	15.0
CovF	ChiselBench	51.24%	58.5	41.7
	LSIR	48.45%	47.9	12.5
	Average	49.84%	53.2	27.1
CovEH	ChiselBench	43.04%	56.8	44.7
	LSIR	49.93%	51.9	12.7
	Average	46.48%	54.3	28.7
Average		54.51%	67.7	35.8

* RAZOR's results are not shown due to RAZOR's failure in handling the sanitizer-instrumented binaries, the lack of stack traces for soundness issue deduplication, and the inabilities of AFL++ and SYMCC in fuzzing RAZOR's debloated programs.

1) Results for RQ1 about the Number of Soundness Issues Introduced and Detected: Table I is a summary of the debloating tools (*DebTool*), the benchmarks (*Benchmark*), the size reduction (*Reduction*) achieved by each debloating tool, the average number of all soundness issues (*#All Issues*) introduced by debloating, and the average number of feature-related soundness issues (*#Related Issues*). The size reduction is calculated as the fraction of LoC (lines of code) removed.

TABLE II
RESULTS OF THE THREE FUZZERS IN TERMS OF THE AVERAGE TIME TAKEN FOR ONE FUZZING TRIAL, THE NUMBER OF TRIALS, AND THE ISSUES FOUND.

FuzzTool	Time per trial	Trials	#All Issues	#Issues Not Detected By Others
SYMCC	1 day	486	627	106
AFL++	1 day	2916	3250	1622
RADAMSA	∼4 min.	10908	4407	3057

TABLE III
SOUNDNESS ISSUES DETECTED BY THE FIVE SANITIZERS AND NO SANITIZER (NOSAN).

Sanitizer	#Issues						#Issues Not Detected By Others	#All Issues
	CHISEL	BLADE	COV	COVA	COVF	COVEH		
NOSAN	323	287	248	118	137	159	NA	1272
ASAN	149	129	304	274	191	168	789	1215
UBSAN	453	388	581	210	356	338	2026	2326
MSAN	87	111	403	225	202	235	883	1263
TSAN	328	253	362	157	190	144	613	1434
LSAN	387	223	294	160	155	107	557	1326

The table presents the results for 6 debloating techniques. RAZOR was excluded due to unresolved technical failures, as will be discussed at the end of this RQ section. The 6 techniques presented in Table I can achieve over 36% size reduction while introducing many, or up to ∼93, soundness issues. The *#Related Issues* column shows that these techniques introduced about 35 feature-related soundness issues, posing serious correctness and security threats to the preservation and implementation of the user-specified features. These results, along with our result showing RAZOR's debloated programs' proneness to crashes, highlight the severe soundness problems of current feature-oriented techniques.

Of all the 6 techniques, CHISEL achieves the highest reduction rate (∼71%) and introduces the second largest number of soundness issues (∼86). This shows that CHISEL's debloating process is overly aggressive and highly unsound. BLADE achieves a similar reduction rate (∼67%) but introduces fewer soundness issues (∼70). COV has the largest number of soundness issues (∼93), while only achieving a moderate reduction rate (∼55%). By comparing the results of BLADE and COV, one can see that it is possible to achieve higher size reduction (∼67% vs. ∼55%) while significantly improving the soundness (∼70 vs. ∼93 issues).

Built on top of COV, COVA, COVF, and COVEH are three approaches that additionally add back code for augmentation purposes. Overall, they achieve better soundness than the other debloating approaches (∼49, ∼53, and ∼54), but at the cost of lower reduction rates (∼37%, ∼50%, and ∼46%). Being the most conservative in terms of size reduction, COVA introduces the lowest number of soundness issues (∼49). Note however that COVA's soundness improvement is achieved at the cost of a significant amount of code restoration — its reduction rate is as low as 36.94%, weakening the goal of debloating in terms of size reduction. Our box plot results [54] reflecting the distribution of the issues across programs show that the median values are often over 40 and 10 for all and related issues respectively. This means that many programs do contain many issues, and it is not the case that the average results are skewed by outliers.

Despite the improvement, there are still many soundness issues unresolved by these augmentation-based approaches. COVA aims at identifying and preserving feature-related code and only improves soundness as a side effect. COVF and COVEH are more soundness-oriented, but their augmentations are still limited. COVF's augmentation is based on fuzzing, a dynamic approach, which is not exhaustive to capture all relevant code addressing soundness issues. COVEH's augmentation is based on static analysis and is more efficient. However, while adding back the error-handling and pointer validation branches to improve soundness is intuitive, soundness issues cannot be fundamentally avoided. For example, deleting a branch that involves computing a value for i can make i hold an incorrect out-of-bound value before using it to access an array, leading to a crash. We further analyze the causes of soundness issues in RQ4 and propose a more effective solution to improve soundness in Section IV.

Table II presents the fuzzing result in terms of the time taken to finish one fuzzing trial (*Time per trial*), the total number of trials (*#Trials*), the total number of soundness issues detected (*#All Issues*), and the number of issues detected exclusively by each tool and not by others (*#Issues Not Detected By Others*). A fuzzing trial as defined in Section III-C is a fuzzing process based on one seed input for one program with one (or no) sanitizer used. Note that RADAMSA has more than 10K trials, as it used all seed inputs for fuzzing whereas AFL++ and SYMCC use selected seed inputs. Our result shows that RADAMSA detected the most, or 4407, soundness issues. AFL++ found 3250 issues. Using symbolic-execution-based fuzzing, SYMCC faces scalability challenges, and it detected only 627 issues. Most (or 521) of the detected issues are however also detected by AFL++ and RADAMSA. The blackbox RADAMSA found even more issues than SYMCC possibly because the issues introduced by current techniques are shallow and can be easily detected.

Table III presents the numbers of issues reported by the five sanitizers and NoSan for programs generated by the five debloating tools excluding RAZOR (columns 2-6), all the issues each sanitizer reported (*#All Issues*), and issues exclusively identified by each sanitizer or NoSan but not the others (*#Issues Not Detected by Others*). Because the

stack trace obtained from the gdb command line for a NoSan issue is different from those reported by the sanitizers—the NoSan stack trace contains only line information for each stack frame and no character locations—we did not examine the issues exclusively identified by NoSan (row 3, column 8 is NA). Our result shows that UBSan detected the most (about 2K) soundness issues and others can find more than 500 issues each. All sanitizers are useful and have complementary powers, as each can find a number of issues that are not detected by others.

RAZOR's evaluation. RAZOR is excluded from the comparison with other tools due to its failure of dealing with sanitizer-instrumented binaries for debloating and its tendency to alter executable structures, which makes us unable to use debugging tools like GDB for soundness issue deduplication.

We were nevertheless able to fuzz RAZOR's output with the RADAMSA tool without deduplication. The experiment was based on the debloated programs generated by RAZOR using all the augmentation heuristics for 15 programs (for which RAZOR ran without errors). We found that 69% of the inputs caused crashes, highlighting the unsoundness of RAZOR's debloating approach.

Weakness of restoration-based augmentation. Current restoration-based augmentation approaches can only partially address soundness issues. COVA and RAZOR are not designed to address soundness problems; COVF, which uses a dynamic fuzzing-based method, is not guaranteed to find all soundness-issue-triggering inputs for augmentation; and COVEH is not perfect at detecting all error-handling and pointer validation code. Also, not all soundness-related code are related to error handling or pointer validation. Note that some approaches (e.g., COVA) can perform more aggressive augmentation to improve soundness. However, such augmentation can easily restore non-feature-related code, leading to a significant increase in program size and thereby compromising the goal of debloating.

