BLADE: Scalable Source Code Debloating Framework

Muaz Ali
University of Arizona
muaz@arizona.edu

Rumaisa Habib
LUMS
23100061@lums.edu.pk

Ashish Gehani
SRI
ashish.gehani@sri.com

Sazzadur Rahaman
University of Arizona
sazz@cs.arizona.edu

Zartash Uzmi
LUMS
zartash@lums.edu.pk

***Abstract*—Existing source code debloaters fall short due to low scalability and high runtime overhead when applied in dynamic cloud settings, where instances are spun up on the fly. To address this challenge, we propose BLADE that leverages the common coding idioms and language restrictions to build simple yet effective heuristics for faster source-code debloating. For example, usually, coding constructs are *defined* before *used*. Thus, the probability of breaking code after the removal of a node reduces with the depth of its position in the syntax tree. Also, while debloating certain functionalities, statements from a basic block have a higher possibility of getting removed together. To utilize these insights, BLADE employs a hierarchical source code reduction, where reduction candidates are chosen with reverse pre-order traversal, so that it removes *uses* before the *definitions*. Low runtime overhead makes BLADE practical to apply code debloating to large workloads. Our evaluation shows that BLADE runs faster than existing source code debloating tools. Compared to Chisel, BLADE is, on average, 2.3× faster and provides comparable reductions in the code size and attack surfaces. In comparison to Debop, another source code debloater, BLADE , on average, is 2.75× faster.**

***Keywords*—Source Code Debloating, Program Debloating**

I. INTRODUCTION

Modern operating systems, productivity software, and utility programs avoid source code duplication by maintaining the source of each component within its own self-contained codebase. This decoupling brings scalability to the software development processes while supporting an ever-increasing variety of hardware configurations. The *generality* of software leaves it *bloated* — that is, inclusive of code that may offer functionality that is unnecessary and thus presents a larger attack surface for adversaries. Security researchers and practitioners, therefore, consider debloating software before its deployment to reduce the attack surface [1], [2]. Furthermore, bloated software may unnecessarily load dynamic libraries that remain unused. It may also consume CPU and memory resources on resource-constrained embedded devices, IoTs, and mid- to low-end smartphones [3].

For software debloating to be used in dynamic cloud environments, where new instances of applications are spun up on the fly – the analysis should run faster. In *continuous integration and continuous delivery (CI/CD)* setups too, there is a requirement to supply updated builds on the fly. This requires a fast and scalable debloating approach without compromising generality. Another important aspect of effective debloating is *auditability*. The property is crucial because it allows the resultant code after debloating to be inspected for any unintended side effects. Debloaters that do not operate at the source code level [2], [4]–[6] are not favorable for this use case.

Recent schemes for debloating source code follow a simple working principle: (1) First, create a test specification to describe the desired functionality. (2) Next, *iteratively* select a *candidate set* of code statements for removal from the codebase. (3) Test the remaining code against the specification. If it passes, permanently remove (debloat) the selected statements. Then continue with the remaining code to search for additional debloating opportunities. If it fails, put those selected statements back in the codebase and consider a new candidate set of statements for removal, until all sets are exhausted. Clearly, an exponential [1] number of iterations are needed if the candidate set of statements to be removed is selected via exhaustive search.

Contemporary source code debloating tools take a long time to finish and occasionally timeout [1], [7]–[9]. Their debloating approaches routinely employ delta debugging [10] under the hood: they start with a large candidate statement set (which is removed and the remaining code is tested against the specification); the candidate statement set is repeatedly reduced by half as the search progresses, resulting in $\mathcal{O}(n^2)$ computational complexity [10]. This can result in prohibitively high running time if the number of statements in the code is large (on the order of tens or hundreds of thousands). Debop [7] uses stochastic optimization to refine the search space of the statement set. However, Debop's reduction efficiency is significantly lower than its existing counterparts. Thus existing source code debloaters are inadequate to meet the demand for dynamic deployment settings.

To address this gap, we propose BLADE . BLADE aims to support fast and scalable debloating. To achieve fast debloating times, BLADE employs a structure-aware [2] debloating process that runs in $\mathcal{O}(n)$ time (where a program has n statements). This makes it practical to use BLADE for debloating large codebases (i.e., nginx, sqlite3). Additionally, BLADE works at the source code level to enable auditability.

[1]If each of the n statements can either be IN or OUT, there would be 2^n combinations to be tested for the optimal solution.

[2]By this, we mean that BLADE recognizes program constructs such as define-before-use, loops, functions, and conditional statements and accordingly takes decisions.

```
char *md5Sum(char *data){...};                                  //st1
char *sha1Sum(char *data){...};                                 //st2
char *sha256Sum(char *data){...};                               //st3
char *sha512Sum(char *data){...};                               //st4

void hashSelector(char *selection, char *data) {
  if (!strcmp(selection, "md5")) return md5Sum(data);           //st5
  if (!strcmp(selection, "sha1")) return sha1Sum(data);         //st6
  if (!strcmp(selection, "sha256")) return sha256Sum(data);     //st7
  if (!strcmp(selection, "sha512")) return sha512Sum(data);     //st8
  }
```

A hashing program that can generate hashes using multiple algorithms.

Iterations	Program State								Oracle
0	st1	st2	st3	st4	st5	st6	st7	st8	✓
1	st1	st2	st3	st4	st5	st6	st7	st8	✗
2	st1	st2	st3	st4	st5	st6	st7	st8	✗
3	st1	st2	st3	st4	st5	st6	st7	st8	✗
4	st1	st2	st3	st4	st5	st6	st7	st8	✗
5	st1	st2	st3	st4	st5	st6	st7	st8	✓
6	st1	st2	st3	st4	st5	st6	st7	st8	✗
7	st1	st2	st3	st4	st5	st6	st7	st8	✗
8	st1	st2	st3	st4	st5	st6	st7	st8	✓
9	st1	st2	st3	st4	st5	st6	st7	st8	✗
10	st1	st2	st3	st4	st5	st6	st7	st8	✓
11	st1	st2	st3	st4	st5	st6	st7	st8	✗
12	st1	st2	st3	st4	st5	st6	st7	st8	✗
13	st1	st2	st3	st4	st5	st6	st7	st8	✗
14	st1	st2	st3	st4	st5	st6	st7	st8	✗
15	st1	st2	st3	st4	st5	st6	st7	st8	✓
16	st1	st2	st3	st4	st5	st6	st7	st8	✗
	st1	st2	st3	st4	st5	st6	st7	st8	✓

(a) Delta Debugging Iterations (iterations 3 and 13 fail due to syntax errors)

Iterations	Program State								Oracle
0	st1	st2	st3	st4	st5	st6	st7	st8	✓
1	st1	st2	st3	st4	st5	st6	st7	st8	✗
2	st1	st2	st3	st4	st5	st6	st7	st8	✗
3	st1	st2	st3	st4	st5	st6	st7	st8	✓
4	st1	st2	st3	st4	st5	st6	st7	st8	✗
5	st1	st2	st3	st4	st5	st6	st7	st8	✓
6	st1	st2	st3	st4	st5	st6	st7	st8	✗
7	st1	st2	st3	st4	st5	st6	st7	st8	✗
8	st1	st2	st3	st4	st5	st6	st7	st8	✓
9	st1	st2	st3	st4	st5	st6	st7	st8	✗
10	st1	st2	st3	st4	st5	st6	st7	st8	✗
11	st1	st2	st4	st4	st5	st6	st7	st8	✗
12	st1	st2	st3	st4	st5	st6	st7	st8	✗
	st1	st2	st3	st4	st5	st6	st7	st8	✓

(b) Chisel Iterations

Iterations	Program State								Oracle
0	st1	st2	st3	st4	st5	st6	st7	st8	✓
1	st1	st2	st3	st4	st5	st6	st7	st8	✓
2	st1	st2	st3	st4	st5	st6	st7	st8	✗
3	st1	st2	st3	st4	st5	st6	st7	st8	✓
4	st1	st2	st3	st4	st5	st6	st7	st8	✗
5	st1	st2	st3	st4	st5	st6	st7	st8	✓
	st1	st2	st3	st4	st5	st6	st7	st8	✓

(c) Blade Multi Processing Iterations

Fig. 1: Comparison of Delta Debugging [10], Chisel, and BLADE on a sample hashing program bloated with unnecessary functionalities. Of the 4 available algorithms in the generic code, a specific environment only needs SHA256. In each iteration, statements, marked green are retained while those in red form the candidate set for removal. The subfigure (a) represents the delta debugging reduction run against Chisel's run on (b) and BLADE 's run on (c). Note that, for the purposes of software debloating, the original Delta Debugging algorithm may not be 1-minimal as it does not consider program dependencies.