Finding 1: The 6 source-based debloating techniques can introduce a number of soundness issues, threatening program correctness and security. The binary-based technique RAZOR is also unsound, with its debloated program crashing on 69% of fuzzed inputs. Most soundness issues are exposed by the gray-box AFL++ and the black-box RADAMSA fuzzers. All sanitizers are useful in detecting soundness issues.

2) Results for RQ2 about the Categories of the Soundness Issues and their Frequencies: To understand the types of the soundness issues, we identified 15 categories based on the keywords contained in the error summary of the issues reported by the symptom-revealing sanitizers. We ignored issues detected by NoSan, as they have no symptom-indicating error summary. The way we identified the categories is as follows. We first took a sample of the soundness issues and extracted the keywords used to describe the symptom in their error message. We then wrote a script to identify issue categories based on keyword matching for the rest of the issues and calculated the category frequencies. Anytime the keyword matching fails, we manually examined the error message of the issue to extract the keywords representing the symptom for a new category, add the keywords to the keyword set, and run the script again. 15 categories were identified in this way.

TABLE IV
CATEGORIES OF SOUNDNESS ISSUES IDENTIFIED BASED ON ERROR MESSAGE KEYWORDS, THE RELATIVE FREQUENCIES OF THESE CATEGORIES, AND THE CWE IDS.

Category ID	Error Message Keywords	Frequency(%)	Related CWE
IssueCat-1	Use of uninitialized variable	13.82	457
IssueCat-2	Stack-based buffer overflow	4.91	121
IssueCat-3	Heap-based buffer overflow	6.16	122
IssueCat-4	Global buffer overflow	3.23	120
IssueCat-5	Read memory access	29.98	125
IssueCat-6	Write memory access	10.43	787
IssueCat-7	Direct leak	2.26	401
IssueCat-8	Allocation size too big	0.78	770
IssueCat-9	Heap use after free	4.61	416
IssueCat-10	Requested allocation size exceeds maximum supported size	0.82	789
IssueCat-11	Attempting free on address not malloced	0.29	590
IssueCat-12	Negative size param	0.07	687
IssueCat-13	Strcpy param overlap	0.03	120
IssueCat-14	Memcpy param overlap	0.08	131
IssueCat-15	Runtime error	22.53	119 125 190 191 476 561 682 787 843

Table IV shows the 15 categories, their relative frequencies, and the CWE IDs mapped to them. Among the most frequent categories are IssueCat-1 (Use of uninitialized variables), IssueCat-5 (Read memory access), IssueCat-6 (Write memory access), and IssueCat-15 (Runtime error). IssueCat-5 and IssueCat-6 are generally due to out-of-bounds accesses, but they exhibit different symptoms such as "dereferenced a NULL/Zero page pointer" and "program counter (PC) at non-executable memory (wild jump)", according to their error messages. IssueCat-15 has many sub-categories. Although we did not obtain an exhaustive list of them, a sample shows that about 33% of the issues in this category are related to "Index out of bounds".

We found that the restoration-based augmentation performed by current techniques can resolve many issues of from each category. However, still many issues remain. For example, COVEH's augmentation helped reduce issues for categories 1, 3, 4, 5, 6, 7, 9, and 15. However, there are still issues unresolved for each category.

These categories of issues are also security-related. Based on the rules provided at [55], we identified the CWE IDs mapped to each category. For example, IssueCat-1, IssueCat-2, IssueCat-3, and IssueCat-9 can be mapped to CWE-457, CWE-121, CWE-122, and CWE-416, whose Likelihood of Exploit are all rated as high. IssueCat-10 can be mapped to CWE-789, which can trigger a Denial of Service (DOS) attack. The "Index out of bounds" sub-category of IssueCat-15 can be mapped to CWE-125, which can be exploited to access private data. Debloated programs with these soundness issues contain severe security threats and are vulnerable to attacks.

Finding 2: Soundness issues introduced by existing debloating techniques can be identified into 15 categories. The most common categories are related to memory read and write accesses, runtime errors, and uninitialized variables. Several categories have CWE entries, implying severe security threats.

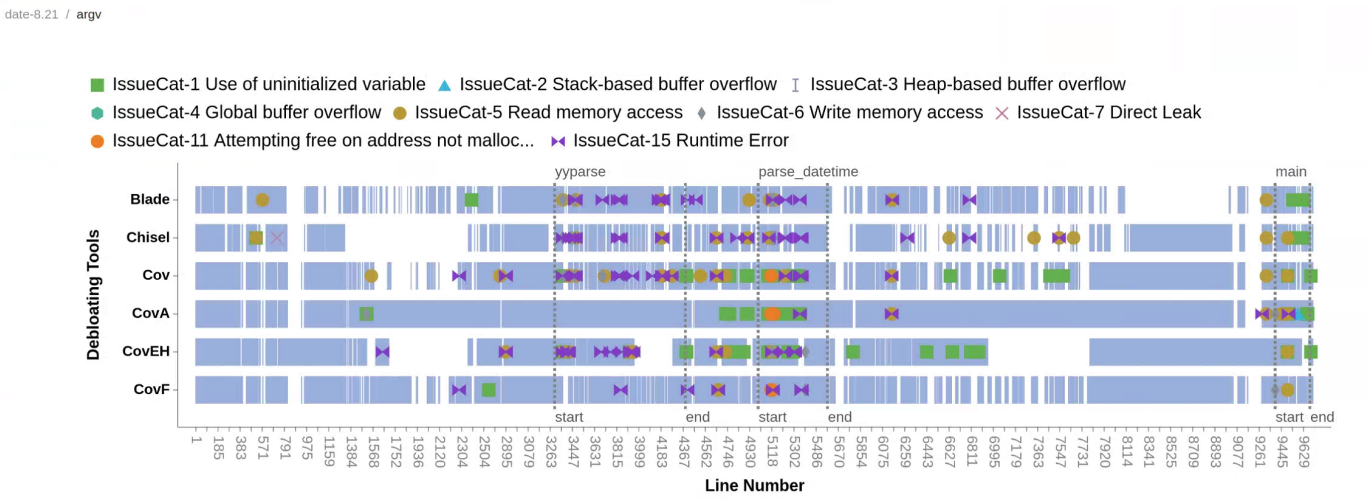

Fig. 3. Soundness issues detected in all the debloated programs of *date*.

TABLE V
PERCENTAGE OF THE FUNCTIONS CONTAINING 50% OF SOUNDNESS ISSUES.

Program	#AllFuncs	%FuncsWithHalfIssues
bzip2-1.0.5	97	2.06%
chown-8.2	115	2.61%
date-8.21	75	2.67%
grep-2.19	308	7.14%
gzip-1.2.4	91	4.39%
mkdir-5.2.1	41	7.32%
rm-8.4	127	2.36%
sort-8.16	220	1.36%
tar-1.14	473	2.32%
uniq-8.16	63	3.17%
flex-2.5.4	162	3.73%
grep-2.4.2	130	9.23%
gzip-1.3	97	4.12%
sed-4.1.5	245	2.86%
space	136	7.35%
make-3.79	248	2.82%
bash-2.05	1003	0.50%
vim-5.8	1699	0.12%
Average	**161**	**3.67%**

3) Results for RQ3 about Soundness Issue Distribution: To understand how the soundness issues are distributed in the debloated programs and the characteristics of the functions where they exist, we counted the number of soundness issues contained in each of the functions and analyzed the functions with the most issues.

Table V presents, for each program, the total number of functions and the percentage of functions containing 50% of the soundness issues detected for the debloated program via fuzzing. Our result shows that the soundness issues are not evenly distributed in the program. In fact, half of the issues occur in only 3.67% of the functions. This result implies that improving the soundness of debloating for only a few functions can help avoid many of the issues.

After analyzing the functions with the most issues, we found that they share two common characteristics: they are frequently invoked during program execution and correspond to core features, and they contain many if-branches checking various conditions.

Figure 3 depicts how the issues (in categories) are distributed in the debloated programs of *date*. The horizontal and vertical axes are the debloating techniques and the line numbers of the program respectively. The shaded areas in the figure are code retained in the debloated program, and the blank areas represent deleted code. Of all functions, *main*, *yyparse*, and *parse_datetime* contain 59.5% of the issues. *main* is the program entry point, which contains a series of if-statements parsing different options. *yyparse* and *parse_datetime* are responsible for date string parsing and time parsing. They also contain many if-branches processing strings of different types and times in different formats and zones. Deletion of conditional branches (due to the absence of inputs triggering the corresponding conditions) can allow program execution to arbitrarily pass through the empty branch without any proper setup or processing from the original branch code, leading to unexpected program state and causing soundness issues. Issue distribution figures of other programs can be found at [56].

Finding 3: Soundness issues are not evenly distributed in the debloated program. About 50% of the issues occur in only 4% of the functions. Functions with the most issues are those with many if-branches deleted.

4) RQ4: Root Causes of the Soundness Issues and Examples: While there are multiple symptoms of the soundness issues as listed in Table IV, the root cause is quite common — the inappropriate deletion of code often manifested as assignments, return statements, and if-statements checking invalid case. Due to the code deletion, the program execution with some unseen input not observed in the usage profile generates incorrect states, whose propagation along the execution can ultimately lead to crashes. In particular, the deletion of assignments can make variables hold incorrect values, causing various problems including crashes; the pruned return statements can mistakenly direct the control-flow; and the

Listing 1. Deletion of an assignment raising the uninitialized variable issue.

```
1  void set_program_name(char const *argv0) {
2      ......
3      slash = (char const *)strrchr(argv0, '/');
4      if ((unsigned long)slash != (unsigned long)((
           void *)0)) {
5        base = slash + 1;
6      } else {
7        //base = argv0;  //Deleted
8      }
9      if (base - argv0 >= 7L) {//Uninitialized base
10         ......
11     }
12     ......
13     return;
14 }
```

Listing 2. Deletion of if-statements checking invalid states leading to null-pointer deference issue.

```
1  int main(int argc, char *argv[]){
2      ......
3      bool valid_date = parse_datetime(..);
4      // if (!valid_date) { //Check of invalid dates
           deleted
5      //     error(...);
6      //     exit(1);
7      // }
8      show_date(...);
9      ......
10 }
11 void show_date(...){
12     // if (!tp) { //Check of null-pointer deleted
13     //     return ('?');
14     // }
15     int tmp = tp->value;
16     ......
17 }
```

Fig. 4. Examples of inappropriate code deletion causing soundness issues.

lack of if-statements can allow invalid cases to be unchecked, raising soundness issues.

Figure 4 presents two examples showing how inappropriate code deletion can generate faulty states that either directly raise soundness issues (Listing 1) or unexpectedly affect the control flow, resulting in unconditioned execution that exposes soundness issues (Listing 2). Both code snippets are from the program *date*. Irrelevant code is omitted for clarity.

In Listing 1, the function sets the name for the program being executed. This is done by parsing *argv0*, the path of the program binary, identifying the binary name starting with the character that comes after the last slash, and assigning the name to *base*. Because the given inputs provided in the usage profile of *date* do not include any case where *argv0* does not contain a slash (in other words, it is always the case that the full path of binary is used), line 7 is never exercised. As a result, existing tools like COV treat the line as unnecessary and delete it. Now when the debloated function is invoked with the argument *argv0* containing no slash, line 7 should have been executed, but it is not. As a consequence, *base* at line 9 is uninitialized, and MSan can report this as an issue.

Listing 2 shows that an incorrect program state can lead to unconditioned execution exhibiting soundness issues. The *parse_datetime* function at line 3 designed to validate the given date string can return false when the string is in an invalid form. Because all the date strings used in the usage profile of *date* are valid, the *valid_date* check (lines 4–7) is never executed, and existing tools delete the check. Now the debloated program can allow an unexpected execution, which should have been terminated at line 6, to continue and cover line 8. The execution causes a null-dereference at line 15 due to a similar inappropriate deletion of the if-check (lines 12–14).

Finding 4: The root cause of the soundness issues introduced by debloating is the inappropriate deletion of soundness-related code, leading to erroneous execution states that could have been avoided if the code was retained.

IV. THE BLOCKAUG APPROACH

Inspired by our previous analysis, which shows that soundness issues introduced by debloating arise from inappropriate code deletion, we propose BLOCKAUG, a blocking-based augmentation approach aiming for sound debloating. BLOCKAUG can be easily applied to coverage-based debloating techniques, which remove branches not exercised by the debloating inputs. The augmentation is performed by blocking the removed branches. The goal is to prevent any execution from passing through a removed branch so as to avoid possible erroneous program states that propagate along the execution and lead to incorrect behaviors.

Algorithm 1: BLOCKAUG algorithm.

Input : Original program P_o, debloating inputs I
Output: Augmented debloated program P_a

```
1  // Perform coverage-based debloating.
2  P_d ← debloatProgram(P_o, I)
3  // Collect all deleted branches.
4  deletedBranches ← {}
5  for condStmt ∈ P_o.getAllConditionalStmts() do
6      if P_d.contains(condStmt.condition) then
7          for branch ∈ condStmt.branches do
8              if not P_d.contains(branch) then
9                  deletedBranches.add(branch)
10             end
11         end
12         specialElse ← condStmt.nextBasicBlock
13         if not P_d.contains(specialElse) then
14             deletedBranches.add(specialElse)
15         end
16     end
17 end
18 // Augment by blocking deleted branches.
19 P_a ← copyFile(P_d)
20 for branch ∈ deletedBranches do
21     P_a.insertBlockingCode(branch)
22 end
```

Algorithm 1 details how the approach works. First, as shown in line 2, a coverage-based debloating technique (e.g.,

COV) is invoked to perform debloating based on the original program P_o and the debloating inputs I to generate a debloated program P_d. In this step, branches (e.g., if-then branches) that are deemed unneeded by the technique are removed. As a post-processing step, the technique may perform dead code elimination (DCE) to remove "dangling" conditional statements with empty branches. For example, COV with DCE can remove a dangling if-statement whose condition is covered in the execution of the debloating inputs but the then-branch is not (since the condition is never evaluated to true in the execution). BLOCKAUG requests that the DCE of the technique (if any) is turned off, as it currently needs the dangling statements to identify branches to block.

After obtaining the initial debloated program P_d, BLOCKAUG performs augmentation by blocking the empty branches to generate the augmented program P_a as the output. It starts by finding all the conditional statements (as if- and switch-statements) in P_o (line 5) and then excluding those that are entirely removed and thus are not dangling conditional statements (line 6). For each of the dangling conditional statements (lines 7–11), BLOCKAUG looks for branches that are emptied by the debloating step and saves them for the augmentation later. Note that BLOCKAUG does not have to do branch-augmentation for any fully-removed conditional statement s, as it can block any parent (empty) branches of s to avoid unexpected execution.