Our evaluation on a diverse set of well-known benchmarks indicates that existing source code debloaters Chisel [1] and Debop [7] take a long time (several hours) to debloat commonly-used *tiny* coreutils software while timing out on large workloads. The evaluation (Section IV) reveals that while BLADE 's performance is comparable in code, attack surface reduction with state-of-the-art, it runs faster than both (2.3× faster than Chisel and 2.75× faster than Debop) enabling it to handle large workloads better.

In summary, our evaluation shows that high code reduction, security hardening, and generality potential with low runtime overhead make BLADE favorable for modern dynamic debloating use cases.

Overall, this paper makes the following contributions:

- **Novel Approach:** BLADE leverages its fast algorithm to effectively scale to large workloads (i.e., nginx, sqlite3). This makes BLADE a practical solution for realistic workloads.
- **Comprehensive Evaluation:** A comparative evaluation of BLADE with existing debloating frameworks shows a significant improvement over other schemes, in terms of *time to debloat* without compromising code reduction.
- **Open Source:** Our framework will be open source and available in a GitHub repository[3].

II. MOTIVATION AND INSIGHTS

Given the source code and a test oracle (based on test cases), Delta Debugging-based code debloating approaches iteratively find a smaller version of a program with minimal code to satisfy the oracle (Figure 1(a)). These approaches are usually exhaustive. A typical strategy is to remove large chunks of code first, before considering individual statements. In Figure 1a(a), we demonstrate the scenario with several iterations of Delta Debugging on a toy example.

This approach is inefficient for two reasons:

1) Statements that are positive candidates for debloating may be dispersed instead of appearing in chunks. Thus, the time it takes to search for the largest chunk of removable code can potentially outweigh the possible time benefit of actually removing a large chunk of code.
2) It does not take into consideration the syntax rules in programming languages that follow the definition before usage principle (such as in C/C++).
   Without this consideration, for example, there is potential to include a *definition* in the candidate set but leave the corresponding *use* intact within the remaining code

[3]https://github.com/pawnsac/BLADE-deb

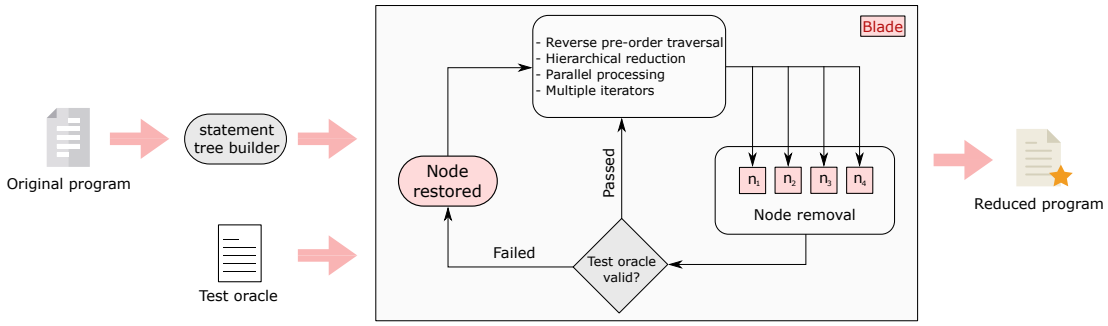

Fig. 2: Overview of BLADE System Design.

for testing against the specification. Clearly, such a code would generate syntax errors and it is needless to even attempt to compile it (for example, iteration 3 in Figure 1a). Time wasted in attempting to compile such a code can be saved by removing the variable *usage* before their *definition*.

Chisel [1] aims to improve efficiency by employing reinforcement learning to smartly navigate the search space. However, it is not free from these deficiencies either, since, it does not inherently leverage the syntactic rigidity of the code constructs. Though it takes fewer iterations than Delta Debugging to converge, it still is inefficient as shown in Figure 1b. This motivated us to design a structure-guided search space reduction technique (named BLADE) for faster code debloating without affecting the reduction and correctness. Next, we discuss the insights we capitalized on while designing BLADE .

Insight I: Leveraging define before use structure. Usually, coding constructs are *defined* before *used*. Specifically, the probability of causing errors after the removal of a statement reduces with the depth of its position in the syntax tree. Our source code debloating tool BLADE leverages this insight to propose a new approach to optimize the number of iterations. As seen in Figure 1b, where the first statement to be removed is the last one in the sequence (iteration 1). The usages of the functions md5sum and sha1sum were removed in iterations 3 and 4 before the definitions were removed in iterations 7 and 8. This is unlike in Delta Debugging, where it attempted to remove the definitions first (iteration 3 in Figure 1a), causing the test specification to fail very early on even though these statements would be removed in further iterations anyway (when it would no longer cause a syntax error).

Insight II: Leveraging syntax tree hierarchy. Given a code block (if-else, function, struct) that contains multiple statements within itself, it is possible that the full block could be removed before considering the individual statements within it. This is different from the removal of large chunks of code in Delta Debugging in that Delta Debugging employs binary search to remove large chunks within a block. BLADE leverages this insight to first remove the full block before considering its child statements. This further optimizes BLADE 's speed.

III. SYSTEM DESIGN

BLADE takes in a target program—the bloated software—along with a test specification developed for a particular deployment setup. It outputs a debloated version of the target program with respect to that test specification. Formally stated, BLADE takes a target program P as input that exhibits a set of functionalities $\mathbb{F} = \{f_1, f_2, f_3, ..., f_n\}$ along with a test specification (also known as *the test oracle*) that represents a function $\mathrm{T}_{\mathrm{test}}$ such that:

$\mathrm{T}_{\mathrm{test}} : \mathbb{P} \rightarrow \mathbb{O}$, where $\mathbb{O} = \{T, F\}$ and $\mathbb{P}$ is the space of all possible programs, where $\mathrm{T}_{\mathrm{test}}(P) = T$. Test oracle $\mathrm{T}_{\mathrm{test}}$ tests the presence of a subset $\mathbb{F}'$ of $\mathbb{F}$ in a given candidate program P' during the debloating process and also checks the fitness (e.g., if new memory leaks, gadgets, etc. arise) of the program P'. If P' retains $\mathbb{F}'$ and is fit, then $\mathrm{T}_{\mathrm{test}}(P') = T$, or else $\mathrm{T}_{\mathrm{test}}(P') = F$. Upon completion of the debloating process, BLADE outputs a reduced (debloated) program P_f that satisfies $\mathrm{T}_{\mathrm{test}}(P_f) = T$.

```
int main(int argc, char *argv[]) {        //node 1
  int x = atoi(argv[1]);                  //node 2
  if (x >= 0) {                           //node 3
    int y = sqrt(x);                      //node 4
    return y;                             //node 5
  } else {                                //node 6
    printf("Number must be >= 0");        //node 7
    return -1;                            //node 8
  }
}
```

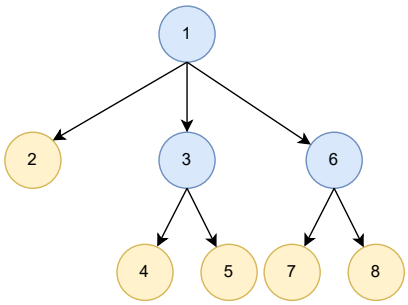

Fig. 3: A statement tree of a C program that calculates the square root of a given number. The blocks are placed at internal nodes (colored blue) while the statements are placed at leaf nodes (colored yellow).