Going beyond searching for the typical conditional branches that are empty, BLOCKAUG also looks for a special conditional branch as the basic block bb (if any) that comes right after an if- or switch-statement s from the same branch, as shown in lines 12–15. If bb is cleaned, then some branch of s should end with a statement re-directing the control flow (e.g., a return statement). Otherwise, s and bb should both be covered in the execution. Consider for example *if (cond) { return 0; } return 1;*, where *return 0;* is a branch of the if-statement. BLOCKAUG considers *return 1;* to be a special else-branch, which is only exercised when *cond* is false.

Finally, for each empty conditional branch identified (either typical or special), to avoid the execution from passing through it, BLOCKAUG inserts two statements: a print statement and an exit statement for augmentation (lines 20–22). The former gives a warning that the branch—identified by its starting source line number—has been deleted. The latter triggers an immediate termination for any execution that reaches the branch. BLOCKAUG forbids the execution from continuing, as the branch has been cleaned, and the output of the execution, due to the lack of original branch code, cannot be trusted.

The key difference between BLOCKAUG and current augmentation approaches [11], [19] including COVEH, is that BLOCKAUG's augmentation is focused on code blocking rather than code restoration. It can actually be combined with previous approaches to have both restoration and blocking. To combine them, one first employs a coverage-based approach (for example COVEH) to prune code not covered during execution and restore the needed branches. After this, BLOCKAUG can be applied to the debloated program by blocking the empty branches cleaned by the approach.

V. IMPLEMENTATION AND EVALUATION OF THE BLOCKAUG APPROACH

We implemented the BLOCKAUG approach and applied it to the four baseline techniques, COV, COVA, COVF, and COVEH, that perform coverage-based debloating and created COV-BLOCKAUG, COVA-BLOCKAUG, COVF-BLOCKAUG, and COVEH-BLOCKAUG. In our implementation, BLOCKAUG additionally blocks the execution of any function whose body is removed. It does this to address the problems of the underlying coverage tracking tools (*gcov* and *llvm-cov*) used by the baseline techniques. Though rare, the tracking tools can fail to log the execution of a function body, which results in the incorrect function body removal. We suspect that the coverage tracking tools may miss such coverage for functions (1) that are called in sub-processes and (2) exit the program. The incorrect empty function, when called, can cause problems such as undefined behavior. Note that if coverage tracking is accurate, a coverage-based debloating technique would ensure that the callsites of a function are all removed before cleaning the function's body, in which case, BLOCKAUG does not need to block the empty body.

We assessed the soundness of the BLOCKAUG-augmented programs and compared BLOCKAUG with the restoration-based augmentation performed by COVA, COVF, and COVEH. Because there are two potential downsides of BLOCKAUG's augmentation approach with respect to the program size and the program generality, we also evaluated the size and generality of the augmented programs.

Specifically, BLOCKAUG's augmentation inserts print and exit statements into the empty branches and thus increases the program size. We compared the size reduction achieved by using both the baseline techniques and BLOCKAUG with those achieved by the baseline techniques alone (e.g., comparing COV-BLOCKAUG to COV).

The second downside is the damage to the program generality [19], which measures the extent to which the program behaves correctly for inputs not observed while debloating. The inserted exit statement can prevent the program execution from passing through a removed branch, which is often beneficial, as it is helpful to avoid unexpected program states leading to crashes and incorrect outputs. However, there are cases where the execution, if not forbidden, could have generated a correct output even though the branch is removed and the code originally in the branch is not executed. Consider for example the extra branch at the end of the loop *while(1) { l1: ... if (!tmp0) {goto while_break_1;} goto l1; }* that contains a single statement *goto l1;*. The branch only re-directs the control-flow to the start of the loop and is semantically unneeded. Blocking it can cause the execution, which could have yielded a correct output, to terminate.

A. Experimental Method and Setup

To investigate the soundness of the debloated programs after BLOCKAUG's augmentation, we followed the method described in Section III by detecting soundness issues via fuzzing, issue validation, and issue deduplication. In addition, we evaluated the size reduction achieved for each of the

TABLE VI
RESULTS OF THE FOUR COVERAGE-BASED DEBLOATING APPROACHES WITH AND WITHOUT BLOCKAUG APPLIED.

Approach	Reduction	#All-Issues	#Related-Issues	Generality	Error Rate†
COV	54.80%	93.3	47.8	45.22%	73.55%
COV-BLOCKAUG	46.03%	0.7	0.3	38.89%	0.10%
COVA	36.94%	49.3	15.0	66.09%	53.77%
COVA-BLOCKAUG	34.56%	1.3	0.5	61.94%	0.05%
COVF	49.84%	53.2	27.1	46.76%	74.50%
COVF-BLOCKAUG	44.33%	0.7	0.3	39.88%	0.10%
COVEH	46.48%	54.3	28.7	46.20%	73.60%
COVEH-BLOCKAUG	38.92%	1.4	0.3	40.18%	0.39%
Baseline Avg.	47.01%	62.5	29.7	51.07%	68.86%
Augmented Avg.	40.96%	1.0	0.4	45.22%	0.16%

† The output error rate. An output error is the incorrect output given by a debloated program. A warning message showing that the program cannot handle the input or a correct output is not considered an output error.

augmented debloated programs by measuring the fraction of LoC removed.

To assess the generality of the debloated program, we followed the previous work [19] by preparing an extra set of inputs I', comparing the execution outputs of the debloated program and the original program for each input in I', and counting the different outputs as the generality issues exposed by I'. Formally, a generality score $gen = \frac{\Sigma_{i' \in I'} p'(i)=p(i)}{|I'|}$, where $p'(\cdot)$ and $p(\cdot)$ denote the outputs of the debloated and the original programs and $|I'|$ is the number of inputs in I'. To have I', we first got the inputs I'_o from the previous evaluation [19] used to assess program generality. Note that I'_o is different from of the set of debloating inputs I used to produce debloated programs. Then, from I'_o, we identified inputs that are feature-related to the debloating inputs I to obtain I' used to evaluate program generality. We did this because the debloating approaches investigated in this paper are feature-oriented and are not designed to handle all possible inputs. Here we used the same method discussed in Section III-C (the "Feature-related soundness issues" part) to determine the feature-relatedness of two inputs and excluded *bash* and *vim* for the same reason.

For feature-related inputs, the debloated programs should give the correct outputs. For inputs that are not feature-related (i.e., feature-unrelated inputs), the debloated programs may not be able to handle them, which is fine. However, from the program trustworthiness perspective, for a feature-unrelated input, the debloated program should not give a wrong result to mislead the user but should instead warn the user that the program may not behave correctly for the input and avoid providing any output. BLOCKAUG can help achieve program trustworthiness. To investigate this, we evaluated the *output error rate* of the debloated programs using the feature-unrelated inputs. This is to assess the ability of the program to reject inputs that it should not handle. Here we consider as an *output error* the incorrect output given by a debloated program. A warning that the program cannot handle the input or a correct output is not considered an output error.

B. Results

Table VI presents our results in terms of the size reduction (measured by fraction of LoC) achieved by each approach, the soundness issues (all and feature-related) it introduced, and the generality and output error rate of the debloated programs it produced. Because the BLOCKAUG approach is applied to the four baseline debloating techniques, COV, COVA, COVF, and COVEH, we also present the results of these techniques for comparison. The results can help us assess the effectiveness of the BLOCKAUG's augmentation.

Our results show that BLOCKAUG is highly effective at improving program soundness. According to the last row of column #All-Issues, the BLOCKAUG-augmented programs have one soundness issue, which is significantly lower than the 62.5 issues of the debloated aprograms generated by the baseline techniques. Also, the number is considerably lower than issues introduced by CHISEL and BLADE (70 or more) as presented in Table I.