Specifically, the debloating process flows as follows: 1) A candidate set of statements is identified and removed from the program, 2) The remaining program is compiled 3) The test oracle executes the required functionalities of the program 4) The test oracle compares the program output to the expected outputs 5) Depending on the test oracle outcome, the candidate statements are permanently removed (debloated) or are reintroduced into the program. This process is repeated until the debloating algorithm cannot debloat the program further.

Figure 2 shows the overall system design. Next, we discuss the individual components of the system.

A. Statement Tree

Given an input program, we start by building a *statement tree*. Unlike the abstract syntax tree [11], a statement tree works at the granularity of statements—the leaf nodes are the complete statements of a programming language. The internal nodes are blocks of code such as functions, loops, if-else conditions, or structs. BLADE keeps the statement blocks as internal nodes to implement hierarchical reduction: removal of an internal node (and its entire subtree) is equivalent to the removal of that entire code block. If the removal of a block, as part of the candidate set, fails the test specification, the block is put back into the code and the reverse pre-order traversal approach is taken inside that block (within the subtree). An illustration of an example program and its derived statement tree is given in Figure 3.

Constructing the statement tree is facilitated by using LLVM's frontend Clang [12] for extracting the tokens of the original code. We then build the statement tree using these tokens and categorizing them into statements or blocks.

B. Reduction Algorithm

The reduction algorithm takes as input the statement tree and performs reduction recursively. The reduction algorithm traverses the statement-tree in the reverse pre-order fashion. The base algorithm (without multiprocessing optimizations) is shown in Algorithm 1. BLADE recursively reduces the size of the context tree by first attempting to remove the entire tree (Line 2), and if the test oracle fails, it restores the tree (Line 5). It then considers the nodes within the restored sub-tree. Since our algorithm traverses each node only once, it runs in linear time with respect to the nodes in the statement tree (the statements in the target program).

*1) **Traversal**:* The reduction algorithm takes the statement tree as a parameter and performs a reverse pre-order traversal for reduction. It begins by attempting to remove the root node, which removes all of its descendant nodes. If the entire block cannot be removed (some part of it includes necessary functionality), it will be restored and statements inside this block of code are considered in accordance with hierarchical reduction. For example, if there is a while loop, the algorithm will first attempt to remove the entire loop. If the test oracle fails at this point, the loop is restored and the iterator continues reduction within this loop (starting from the last node in this block). Using this approach, entire blocks of code can be removed at once if the functionality within them is not required.

C. Parallel-processing

*1) **Simultaneous multi-node reduction**:* We further optimize BLADE 's hierarchical traversal approach by considering multiple children nodes (under a given node) simultaneously in one iteration, where the output is the code with the maximal set of removable candidates. Specifically, we run Algorithm 1 in multiple processes to remove a different combination of children nodes in parallel. The process which passes the test oracle with the largest candidate set is our *best reducer* and it updates the global state by permanently removing those nodes from the currently debloated program. This optimization is shown as Algorithm 2. Lines 5 and 6 assign the nodes (the candidate set) to the processes. In Line 7, the processes remove their assigned nodes in parallel (in separate copies of the current global state of the reduced program) and indicate whether or not the test script passed. The best reducer is determined in Line 15. Note that, given n is the number of children of a node, there can be a 2^n combination to try, which is prohibitively high. Thus, we use utilize the "define before use" insight to create an ordering of the combinations, so that based on the computation power the reduction gain can be maximized. However, in our prototype implementation, we only consider the first n combinations. At a given node, BLADE picks the first upcoming node as the candidate set for the first process; the second process gets the first two upcoming nodes, and so on, with the n-th process getting the first n upcoming nodes as the candidate set for running Algorithm 1.

Algorithm 1 Basic Reduction Algorithm

```
1: function REDUCTION(tree: block)
2:     remove(tree)
3:     returnCode ← exec(oracle)
4:     if returnCode ≠ 0 then
5:         restore(tree)
6:     else
7:         return
8:     end if
9:     for node in reversed(tree) do
10:        if type(node) = block then
11:            REDUCTION(node)
12:        else
13:            remove(node)
14:            returnCode ← exec(oracle)
15:            if returnCode ≠ 0 then
16:                restore(node)
17:            end if
18:        end if
19:    end for
20: end function
```

Our multi-node processing approach has two main benefits:

1) Speed: With multi-node processing, larger chunks of code may get removed in one iteration, which enables faster traversal. For instance, with 5 processes running in parallel, up to 5 sibling nodes can be removed in one go. This is illustrated in Figure 4 when debloating the chown-8.2 utility program. It can be seen that in about 10% of

Algorithm 2 Multiprocess Reduction Algorithm

```
1: function REDUCTION(tree: block)
2:     i ← 0
3:     nodes ← reversed(tree)
4:     while true do
5:         queue ← nodes[i : i + n]
6:         batch ← accumulateNodes(queue)
7:         result ← removeInParallel(batch)
8:         result ← reversed(result)
9:         if result = [False] ∗ n then
10:            if type(nodes[i]) = block then
11:                REDUCTION(nodes[i])
12:            end if
13:            i ← i + 1
14:        else
15:            i ← len(result) − (result).index(True) + i
16:        end if
17:        if i = len(nodes) then
18:            break
19:        end if
20:    end while
21: end function
```

cases, the process working with the first four upcoming nodes in the traversal is chosen as the *best reducer* leading to the removal of 4 code statements in one go. The figure also demonstrates that in 67.6% of the cases more than a single statement is removed in one step.

2) Reduction: it allows the algorithm to consider multiple combinations of nodes for reduction. This is especially helpful in removing code with neutralizing effects. As an example, consider the following target code:

```
int x = 5;
x++;
x--;
printf("%d ",x);
```

Consider the scenario where the test oracle is expecting the number 5 to be printed. Statements 2 and 3 are neutralizing when considered together, but if one statement is removed without the other, the test specification will fail. Having multiple processes helps avail such debloating opportunities for a higher reduction in code size.

The number of processes allocated to the reduction process is dependent on the resources the user is willing to provide. In general, increasing the number of processes allocated to reduction increases both the speed and reduction. Figure 4 illustrates the benefit of using multiprocessing.

*2) **Secondary reduction**:* Since trying 2^n combinations is prohibitively high and BLADE only uses a prioritized set of combinations – there is always a chance of producing more reduction with a secondary set of iterations. To leverage this insight, we run another set of processes to perform a secondary simultaneous traversal, while the previous traversals are in progress, on the code that is already been processed once. These secondary iterations have the following purposes: 1) reduction of new combinations of nodes made after the first pass of reduction, and 2) dead code elimination (including the removal of label definitions, which do not follow the definition before usage principle).

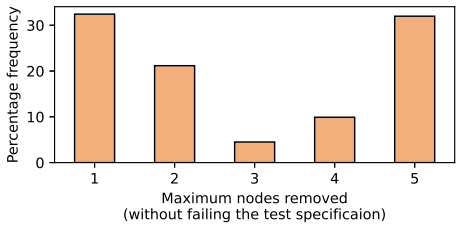

Fig. 4: Demonstrating the percent frequency of the number of candidate statements for the *best reducer* while debloating chown-8.2 with 5 processes. In about 30% of cases, 5 nodes were removed at once.

These secondary iterations perform Algorithm 2 with varying degrees of progress in the statement tree. However, it is important to keep a distance of *at least one function* between the parallel iterators for two reasons: 1) so that the reduction pathways of multiple iterators do not overlap (which would cause redundancy) and 2) it allows an iterator to comprehensively explore a function before creating new opportunities of reduction for the next iterator. BLADE provides the opportunity to adjust the number of parallel iterators and their respective processes. Having multiple iterators reduces the need to perform multiple full traversals of the program and significantly improves the code reduction levels.