BLOCKAUG can successfully resolve many categories of issues listed in Table IV. To have the statistics, we obtained the numbers of issues for each category before and after BLOCKAUG is applied and computed the reduction rate. We found that BLOCKAUG successfully resolved all issues for 9 of the 15 categories. For the remaining 6 categories (IssueCat-1, 4, 5, 6, 7, and 15), the average reduction rate is as high as 91.43%. Detailed results can be found at [57]. Note that for the statistics, we excluded BLADE's and CHISEL's results, because BLOCKAUG does not apply to the two tools.

```
1 while (*ptr) {
2     ret[rc] = ptr; //Out-of-bounds index reference for
          ret, a 1000-dimension array
3     while (*ptr && (*ptr !='|') ) ptr++;
4     *ptr = '\0';
5     ptr++;
6     while (*ptr && (*ptr =='|') ) ptr++;
7     rc++;
8 }
```

Fig. 5. Vulnerable code in argv-fuzz-inl.h for program instrumentation.

BLOCKAUG does not eliminate all issues. The unresolved issues are not related to the BLOCKAUG approach per se, but

due to (1) the vulnerability of the AFL++'s instrumentation for fuzzing and the inconsistent error messages reported by the UBSan sanitizer for the original and debloated programs, leading to an incorrect analysis of the cause of the soundness issue and (2) the failure of the underlying code coverage tools that BLOCKAUG uses.

Specifically, for (1), in the *argv-fuzz* experiments (Section III-C), we slightly modified the original and debloated programs by introducing code referencing the header file *argv-fuzz-inl.h*, which is a file associated with the AFL++ tool (not created by us), for program instrumentation. Using the header file, the program allows for passing the fuzzed inputs into the *argv* array for execution. Unfortunately, the header code does not have a bounds check for the `ret` array shown in the Figure V-B. As a result, when a fuzzed input has too many arguments (over 1000), `ret[rc] = ptr` causes a crash due to an out-of-bounds index reference. Normally, the original and debloated programs will expose the same crash, which does not count as an issue introduced by debloating. However, in some cases where UBSan is used as the sanitizer for fuzzing, the crashes for original and debloated programs are not the same (reflecting a weird behavior of UBSan) — for the debloated program, the error message indicates an out-of-bounds index reference, and the program exit code is somewhere between 130 and 139; for the original program, however, the error message only suggests a general runtime failure, and the exit code is -1. With the misleading disparity, our issue checker mistakenly considers that the crash is not due to an inherent weakness shared by the original and debloated programs but an issue introduced by debloating, which is not the case.

For (2), some unresolved issues are due to the weakness of the underlying code coverage tools in accurately tracking and deleting code. Though rare, the COV tool may fail to remove a branch B not covered by any execution due to incomplete coverage tracking. We saw this happen for the special branch (described in Section IV) that comes after an if-statement containing a control-flow re-directing statement, such as B in `if (cond) { goto l; } B`. When the unneeded branch B is left untouched or not fully cleaned, BLOCKAUG does not block B, and as a result, an execution falsely passing through B can make the program crash.

BLOCKAUG does not achieve a non-zero error rate also for two reasons. The first reason is the weakness of the underlying code coverage tools as just explained. The second reason is BLOCKAUG's imperfect support for graceful exit — some necessary cleanup work before exit is left undone. For the *bash* program, an ungraceful exit can corrupt the environment, making the next run crash. While the problem of the coverage tools' inaccurate code tracking may be fixed, graceful exit of blocking cannot be guaranteed, since not all changes before exit are reversible. Therefore, a zero error rate of BLOCKAUG in all cases is not feasible.

Our results also show that the proposed augmentation strategy only slightly increase LoC (with respect to the program generated by each baseline approach). The augmentation causes only a 6.05% decrease in size reduction.

Depending on the aggressiveness of the debloating techniques, the four baseline approaches can produce programs that expose generality issues of varying severity. That is, their debloated programs fail to handle a portion of the inputs that are related to the user-specified features. Although BLOCKAUG can create more generality issues, the empirical influence is very limited. On average, the augmented programs only increase the number of generality issues by 5.85% (from 45.22% to 51.07%) compared to their baseline counterparts.

The increase in generality issues is due to the fact that the augmentation of BLOCKAUG may cause the program to exit prematurely when it could have produced a correct output. This can happen because a branch b may be semantically unneeded with respect to the execution of the given input or its deletion, though leading to erroneous states generated and propagated along the execution, somehow does not affect the final output. If b is blocked, however, the execution will surely yield an incorrect output, as it terminates before generating any output.

At the cost of a small increase of generality issues, the augmentation of BLOCKAUG not only effectively improves soundness, but also produces debloated programs with more credible outputs. The debloated programs generated by the four baseline approaches have an output error rate of 68.86%. This result shows that the programs give wrong results for about 70% of the feature-unrelated inputs for which the programs should not have handled, creating severe trust issues. In contrast, the augmented programs generated by BLOCKAUG can terminate the execution with an error message indicating the program's inability to handle the input. The output error rate is as low as 0.16%. It is not 0, as we found that there are branches that are not correctly blocked by BLOCKAUG due to the failure of underlying coverage tracking tools (*gcov* and *llvm-cov*) in accurately capturing the executed branches. Overall, the ability of blocking unintended execution offers benefits as it makes the programs augmented by BLOCKAUG more trustworthy and reliable to use.

Our results show that the restoration-based augmentation approaches performed by COVF, COVA, and COVEH cannot fundamentally solve the soundness problem of feature-oriented debloating. BLOCKAUG on the other hand is much more effective in improving the soundness. Using different baseline approaches for debloating, BLOCKAUG can achieve similar soundness improvement but different size reduction and generality results. COVF's and COVEH's augmentations address soundness issues. Pairing BLOCKAUG with these techniques can improve soundness while making the program behave correctly for more invalid inputs (rather than simply forbidding the wrong execution). COVA, on the other hand, is designed to improve generality in handling more feature-related inputs (achieving a 66.09% generality compared to 45.22% of COV). Pairing COVA with BLOCKAUG can lead to effective improvement in both soundness and generality.

VI. DISCUSSION

Unsoundness of current input-based techniques. Previous debloating approaches are highly unsound. They tend to pursue high size reduction and can remove key soundness-related

code such as the "defensive" if-checks and variable assignments, introducing severe correctness and security issues. Although approaches like COVF and COVA use fuzzing and heuristics to identify and restore deleted code that affects program soundness or that corresponds to the needed features specified, their effectiveness in terms of soundness improvement is limited, as evidenced by our results. The alternative approach, COVEH, which uses purely static methods to detect code for restoration cannot avoid soundness issues either. It is still possible to further extend COVEH's approach by identifying and restoring more types of code for soundness improvement. However, many issues can still exist, and the soundness improvement can come with a non-trivial program size increase. To test this, we extended COVEH to identify any original branch that exclusively contains assignments or return statements (their deletion is identified as related to the root cause of soundness problems in Section III-G4) for restoration and evaluated the extended approach on the 10 utility programs from the CHISELBENCH benchmark. Our results show that the extended approach achieves slightly better soundness (average issue count drops by 5.3%) but at the cost of a lower reduction (reduction rate drops by 10.9%). Based on the results and analyses, code restoration alone cannot fundamentally resolve the soundness challenges faced by feature-oriented debloating.

BLOCKAUG's role. BLOCKAUG is not another code restoring approach that identifies and preserves additional code to handle more unseen inputs. Rather, it is designed to forbid the execution with inputs that the program is unable to handle. As such, it is unfair to directly compare BLOCKAUG with current techniques like COVF addressing soundness issues. The role of BLOCKAUG is to complement existing coverage-based techniques to improve their soundness.