D. The Test Oracle

BLADE uses two types of test oracles. To ensure functional correctness it uses functionality checkers and to preserve the quality of the reduced code it uses fitness checkers. These checkers combinely output either True or False, indicating whether or not the compiled program has correctly retained a given set of functionalities and fitness. Next, we discuss our test oracles in detail.

Functionality Checkers. To guide the debloating of a program to retain a specific functionality, we require a *functionality checker* to confirm the said functionality. We use a set of test scripts to play the role of the functionality checker oracle. To ensure the generality of the binary, test scripts should extensively cover the core functionalities. The generality of the program w.r.t. a given functionality is defined as the fitness of a program to handle a general range of inputs for the specified functionality. However, a side effect of increasing the number of training cases is that it increases the debloating time for each iteration. Hence, reducing the number of iterations effectively allows more training cases, which directly impacts the generality of the reduced program. Thus, by optimizing the number of iterations, BLADE enables better generality for a wider range of inputs.

Fitness Checkers. BLADE uses fitness checkers to maintain the quality of the debloated code. For example, source code debloaters can occasionally introduce vulnerabilities, increase gadget counts, etc. Fitness checker oracles can be realized to prevent that. In the current implementation, we used the Clang static analyzer to ensure the debloating fitness. These sanitizers

included Control Flow Integrity Sanitizer, Address Sanitizer, Memory Sanitizer, Undefined Behavior Sanitizer, and Leak Sanitizer as implemented in [1].

The test oracle is implemented as a bash script. An example is provided in Figure 12 in the Appendix. The code represents the test oracle of *gzip* (one of the target programs). The *args_test* function tests the compression and decompression functionalities of gzip using *run-d* and *run-c* as helper functions. Furthermore, the compile function leverages Clang's sanitizers during the compilation to statically check for any new security vulnerabilities introduced during the debloating process.

IV. Evaluation

We compare BLADE with three recent software debloating frameworks: Chisel, Debop [7], and Razor. Chisel and Debop operate on the source code and Razor [2] directly debloats the binary program.

A. Criteria

We have considered the following aspects to report on the performance of BLADE :

- **Analysis Efficiency:** What is the runtime performance gain BLADE provides in comparison to the state-of-the-art debloating tools?
- **Reduction:** How much code reduction is achieved by BLADE in comparison the state-of-the-art debloating tools?
- **Security:** How much security benefits does BLADE yield compared to the state-of-the-art debloating tools?
- **Generality:** How does BLADE perform in maintaining the generality of the debloated program?

B. *Sequence of Evaluation*

First, we evaluate BLADE on small sized programs to perform a comprehensive comparison with other debloaters (Chisel, Debop, and Razor). This is done in Sections IV-E, IV-F, IV-G, and IV-H.

Next, we evaluate BLADE on big applications on all of the performance metrics mentioned in Section IV-A. This evaluation, which is given in Section IV-I, highlights the main strength of BLADE : scalability on real-life workloads.

C. Evaluation environment

We performed program debloating on a Linux Server that housed two Intel Xeon Silver 4210 CPU@2.20GHz processors along with 512 GB DDR4 memory. The operating system was Ubuntu 20.04.

D. Experimental Setup

*1) **Target Programs and Functionality Selection**:* We used ***nginx*, *sqlite3*, *make*, and *tar*** as big program samples to measure the scalability. For the comparison with other debloaters, we used 9 coreutil programs from ChiselBench [1]. For the selection of arguments to consider in the test specification (and hence keep in the reduced program), we aimed to pick those that represented the core functionality of each of the programs (given in the Appendix in Table VI). We left out the arguments that carried the auxiliary functionalities. For each of these arguments, we ensured that the functionality checker of our test oracle comprehensively covered a diverse range of realistic inputs to the program. For example, for sort, uniq, and grep, our input files included text, binary, image and archive files. This diversity is shown in Figure 13 in the Appendix. Within these input formats, we varied the file sizes (by including larger files) to ensure that the required functionality was extensively covered within the training cases. Some training cases were taken from Razor-benchmarks [2].

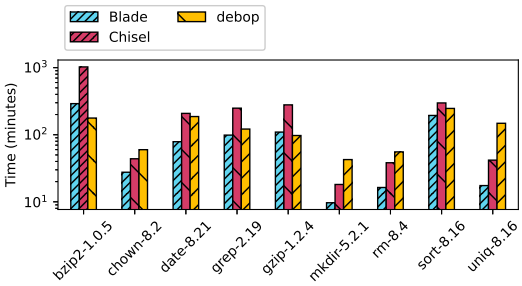

Fig. 5: Runtime comparison of Chisel, Debop, and BLADE (log-scale)

*2) **Setup of** BLADE **, Chisel, Debop, and Razor**:* We ran BLADE , Chisel, and Razor using the same test oracle (in terms of functionality testing), modified to fit the requirements of different tools. For example, Razor requires that its oracle is written in python. Debop requires its oracle to output a set of measurements at the end of each run.

Debop also requires a set of parameters for debloating. As used in the evaluation in their paper [7], we set 1000 sampling iterations, with α (weight for reduction) as 0.5, β (weight for generality) as 0.75, and the density value as 50. We gave a higher value for generality as it is a highly important metric for a debloater.

E. Analysis Efficiency

Using the same functionality checker, we compared the time efficiency (debloating time) of BLADE with Chisel and Debop, the state-of-the-art source code debloating approaches. We performed the experimentation on ChiselBench, the results of which are shown in Figure 5. It can be seen that our approach performs the best overall by taking the least time. For small coreutils in ChiselBench, BLADE is on average 2.3× faster than Chisel and 2.75× faster than Debop. This speed-up is mainly attributed to the linear complexity of our reduction algorithm, coupled with parallel processing, which further optimizes the time complexity. For larger sized-programs, BLADE offers a better speedup.

Moreover, BLADE scales well as the test specifications turn more comprehensive (to ensure Generality with a broader range of inputs). Increasing test case intricacy increases the time it takes to debloat a program as more scenarios must be tested against. By speeding up the reduction of analysis time,

Generality can be ensured without fear of an exorbitant time for debloating.

Summary: BLADE is on average 2.3× faster than Chisel and 2.75× faster than Debop. This allows it to scale to large codebases and incentivizes adding more intricate test cases to the test oracle.

F. Reduction

While BLADE significantly outperforms Debop and Chisel in terms of time to debloat, it is crucial that this speedup does not come at the cost of program reduction. We evaluate reduction with respect to four metrics: binary size, executable code size, basic blocks, and lines of code. We now present the binary size reduction results. The rest are given in Appendix A.

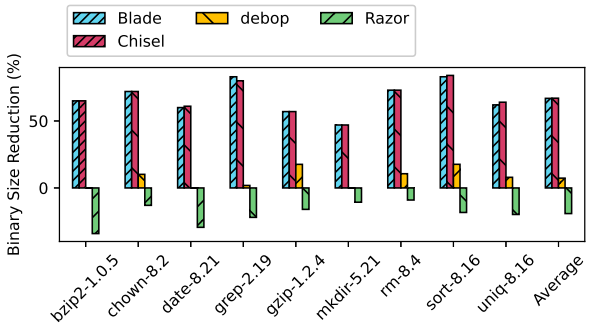

Fig. 6: Comparison of binary size reduction of BLADE , Chisel, Debop, and Razor

*1) **Binary Size**:* We compiled the resultant source code from Chisel, Debop, and BLADE and compared the size of resultant binary, results of which are shown in Figure 6. BLADE produces an average reduction of 67% compared with Chisel's 67%, Debop's 7%, and Razor's -19%. Blade and Chisel perform the highest percentage of reduction, whereas Razor actually increases the size of the binary on average by 13%. This is because the current implementation of Razor makes a new .text section in the binary, while making the previous .text section read-only. Debop yields low reduction levels of only 7%. All in all, BLADE 's high reduction is a testament to the fact that BLADE 's analysis efficiency does not come at the cost of reduction.