Strengths and weaknesses of BLOCKAUG. BLOCKAUG's blocking method provides a new approach to improving the soundness of debloating. Its advantage over COVEH and other restoration-based techniques stems from the blocking mechanism, which, unlike code restoration, prevents any unexpected execution from passing through a deleted branch — effectively forbidding execution of original code that was removed but is still required. As a result, it is very effective at preventing unsound behavior. Although the augmentation induced by the blocking method adds new code (for warning generation and execution termination), the actual size increase is only slight. This is in contrast with the code restoration methods, which can increase program size by for example up to 18% (COVA). Another key benefit of the blocking method is that it can forbid the execution of any input for which a debloated program would have generated an incorrect output. Instead of blindly reporting a wrong output, the block-augmented program can give warnings for potential mistakes and terminate the execution, enhancing the trustworthiness of debloating.

The soundness improvement brought by the block augmentation does not come at no cost. Our results showed that it may adversely affect the generality. We nevertheless note that the actual generality loss caused by block augmentation is only about 6% and that the loss can be further mitigated with better strategies for code restoration, which is orthogonal to this work. Overall, the benefits of the block augmentation outweigh its weaknesses. We thus advocate adopting block augmentation for feature-oriented debloating to improve soundness, while also encouraging the development of complementary code restoration strategies to further enhance program generality.

Applicability of BLOCKAUG. BLOCKAUG is particularly designed to enhance the soundness of input-based debloating that prunes code in the form of complete branches, driven by coverage. We have tested the method over all the source-based techniques performing coverage-based code pruning (with and without code restoration) and demonstrated its effectiveness. While we did not implement a block-augmented version for RAZOR, which realizes coverage-based code pruning at the binary level, due to the technical issues and challenges we faced while using the tool (as discussed in Section III-G1), we believe the method could be nevertheless applied to a binary-based approach like RAZOR by identifying and augmenting deleted instructions corresponding to the deleted branches at the source level. We leave as future work the investigation of applying block augmentation to binary-based approaches.

Unlike coverage-based techniques, more aggressive delta-debugging-based techniques like CHISEL can delete single statements within a code block, as long as the deletion does not affect the execution result for the debloating inputs. BLOCKAUG cannot trivially augment any reduced block resulting from delta-debugging-based code pruning. This is because forbidding any execution from passing through a code block with deleted code can prevent the program from processing any inputs and make it useless (consider forbiding the execution from within the top-level block of main). That said, it is still possible to use BLOCKAUG's approach to augment a fully reduced code block. However, in that case, additional check is needed to ensure that the augmented program still behaves correctly for the debloating inputs.

BLOCKAUG also does not work in the configuration-based debloating scenario assumed by OCCAM and TRIMMER. Since OCCAM and TRIMMER are supposed to produce sound programs tailored for a given configuration, one need not apply BLOCKAUG to improve the soundness of OCCAM's or TRIMMER's debloating result. That said, one could still attempt to run BLOCKAUG on OCCAM's or TRIMMER's debloated program by identifying any removed branch (in the form of a dangling branch with an empty body) and blocking it, even though dangling branches in OCCAM's and TRIMMER's debloated programs are not common, as they are often optimized away (unless the conditionals have side effects). When applied, BLOCKAUG's approach would either have no effect or break the program's functionality, which is harmful. For a dangling branch whose body is never executed by any inputs under the specified configuration, blocking it has no effect. However, if the body of a dangling branch can be covered, blocking the branch can break the program's intended functionality: the program can fail on inputs whose execution could have passed through the branch body without doing anything yet still produce a correct result under that configuration. In such cases, blocking the branch will cause the execution to fail.

BLOCKAUG's blocking mechanism. While we have im-

plemented the blocking method by inserting exit statements at the deleted branches for exploration purposes, we note that the blocking mechanism can be realized in other ways by for example re-directing any unintended execution to an upper-level service or rolling it back to a previous checkpoint instead of just killing it.

BLOCKAUG's support for graceful exit. We also note that a blocking method that inserts exit statements to forbid unintended execution does not necessarily lead to a graceful exit [58] due to the lack of cleanup work that may be needed to revert what has been done up to the exit point. Although the OS can automatically release some resources for cleanup by for example reclaiming the allocated memory after the program exits, there will be changes (such as file updates) that are not revertible by the OS. For example, if a debloated *bzip2* exits before completing the compression of a file, the halfway-compressed file may be left in the output directory. To address the issue, we developed a method [59] that goes with the block-augmentation for file-change-related cleanup by intercepting system calls to perform backup on every resource-related operation (e.g., file creation and modification) and rolling back the changes when the program exits for blocking purpose. We do not however claim that these operations are sufficient to revert all kinds of changes (in fact, not all changes are revertible). Adding more operations to deal with more types of changes will be left as future engineering work. As an alternative, one can just rely on a system's backup-and-restoration mechanism (for example the time machine of the Mac OS) for cleanup, which is however not lightweight.

A soundness investigation of compiler-optimization-based techniques. While the study is centered around input-based techniques, we also conducted an experiment investigating another important type of feature-oriented approaches that leverage compiler optimization for debloating. In this experiment, we chose OCCAM as an exemplary compiler-based technique [22], [23], and ran it to generate debloated programs for the 10 utilities. Since OCCAM is configuration-based, for debloating, we chose for each utility a specific configuration (specified by one option or a combination of options used in any of the debloating inputs) and ran the OCCAM tool [60] to generate a debloated binary. We evaluated the soundness of the debloated binary with the Radamsa fuzzing tool. In the fuzzing process, we used as seeds the debloating inputs that conform to the configuration (i.e., use the same options specified in the configuration) and generated 500 fuzzed inputs for each seed, following the fuzzing method in our study. For comparison, we used COVA, which has the best soundness among the studied tools, and COVA-BLOCKAUG, the block-augmented version of COVA, to produce debloated programs. To ensure fairness, debloating was performed on inputs conforming to OCCAM's configuration, and the same fuzzed inputs were used for evaluation.

Our results show that COVA-BLOCKAUG and OCCAM are highly sound, with the crash rates being as low as 0% and 1% respectively. Comparatively, COVA's crash rate is much higher, 31%. Although compiler-based techniques like OCCAM are generally considered sound, according to our results, the crash rate is not strictly 0%. We looked into one case where the debloated binary for gzip crashed, and found that the debloated *huft_build*, a helper function for decompression, caused pointer out-of-bounds access. Detailed results can be accessed at [61].

VII. THREATS TO VALIDITY AND LIMITATIONS

In this section, we discuss the external, internal, and contruct threats to validity.

External threats. First, we used 18 programs from two established benchmarks (CHISELBENCH [27] and LSIR [19], [41]) for our study and the evaluation of BLOCKAUG. We chose these two benchmarks as they have been widely used to evaluate existing debloating approaches [10], [19], [62]–[64], and all the tools used in our study can perform debloating without errors for the programs. Although our results and findings may potentially not generalize to more programs, we believe they have already revealed the unsound nature of current debloating techniques and provided useful insights towards improving the soundness of these techniques. Second, we investigated 7 tools that rely on input specification for feature-oriented debloating. The results and findings may not generalize to feature-oriented techniques that use other forms of specification or other types of debloating techniques. We nevertheless note that feature-oriented debloating that uses inputs for specification is widely applied and thus studying such techniques is of significance. Investigating more debloating approaches will be left as future work.

Internal threats. The internal threats stem from the potential implementation errors of COVEH and BLOCKAUG. We have carefully tested the implementations of both tools, and performed internal code review. We also made the tools and the results available for public review and verification.