The results of the executable code size, basic blocks, and the lines of code reduction are presented in Appendix A.

Summary: Chisel and BLADE are comparable in binary size, executable code section, and basic block reduction, with both providing significantly more reduction than Razor and Debop. BLADE , on average, produces an output with the fewest lines of code.

G. Security

One of the most important goals of program debloating is to reduce the attack surface or the number of vulnerabilities in the program, and ensure that any new vulnerabilities are not introduced. We measure the security impact from three perspectives, which we present next.

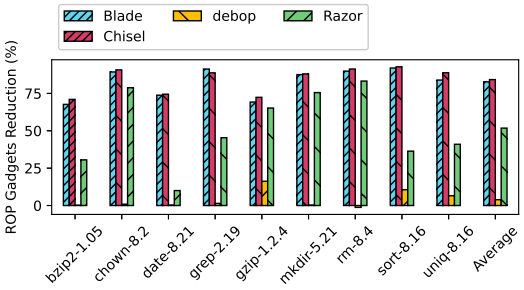

Fig. 7: Comparison of ROP Gadgets Reduction of BLADE , Chisel, Debop, and Razor

*1) **ROP Gadgets**:* A security measure commonly used for evaluating programs is the count of ROP gadgets. These are instructions in the code that can potentially be used for a code reuse attack. We calculate the total unique ROP gadgets using Salwan's ROP Gadget tool [13]. Overall statistics for the percentage of ROP gadgets reduced using the three tools can be seen in Figure 7. BLADE and Chisel both result in significant reduction in ROP gadgets: 83% and 84%, respectively. Razor provides a lower reduction (52%, on average) in ROP gadgets, which can be attributed to the fact that Razor performs significantly less code reduction. Debop provides the lowest reduction (4%, on average) in ROP gadgets.

Program	Original	Blade	Chisel	Debop
bzip2-1.05	0	2NF 2PL	2NF	0 NF 0 PL
chown-8.2	3NF 12PL	4NF	1NF	3NF 6 PL
date-8.21	1NF 4PL	0	0	1 NF 4 PL
grep-2.19	4NF 8PL	3NL 1PL	2NF 1PL	5 NF 8 PL
gzip-2.1.4	1NF 1PL	1PL	0	1 NF 1 PL
mkdir-5.2.1	1NF 1PL	1NF	1NF	1 NF 2 PL
rm-8.4	1NF 9PL	1PL	1PL	2 NF 7 PL
sort-8.16	3NF 4PL	4NF	2NF	9 NF 2 PL
uniq-8.16	1NF 1PL	0	0	1 NF 0 PL

TABLE I: Comparison of Blade, Chisel, and Debop on security alarms generated by Saber [14]. NF indicates never freed allocations and PL indicates partial leaks.

*2) **Memory leaks**:* Memory leaks pose a security threat as they can be leveraged by an attacker to cause a denial-of-service attack if the leaks can be maliciously induced by attackers. Thus, reducing memory leaks or preventing the introduction of new memory leaks is important for debloaters. We evaluate the memory leaks of a given program with a static analysis tool called Saber [14]. We use Saber to report the number of partial leaks and memory allocations that are never freed before and after debloating with BLADE , Chisel, and Debop (shown in Table I). Our evaluation shows that the performance of BLADE and Chisel is comparable in terms of reducing memory leaks since both of them use fitness checkers to specifically handle memory leaks. Note that, in some cases, BLADE introduced new leaks. This was because BLADE 's fitness checkers use Clang's static analyzer [12], which suffers from *missed detections*. This can be avoided by using a better fitness checker.

TABLE II: ✓ indicates that the debloated program produced by the given tool no longer has the mentioned CVE. ✗ indicates that the CVE is still present in the reduced program.

Program	CVE (CVSS Score)	Blade	Chisel	Razor
bzip2-1.0.5	CVE-2011-4089 (4.6) [15]	✗	✗	✗
chown-8.2	CVE-2017-18018 (1.9) [16]	✓	✓	✓
date-8.21	CVE-2014-9471 (7.5) [17]	✓	✓	✗
grep-2.19	CVE-2015-1345 (2.1) [18]	✓	✓	✓
gzip-1.2.4	CVE-2005-1228 (5.0) [19]	✓	✓	✓
mkdir-5.2.1	CVE-2005-1039 (3.7) [20]	✗	✗	✗
rm-8.4	CVE-2015-1865 (3.3) [21]	✗	✗	✗
sort-8.16	CVE-2013-0221 (4.3) [22]	✓	✓	✓
uniq-8.16	CVE-2013-0222 (2.1) [23]	✗	✗	✗

*3) **CVEs**:* Common Vulnerabilities and Exposures (CVEs) are known vulnerabilities of specific programs that are identified and then shared with the public. For each target program, selected CVEs that were identified in the original version were rechecked to see whether they still existed in the reduced binaries after debloating. Table II summarizes these results for BLADE , Chisel, and Razor. We excluded Debop for this comparison since Debop's performance was consistently worse in all the previous measures. We note that, with the new test scripts, BLADE successfully removed 5 out of 9 CVEs–the same count produced by Chisel. In comparison, Razor removed 4 CVEs from the 9 programs. Although, the debloating process does not specifically target CVEs, however, one observation to be taken from the results in Table II is that CVE removal is highly correlated with the functionalities that are being removed.

Summary: BLADE and Chisel remove the most ROP gadgets from the original code (with Chisel removing slightly more ROP gadgets). BLADE successfully removes a significant number of memory leaks in a majority of cases, however owing to an ineffective fitness checking, the number of memory leaks increase in 2 out of the 9 programs. BLADE and Chisel removed the same number of CVEs (5 out of 9), while Razor removed 4 out of 9 CVEs.

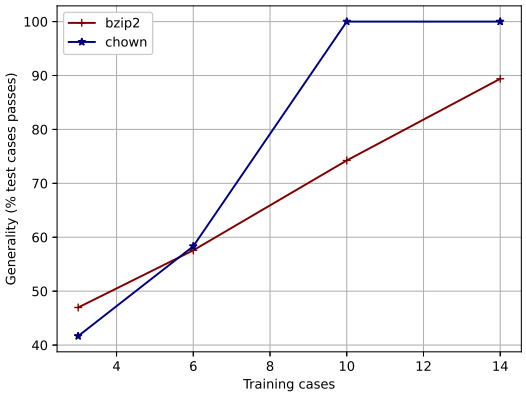

Fig. 8: The relationship between training cases and generality (quantified through the % of test cases passed) for two selected applications bzip2, and chown.

Program	Total	Blade	Chisel	Razor	Debop
bzip2-1.05	50	46	45	**50**	**50**
chown-8.2	10	**10**	**10**	**10**	**10**
date-8.21	30	**30**	**30**	**30**	**30**
grep-2.19	45	42	42	**45**	**45**
gzip-1.2.4	50	**50**	46	49	**50**
mkdir-5.21	20	**20**	**20**	**20**	**20**
rm-8.4	15	**15**	**15**	**15**	**15**
sort-8.16	28	24	24	**28**	**28**
uniq-8.16	28	25	25	**28**	**28**

TABLE III: The number of validation test cases passed by debloated programs produced by BLADE , Chisel, Debop, and Razor. Bold results indicate all test cases passed.