Limitations. In our study, we used a dynamic method employing fuzzers and sanitizers to detect soundness issues exposed as crashes. A limitation is that such a dynamic method is insufficient to identify all sorts of issues. It is possible to use other methods such as static analysis and formal verification to detect other forms of defects revealing soundness issues. We did not consider such methods in the study, as they suffer from scalability limitations and false positives. Comparatively, a dynamic analysis with fuzzers and sanitizers offer a scalable and effective means to detect unsoundness where the ground truth is hard to establish.

As elaborated in Section VI, our proposed BLOCKAUG has limitations in that (1) its blocking augmentation can harm program generality; (2) it is restricted to input-based techniques that employ a coverage-based strategy for debloating; and (3) its blocking mechanism can lead to ungraceful exit. We nevertheless note that the generality loss due to BLOCKAUG's augmentation is empirically small. Also, because coverage-based debloating techniques are often more efficient and effective (in terms of the reduction-generality tradeoff) than delta-debugging-based techniques for complex program debloating [19], [62], the applicability of BLOCKAUG is promising. Finally, ungraceful exit cannot be fundamentally avoided but can be mitigated with more engineering work, which we consider to do in the future.

VIII. Related Work

Studies on feature-oriented software debloating. Related to our research are four studies that compare and evaluate existing feature-oriented debloating techniques. The study conducted by Brown et al. [62] investigated various techniques and used a variety of metrics related to performance, security, and correctness. In our research, we focused on program soundness. Our study is more comprehensive for soundness evaluation in that three fuzzers of different types paired with multiple sanitizers are used to detect a variety of soundness issues reported as crashes. We analyzed the issues and provided insights into their categorization, distribution, and root causes.

Xin et al. [19] did experiments to study the reduction-generality tradeoffs achieved by current techniques. Their study is designed primarily to understand the extent to which the debloated programs can handle features not exercised by the given inputs, while ours is centered around program soundness. Although Xin et al. also did a fuzzing experiment to test the robustness of the debloated programs, only two fuzzing tools Radamsa and AFL were considered, a limited time budget was used (for example, AFL was only run for 30 minutes for each trial), and no sanitizers were involved. Also, Xin et al. did not identify issues that were only introduced by the debloating tools, and for this reason, the issues may not be debloating-related.

Studies by Ali et al. [63] and Brown et al. [64] also compared current debloating techniques. However, the former only evaluated correctness of debloated programs by existing test cases and did not run fuzzers to test soundness. The latter is security-oriented and evaluated debloating techniques by the number of various gadgets reduced. Because the two studies do not target program soundness, they are different from our research.

There are debloating tools investigated in the above studies that we did not include in our research. TRIMMER [17], [44], OCCAM [22], [23], LMCAS [45], Libfilter [65], and Piece-wise [66] were not included because they are not input-based debloating techniques, and thus are not compatible with our experimental setup which uses the same set of inputs for all debloating tools to ensure a fair comparison. Chisel-GT [10], [62] and GTIRB Binary Reduce [62], [67] were not included because they are proprietary tools developed by GrammaTech [68] and Trail of Bits [69] respectively and are not publicly available. BinRec-ToB [70] was not included because it failed to debloat any of the benchmark programs, according to Brown et al. [62].

Software debloating. Software debloating techniques aim to reduce size and improve security and performance of the program by removing unnecessary code. Feature-oriented debloating takes the original program and a feature specification as input, and removes code that is not related to the specified features. There are multiple ways to specify features, among which input-based debloating is widely applicable, as it only requires a set of inputs (test cases) as specification. Input-based debloating tools can use delta debugging (e.g., CHISEL [10], BLADE [20], Perses [71], C-Reduce [72], and J-Reduce [73]), code coverage (e.g., COVA, COVF [19], and RAZOR [11]), deep learning (e.g., Hecate [13]), stochastic optimization (e.g., Debop [14]), or other approaches that for example combine static and dynamic analyses (e.g., JShrink [16]) to identify and remove unnecessary code based on the specification.

Similar to COVEH and BLOCKAUG, COVF and COVA [19] also perform augmentation based on COV. Although COVF shares a similar goal with COVEH and BLOCKAUG in improving soundness, it differs in that it uses fuzzing, a dynamic method, to enrich the input set for feature specification and debloating, whereas COVEH uses a purely static analysis to restore soundness-related code. BLOCKAUG is significantly different in that it tackles the root cause of soundness issues by blocking the execution of removed branches. COVA, on the other hand, was designed with a different goal, which is to improve generality.

In addition to input-based specification, there are other types of feature specification based on for example static configuration [17], [22], [23], [44], [45], human-developed domain sampling [40], program logs [21], android activities [18], and manual annotations of feature mapping [12]. Beyond feature-oriented debloating techniques, there are others that do not require explicit specifications such as debloating techniques that remove unused libraries [46], [65], [66], [74], but they are not the focus of this paper.

IX. Conclusion and Future Work

Current feature-oriented debloating techniques are highly unsound, as evidenced by our study using various fuzzers and sanitizers to extensively test the debloated programs produced by these techniques. We found that these techniques introduce many soundness issues that can lead to program crashes. To address this challenge, we developed BLOCKAUG, a blocking method applicable to coverage-based debloating. BLOCKAUG augments deleted branches with exit statements to prevent unintended execution and the resulting failures. Our evaluation shows that block augmentation is highly effective at mitigating soundness issues and reducing erroneous outputs in debloated programs.

For future work, we plan to extend block augmentation to a broader range of debloating techniques, including feature-oriented approaches at the binary level and potentially non-feature-oriented techniques. We also aim to explore non-exit-based augmentations and additional change-reverting operations to enhance the method's practicality. Furthermore, we hope to develop more effective code restoration strategies that complement the blocking mechanism, improving program generality while maintaining high soundness.

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] C. Hibbs, S. Jewett, and M. Sullivan, *The art of lean software development: a practical and incremental approach*. "O'Reilly Media, Inc.", 2009.

[2] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "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)*, 2017, pp. 65–70.

[3] S. Bhattacharya, K. Gopinath, K. Rajamani, and M. Gupta, "Software bloat and wasted joules: Is modularity a hurdle to green software?" *Computer*, pp. 97–101, 2011.

[4] J. McGrenere and G. Moore, "Are we all in the same "bloat"?" in *Graphics interface*, 2000, pp. 187–196.

[5] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "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, 2010, pp. 421–426.

[6] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta, "The interplay of software bloat, hardware energy proportionality and system bottlenecks," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, 2011, pp. 1–5.

[7] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "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, 2009, pp. 419–430.

[8] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: quantifying the security benefits of debloating web applications," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1697–1714.

[9] H. Shacham *et al.*, "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, 2007, pp. 552–561.

[10] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 380–394.

[11] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "Razor: A framework for post-deployment software debloating," in *Proceedings of the 28th USENIX Conference on Security Symposium (USENIX Security)*, 2019, pp. 1733–1750.

[12] M. D. Brown and S. Pande, "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*, 2019, pp. 1–7.

[13] H. Xue, Y. Chen, G. Venkataramani, and T. Lan, "Hecate: Automated customization of program and communication features to reduce attack surfaces," in *Security and Privacy in Communication Networks: 15th EAI International Conference, SecureComm 2019, Orlando, FL, USA, October 23–25, 2019, Proceedings, Part II 15*. Springer, 2019, pp. 305–319.

[14] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Program debloating via stochastic optimization," in *2020 IEEE/ACM 42st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2020.

[15] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, "Slimium: Debloating the chromium browser with feature subsetting," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 461–476.