H. Generality

One of the major goals BLADE is to ensure that the debloated programs retain generality. The generality of the program w.r.t. a given functionality is defined as the fitness of a program to handle a general range of inputs for the specified functionality/arguments to be retained after it has been debloated. For example, the generality of bzip2 for the *-fc argument* (compression functionality) would be how many inputs can the bzip2 successfully compress using the *-fc argument* after debloating. To test generality we designed a new set of test cases that are not part of the functionality checker oracle for each of the target programs (Table III). These test cases, though different than the training cases, checked similar functionality that was tested in the training cases. 5 out of 9 programs pass all of the test cases for BLADE (4 programs fail some test cases). In comparison, Chisel fails on 5 out of 9 programs. All the debloated programs from Debop passed all the test cases. Razor performed almost perfectly in the generality.

*1) **Generality and the number of training cases**:* Figure 8 shows the correlation between the number of training cases in the functionality checker test oracle and the percentage of test case passing (generality retained) from the validation set. It roughly demonstrates that by increasing the number of training cases in the test oracle, generality increases for the target program. We chose two different types of programs for this experiment: **bzip2** belongs to the class of compression utilities, and **chown** belongs to the class of system utilities as it changes permissions for directories. bzip2 is a utility that supports a wide range of inputs, and for such utilities it may take a high number of corresponding inputs as training cases to attain high generality, as demonstrated in Figure 8. For utilities like chown that do not support a diverse range of inputs, even a small number of training cases in the test oracle can guarantee optimal generality, as demonstrated in Figure 8. This result further supports the thesis that BLADE can get optimal generality performance when provided an optimal set of inputs in the test oracle.

Summary: Our evaluation demonstrates that BLADE , while not specifically geared towards optimizing for generality, produces results that are at least as good or comparable to the state-of-the-art (Chisel, Razor, Debop). This generality can be

CVE-ID	Description	CVSS Score
CVE-2013-0337	allows local users to obtain sensitive information from log files	7.5
CVE-2016-0746	allows remote attackers to cause a denial of service via a crafted DNS response related to CNAME response processing	7.5
CVE-2016-0747	allows remote attackers to cause a denial of service via vectors related to arbitrary name resolution	5.0
CVE-2016-0742	allows remote attackers to cause a denial of service via a crafted UDP DNS response	5.0
CVE-2017-7529	integer overflow vulnerability in nginx range filter module resulting into leak of potentially sensitive information	5.0
CVE-2012-1180	allows remote HTTP servers to obtain sensitive information via a crafted backend response	5.0

TABLE IV: This table lists down different CVEs removed from nginx after debloating with BLADE

further enhanced by increasing the number of training cases in the test oracle.

I. Scalability: Debloating Large Applications

To study whether source code debloaters can handle real-world applications, we selected two popular large-sized applications (*nginx* and *sqlite3*), and two medium-sized applications (*tar* and *make*). *nginx* has 76K lines of code (LoC) while *sqlite3-manager* has 75K LoC. They are both $7.5\times$ larger than the average program in ChiselBench. *tar* is 31K LoC and *make* is 27K LoC. These medium-sized programs are about $2.7 - 3\times$ larger than the 9 coreutils we picked earlier from ChiselBench.

1) ***Existing debloaters on large workloads****:* We ran BLADE , Chisel, and Razor on the 4 applications. Since Debop achieves much less code and attack surface reduction on small programs than existing tools (as seen in Sections IV-F and IV-G), we did not run it on large ones.

When using our training cases for debloating, Chisel either did not manage to complete its debloating in the timeframe of 24 hours or an error occurred while debloating the 4 target programs. For example, we observe that debloating *tar* with the training cases that Chisel originally used, produce 0% generality, thus we augment the set of training cases to produce meaningful results. Specifically, we increased the number of test files to 27 (9 per each argument). However, after this change, Chisel was unable to finish debloating *tar* in the 24 hour time frame. We were not able to successfully run Chisel on *make* because of a version mismatch issue. The debloating process got stuck while debloating *nginx*, where *sqlite3* timed out. Razor was not able to debloat any application except tar due to an internal error during the debloating process. Since Razor is not a source-code debloater, we did not invest significant effort to debug the error.

2) ***nginx****:* This is a popular web server. We chose it for two reasons: (a) its large size. (b) documentation of its vulnerabilities in CVEs (Table IV). The results after debloating are shown in Table V. We see a large reduction in size (87% binary reduction and 94.9% code size reduction) and attack surface (93.7% ROP gadgets reduction), while still maintaining a high level of generality.

BLADE removes a total of 6 CVEs from *nginx* as shown in Table IV. This is mainly due vulnerable functionality that is removed in the reduction process. This demonstrates the utility of debloating as a means to remove code that could potentially be used to launch an attack. For generality testing of the debloated binary, we hosted 20 different HTML pages using *nginx*, out of which 18 successfully worked.

3) ***sqlite3-manager****:* This is a popular database manager. It was selected due to its large size and to demonstrate how security could be enhanced by removing dangerous functionality. For example, it is safer to remove "update" or "delete" support (to prevent SQL injection attacks) when unused by programs that utilize *sqlite3*. We debloated *sqlite3-manager* to illustrate this. The results after debloating with BLADE are shown in Table V. We see a large reduction in size (72.6% binary reduction and 78.5% code size reduction) and attack surface (75.9% ROP gadgets reduction), while still maintaining a high level of generality.

4) ***tar****:* This is a utility that provides the ability to create and extract file archives. It contains many archiving and compression options. This leads to bloat in the resultant codebase. The results after debloating with BLADE are shown in Table V. We get a large reduction in size (83.0% binary reduction and 96.4% code size reduction) and attack surface (94.3% ROP gadgets reduction), while still maintaining perfect generality.

5) ***make****:* This is a popular build-system used to automate the building of applications. It contains various functionality that is rarely used, making it a good candidate to perform debloating. The results after debloating with BLADE are shown in Table V. We get a large reduction in size (70.0% binary reduction and 74.6% code size reduction) and attack surface (74.0% ROP gadgets reduction), while still maintaining perfect generality.

V. DISCUSSION

Generality and training cases: Like other source code debloating solutions, BLADE 's generality relies on the comprehensiveness of the functionality checker oracle. Our evaluation (Figure 8) shows that the generality of BLADE increases with the number of training cases in the oracle. Addressing generality with a minimal number of training cases is an open problem [7]. However, for certain types of the target programs that have discrete effects on the system (such as rm, mkdir, and chown), designing a comprehensive functionality checker with a smaller set of high-quality training cases is feasible.

Programming languages: The current implementation of BLADE is able to reduce C/C++ programs, but the algorithm and concept can, theoretically, be applied to any programming language that follows the *definition-before-usage* structure. We acknowledge that this still makes BLADE a language-specified approach (unlike the traditional delta debugging algorithm which does not consider syntactical dependencies and can hence be effective for an even wider variety of programming languages), however, utilizing language information makes our approach more efficient approach for the languages that can support it. To cater to a wider variety of languages, future implementations of BLADE can take as an input LLVM IR, a low-level representation of a program used by the LLVM framework [24]. Considering LLVM IR still maintains an SSA

Program	Target Functionality	Time to debloat (h)	Binary size reduction (%)	Code size reduction (%)	Basic blocks reduction (%)	Attack surface reduction (ROPgadgets) (%)	Generality (%)
nginx	hosting a HTML server	10.3	87.1	94.9	94.5	93.7	90
sqlite3-manager	make, list, query, and insert into tables	5.6	72.6	78.5	78.2	75.9	88.8
tar	archive files and extract them using cz, czf, and xf flags	10.8	83.0	96.4	98.2	94.3	100.0
make	make with given arguments	7.2	70.0	74.6	76.8	74.0	100.0

TABLE V: This table shows the results of debloating large applications with BLADE .

form (which defines variables before use), a program written in any language can first be compiled to LLVM IR, and then can be reduced using BLADE .

VI. RELATED WORK

A number of techniques for software debloating exist in the literature. In this section, we categorize these approaches and explain how BLADE differs from them.

Source code debloaters: Many debloaters work directly with the source code. These include C-reduce [8], Perses [9], Chisel [1], Debop [7], DomGad [25], and Cov-A/F [26]. Most of the *source code debloating* techniques utilize variations of the Delta Debugging (DD) algorithm [10] which was originally developed to establish the minimum failure-inducing code in a program that worked 'yesterday' (a working template program) but not 'today' (erroneous program). Thus, DD discovered the *buggy* code, removal of which will allow the program to work; software debloating may use the same principle to discover (and remove) the *unnecessary* code, removal of which will let the program work normally.

Using DD under the hood, C-reduce [8] applies a series of transformations (changing identifiers, the scope of variables, combining definitions, and substituting function calls with function bodies) to reduce the source code. This not only takes a long time which may be infeasible, but it also results in code that is not auditable due to the above-mentioned transformations. In contrast, BLADE can debloat fast in $\mathcal{O}(n)$ time while maintaining the auditability and readability of the source code.

Another approach is Perses [9], which is a syntax-guided program reducer that also employs DD and aims to deal with the inefficiency of previous DD approaches by leveraging the syntax of the language to avoid syntactically invalid outputs. While it reduced the amount of time the deloating process takes (as compared to C-reduce), it still times out in several cases, deeming it impractical for general usage, as demonstrated in [1]. BLADE 's efficient algorithm avoids timeouts and scales linearly with the size of the program.

Chisel [1] (which is considered the state-of-the-art source code debloating tool) was developed specifically to reduce the attack surface of a program. It implements an optimized DD algorithm that utilizes reinforcement learning. Chisel shows a marked improvement in terms of time to debloat when compared with C-Reduce and Perses [1]. BLADE performs on average 2.3× faster than Chisel (Section IV-E), making it usable for practical workloads.

While the aforementioned approaches differ in the specific optimizations they apply, they all use DD under the hood. BLADE differs from them in that it does not use DD, rather it presents a novel algorithm for selecting a candidate set of statements (Section III-B) to tackle source code debloating. DomGad and Debop use stochastic optimization to eliminate reliance on DD, reaching a trade-off between reduction and generality. BLADE differs from DomGad [25] and Debop [7] in that it focuses on speed, which in turn incentivizes test script intricacy that leads to high generality (as discussed in Sections III-D, IV-H). The efficiency also makes BLADE suitable for practical workloads, something which is not catered by DomGad [25] and Debop [7]. A concurrent work proposed the use of Fuzzer and static analysis-based methods to prevent overfitting the training cases [26]. Similar methods can be adopted to enhance the training case generation for BLADE .

IR & binary debloaters: Many other techniques (that do not target the source code) for software debloating exist as well. Some prominent ones include Java bytecode debloaters such as JShrink [27], which conducts static and dynamic extended call graph analysis on Java bytecode to perform transformations, and J-Reduce [28], which improves upon delta debugging to efficiently reduce Java bytecode. IR debloaters include Trimmer [5] and OCCAM [4] which perform reduction on the LLVM IR [24] of a program by using techniques such as input specialization, loop unrolling, and constant propagation. Another set of techniques tries to directly reduce the size of binary executable file [2]. One major drawback of this class of approaches is that it does not support auditability.

Library debloaters: There also exist techniques which target unused library functions. Nibbler [6] is one such approach that debloats unused functions in shared libraries. Piecewise [29] is yet another approach that targets both shared and static library code used by a program. It identifies inter-module dependencies and removes the unused code between them. Such techniques are indirect and may be used in conjunction with BLADE and other source code debloating techniques.

Miscellaneous: Apart from these, there exists a body of work [30]–[40] that looks into debloating code at different starting points (e.g. javascript, browsers, apps, configuration files, etc). There is also a concurrent work that performs debloating

comparison [41].

VII. Conclusion

We proposed a new source code debloating framework, BLADE , that leverages the structure of programming languages to efficiently perform code reduction to allow it to scale well with larger codebases. It meets all of the goals set out for an effective debloater (reduced code size and attack surface, fast reduction, maintenance of correctness, and auditability). We report that BLADE exhibits a marked improvement from previous approaches of program debloating in terms of achieving scalability (debloating large workloads).

Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) under Grant ACI-1440800 and the Office of Naval Research (ONR) under Contracts N68335-17-C-0558 and N00014-18-1-2660. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

References

[1] 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 '18, (New York, NY, USA), p. 380–394, Association for Computing Machinery, 2018.

[2] 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*, SEC'19, (USA), p. 1733–1750, USENIX Association, 2019.

[3] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, and S. W. Kim, "A survey on resource management in iot operating systems," *IEEE Access*, vol. 6, pp. 8459–8482, 2018.

[4] G. Malecha, A. Gehani, and N. Shankar, "Automated software winnowing," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, (New York, NY, USA), p. 1504–1511, Association for Computing Machinery, 2015.

[5] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, *TRIMMER: Application Specialization for Code Debloating*, p. 329–339. New York, NY, USA: Association for Computing Machinery, 2018.

[6] 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 '19, (New York, NY, USA), p. 70–83, Association for Computing Machinery, 2019.

[7] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Program debloating via stochastic optimization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, (New York, NY, USA), p. 65–68, Association for Computing Machinery, 2020.

[8] 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 '12, (New York, NY, USA), p. 335–346, Association for Computing Machinery, 2012.

[9] 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 '18, (New York, NY, USA), p. 361–371, Association for Computing Machinery, 2018.

[10] A. Zeller, "Yesterday, my program worked. today, it does not. why?," *SIGSOFT Softw. Eng. Notes*, vol. 24, p. 253–267, oct 1999.

[11] D. Knuth, "Semantics of context-free languages," in *Semantics of context-free languages*, pp. 127–145, Mathematical systems theory, 1968.

[12] "Clang documentation. https://clang.llvm.org/docs/,"

[13] "Ropgadget. https://github.com/JonathanSalwan/ROPgadget,"

[14] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, (New York, NY, USA), p. 254–264, Association for Computing Machinery, 2012.

[15] "Cve-2011-4089. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4089,"

[16] "Cve-2017-18018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-18018,"

[17] "Cve-2014-9471. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9471,"

[18] "Cve-2015-1345. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1345,"

[19] "Cve-2005-1228. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1228,"

[20] "Cve-2005-1039. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1039,"

[21] "Cve-2015-1865. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1865,"

[22] "Cve-2013-0221. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0221,"

[23] "Cve-2013-0222. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0222,"

[24] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, (USA), IEEE Computer Society, 2004.

[25] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Subdomain-based generality-aware debloating," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, (New York, NY, USA), p. 224–236, Association for Computing Machinery, 2020.

[26] 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*, ASE '22, (New York, NY, USA), Association for Computing Machinery, 2023.

[27] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, *JShrink: In-Depth Investigation into Debloating Modern Java Applications*, p. 135–146. New York, NY, USA: Association for Computing Machinery, 2020.

[28] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 556–566, Association for Computing Machinery, 2019.

[29] A. Quach, A. Prakash, and L. Yan, "Debloating software through Piece-Wise compilation and loading," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 869–886, USENIX Association, Aug. 2018.

[30] R. Williams, T. Ren, L. De Carli, L. Lu, and G. Smith, "Guided feature identification and removal for resource-constrained firmware," vol. 31, (New York, NY, USA), Association for Computing Machinery, dec 2021.

[31] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2c: Fine-grained configuration-driven system call filtering," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, (New York, NY, USA), p. 1243–1257, Association for Computing Machinery, 2022.

[32] X. Tërnava, M. Acher, L. Lesoil, A. Blouin, and J.-M. Jézéquel, "Scratching theandnbsp;surface ofandnbsp;./configure: Learning theandnbsp;effects ofandnbsp;compile-time options onandnbsp;binary size andandnbsp;gadgets," in *Reuse and Software Quality: 20th International Conference on Software and Systems Reuse, ICSR 2022, Montpellier, France, June 15–17, 2022, Proceedings*, (Berlin, Heidelberg), p. 41–58, Springer-Verlag, 2022.