[16] B. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "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)*, 2020.

[17] A. A. Ahmad, A. R. Noor, H. Sharif, U. Hameed, S. Asif, M. Anwar, A. Gehani, F. Zaffar, and J. H. Siddiqui, "Trimmer: an automated system for configuration-based software debloating," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3485–3505, 2021.

[18] Y. Tang, H. Zhou, X. Luo, T. Chen, H. Wang, Z. Xu, and Y. Cai, "Xdebloat: Towards automated feature-oriented app debloating," *IEEE Transactions on Software Engineering*, 2021.

[19] Q. Xin, Q. Zhang, and A. Orso, "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*, 2022, pp. 1–13.

[20] M. Ali, R. Habib, A. Gehani, S. Rahaman, and Z. Uzmi, "Blade: Towards scalable source code debloating," in *2023 IEEE Secure Development Conference (SecDev)*. IEEE, 2023, pp. 75–87.

[21] J. Liu, Z. Zhang, X. Hu, F. Thung, S. Maoz, D. Gao, E. Toch, Z. Zhao, and D. Lo, "Minimon: Minimizing android applications with intelligent monitoring-based debloating," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[22] G. Malecha, A. Gehani, and N. Shankar, "Automated software winnowing," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 1504–1511.

[23] J. A. Navas and A. Gehani, "Occam-v2: Combining static and dynamic analysis for effective and efficient whole-program specialization," *Commun. ACM*, vol. 66, no. 4, p. 40–47, 2023.

[24] A. Zeller, "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 (ESEC/FSE)*. Springer, 1999, pp. 253–267.

[25] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th international conference on Software engineering (ICSE)*, 2006, pp. 142–151.

[26] H. Liu, Z. Jia, S. Li, Y. Lei, Y. Yu, Y. Jiang, X. Mao, and X. Liao, "Cut to the chase: An error-oriented approach to detect error-handling bugs," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660787

[27] (2024) Chiselbench. [Online]. Available: https://github.com/aspire-project/chisel-bench

[28] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[29] A. Helin. (2024) Radamsa. [Online]. Available: https://gitlab.com/akihe/radamsa

[30] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[31] (2024) Addresssanitizer. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

[32] (2024) Undefinedbehaviorsanitizer. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[33] (2024) Memorysanitizer. [Online]. Available: https://clang.llvm.org/docs/MemorySanitizer.html

[34] (2024) Threadsanitizer. [Online]. Available: https://clang.llvm.org/docs/ThreadSanitizer.html

[35] (2024) Leaksanitizer. [Online]. Available: https://clang.llvm.org/docs/LeakSanitizer.html

[36] (2024) Studying and improving the soundness of debloating. [Online]. Available: https://github.com/the-Soundness-of-Debloating/debloating_soundness_study_and_improving

[37] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, p. 183–200, Feb. 2002. [Online]. Available: https://doi.org/10.1109/32.988498

[38] (2024) Evaluation of coveh-extend and the influence of coveh threshold. [Online]. Available: https://github.com/the-Soundness-of-Debloating/deb-soundnessIssue/blob/main/major_revision/compare_cov-eh.md

[39] (2024) Chisel. [Online]. Available: https://github.com/aspire-project/chisel

[40] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Subdomain-based generality-aware debloating," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 224–236.

[41] (2024) Software-artifact infrastructure repository. [Online]. Available: https://sir.csc.ncsu.edu/portal/index.php

[42] (2024) Blade. [Online]. Available: https://github.com/pawnsac/BLADE-deb

[43] (2024) Cov, covf, and cova from the debloating study repository. [Online]. Available: https://github.com/qixin5/debloating_study

[44] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 329–339.

[45] M. Alhanahnah, R. Jain, V. Rastogi, S. Jha, and T. Reps, "Lightweight, multi-stage, compiler-assisted application specialization," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 251–269.

[46] C. Porter, G. Mururu, P. Barua, and S. Pande, "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)*, 2020, pp. 164–180.

[47] (2024) Clang. [Online]. Available: https://clang.llvm.org/

[48] (2024) Afl++ argv_fuzzing. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/argv_fuzzing

[49] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2693–2707.

[50] Y. Liu and W. Meng, "Dsfuzz: Detecting deep state bugs with dependent state exploration," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1242–1256.

[51] Y. Zhang, C. Pang, S. Nagy, X. Chen, and J. Xu, "Profile-guided system optimizations for accelerated greybox fuzzing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1257–1271.

[52] P. Deng, Z. Yang, L. Zhang, G. Yang, W. Hong, Y. Zhang, and M. Yang, "Nestfuzz: Enhancing fuzzing with comprehensive understanding of input processing logic," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1272–1286.

[53] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2289–2306.

[54] (2025) Box plots of issues. [Online]. Available: https://github.com/the-Soundness-of-Debloating/deb-soundnessIssue/tree/main/scripts_data/issue_data

[55] (2024) CWE mapping rules and tips. [Online]. Available: https://cwe.mitre.org/documents/cwe_usage/quick_tips.html

[56] (2024) The soundness distribution charts. [Online]. Available: https://github.com/the-Soundness-of-Debloating/deb-soundnessIssue-distribution-chart/tree/main/public

[57] (2025) Categories of issues reduced by blockaug. [Online]. Available: https://github.com/the-Soundness-of-Debloating/deb-soundnessIssue/blob/main/minor_revision/blockaug_effectiveness.md

[58] Z. Jia, S. Li, T. Yu, X. Liao, and J. Wang, "Automatically detecting missing cleanup for ungraceful exits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 751–762. [Online]. Available: https://doi.org/10.1145/3338906.3338938

[59] (2024) The auto cleanup debloating analysis tool. [Online]. Available: https://github.com/the-Soundness-of-Debloating/debloating_analysis_tools/tree/main/auto_cleanup

[60] (2024) Occam. [Online]. Available: https://github.com/ashish-gehani/OCCAM

[61] (2024) Evaluation of occam soundness. [Online]. Available: https://github.com/the-Soundness-of-Debloating/deb-soundnessIssue/blob/main/major_revision/compare_occam_result

[62] M. D. Brown, A. Meily, B. Fairservice, A. Sood, J. Dorn, E. Kilmer, and R. Eytchison, "Sok: A broad comparative evaluation of software debloating tools," *arXiv preprint arXiv:2312.13274*, 2023.

[63] M. Ali, M. Muzammil, F. Karim, A. Naeem, R. Haroon, M. Haris, H. Nadeem, W. Sabir, F. Shaon, F. Zaffar *et al.*, "Sok: A tale of reduction, security, and correctness-evaluating program debloating paradigms and their compositions." ESORICS, 2023.

[64] M. D. Brown and S. Pande, "Is less really more? towards better metrics for measuring security improvements realized through software debloating," in *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2019.

[65] I. Agadakos, N. Demarinis, D. Jin, K. Williams-King, J. Alfajardo, B. Shteinfeld, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Large-scale debloating of binary shared libraries," *Digital Threats*, vol. 1, no. 4, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3414997

[66] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018, pp. 869–886.

[67] (2022) Binary reduction. [Online]. Available: https://grammatech.github.io/prj/binary-reduce/

[68] (2024) Grammatech. [Online]. Available: https://www.grammatech.com/

[69] (2024) Trail of bits. [Online]. Available: https://www.trailofbits.com/

[70] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz, "Binrec: dynamic binary lifting and recompilation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387550

[71] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 361–371.

[72] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 335–346.

[73] C. Gram Kalhauge and J. Palsberg, "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, 2019, pp. 556–566.

[74] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 70–83.