[33] H. Zhang, M. Ren, Y. Lei, and J. Ming, "One size does not fit all: Security hardening of mips embedded systems via static binary debloating for shared libraries," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), p. 255–270, Association for Computing Machinery, 2022.

[34] T. Kupoluyi, M. Chaqfeh, M. Varvello, W. Hashmi, L. Subramanian, and Y. Zaki, "Muzeel: A dynamic javascript analyzer for dead code elimination in today's web," 2021.

[35] A. Turcotte, E. Arteca, A. Mishra, S. Alimadadi, and F. Tip, "Stubbifier: Debloating dynamic server-side javascript applications," 10 2021.
[36] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," 2021.
[37] M. Chaqfeh, R. Asim, B. AlShebli, M. F. Zaffar, T. Rahwan, and Y. Zaki, "Towards a world wide web without digital inequality," *Proceedings of the National Academy of Sciences*, vol. 120, no. 3, p. e2212649120, 2023.
[38] U. Naseer, T. Benson, and R. Netravali, "Webmedic: Disentangling the memory-functionality tension for the next billion mobile web users," in *HotMobile 2021 - Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, HotMobile 2021 - Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications, pp. 71–77, Association for Computing Machinery, Inc, Feb. 2021. Publisher Copyright: © 2021 ACM.; 22nd International Workshop on Mobile Computing Systems and Applications, HotMobile 2021 ; Conference date: 24-02-2021 Through 26-02-2021.
[39] J. Huang, Y. Aafer, D. Perry, X. Zhang, and C. Tian, "Ui driven android application reduction," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, p. 286–296, IEEE Press, 2017.
[40] J. Wu, R. Wu, D. Antonioli, M. Payer, N. O. Tippenhauer, D. Xu, D. J. Tian, and A. Bianchi, "LIGHTBLUE: Automatic Profile-Aware debloating of bluetooth stacks," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 339–356, USENIX Association, Aug. 2021.
[41] M. Ali, M. Muzammil, F. Karim, A. Naeem, R. Haroon, M. Haris, H. Nadeem, W. Sabir, F. Shaon, F. Zaffar, V. Yegneswaran, A. Gehani, and S. Rahaman, "A tale of reduction, security and correctness: Evaluating program debloating paradigms and their compositions," in *Computer Security – ESORICS 2023*, Springer International Publishing, 2023.

APPENDIX

Reductions: The following three subsections present the executable code, basic blocks, and lines of code reduction evaluation.

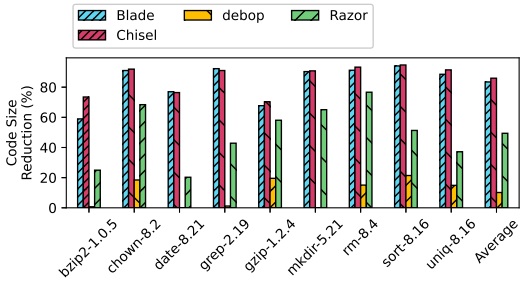

Fig. 9: Comparison of executable code size reduction of BLADE , Chisel, Debop, and Razor

*1) **Executable Code Size**:* We chose to measure the executable code size to make a comparison fair with Razor [2], which only performs a reduction in the executable portion of the binary file. The debloated programs produced by BLADE , Chisel, and Debop are compiled to get a working binary. We then measured the executable section of the binary for this evaluation. For Debop, Chisel, and BLADE , this was the *text section*. Moreover, we compiled the original program and debloated the resultant binary using Razor. We noticed that Razor retains the original .text section (but makes it read-only) and creates a new executable .text section (labelled .mytext) during its reduction process. Razor's *.mytext section* was measured for its evaluation.

Figure 9 demonstrates that the executable code size produced by BLADE and Chisel are roughly similar. On average, BLADE gives a reduction of 83%, compared with Chisel's 86%, Razor's 50%, and Debop's 10%.

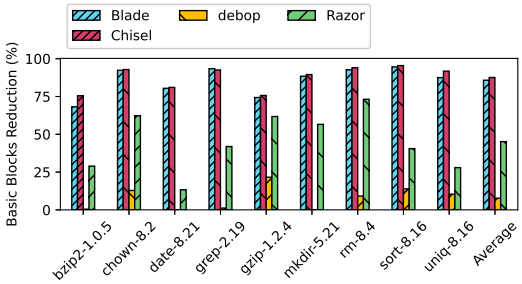

Fig. 10: Comparison of basic blocks reduction of BLADE , Chisel, Debop, and Razor

*2) **Basic Blocks**:* We also measure the basic blocks from the executable section of the binary, the results of which are shown in Figure 10. On chisel benchmarks, BLADE produces an average reduction of 86% compared with Chisel's 88%, Razor's 45%, and Debop's 7%. Compared to Razor and Debop, our approach performs significantly better. Whereas, our approach produces a comparable reduction to Chisel.

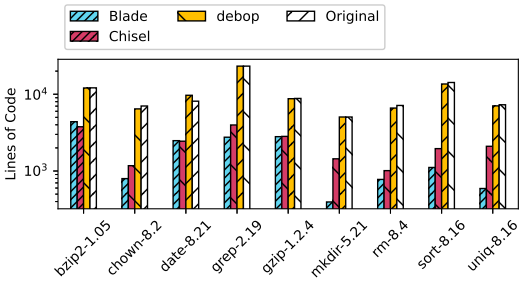

Fig. 11: Comparison of lines of code of debloated programs against the original baseline

*3) **Lines of code**:* Figure 11 illustrates that as compared to Chisel and Debop, debloating with BLADE results in fewer lines of code. On average BLADE removes 83% of the original lines of code, while Chisel removes 76% lines of code, and Debop only removes 1% lines of code (Figure 11).

The largest percentage difference in the lines of code between BLADE and Chisel is seen for mkdir, in which BLADE removes 92% of the original lines of code, as compared to 71% by Chisel. To put this into perspective, the resultant mkdir code produced by BLADE has 72.7% fewer lines of code than the output produced by Chisel (a difference of 1050 lines of code).

We observe a marked improvement in the reduction of lines of code from Chisel to BLADE because of two main factors that Chisel does not consider but BLADE does: 1) The reduction of unused declarations and 2) reduction within structs. Both these factors contribute significantly to the lines of code.

The arguments used in the test scripts are given in Table VI for Chisel benchmarks.

Fig. 12: Test Oracle: Snippet taken from gzip

```
function run-c() {
  { timeout $TIMEOUT_LIMIT $REDUCED_BINARY $1 <train/$2 > $2out.tmp; } &>$LOG || exit 1
   timeout $TIMEOUT_LIMIT $ORIGINAL_BINARY $1 <train/$2 > $2outog.tmp
   cmp $2outog.tmp $2out.tmp >&/dev/null || exit 1
    rm -rf *.tmp
    return 0
}
function run-d() {
    timeout $TIMEOUT_LIMIT $ORIGINAL_BINARY -c <train/$2 > $2outog.tmp
    { timeout $TIMEOUT_LIMIT $REDUCED_BINARY $1 <$2outog.tmp > $2out.tmp; } &>$LOG  || exit 1
    cmp train/$2 $2out.tmp >&/dev/null || exit 1
    rm -rf *.tmp
    return 0
}
# The following function represents the functionality checker of a test oracle
function args_test() {
    cd $DIR

  for file in $(ls train/) ; do
    run-c "-c" $file   || exit 1
    run-d "-d" $file   || exit 1
  done

return 0
}
# The following code represents the fitness checker of a test oracle.
# It is taken from Chisel's implementation
sanitizers=("-fsanitize=cfi -flto -fvisibility=hidden" "-fsanitize=address"
  "-fsanitize=memory -fsanitize-memory-use-after-dtor"
  "-fno-sanitize-recover=undefined,nullability"
  "-fsanitize=leak")

function compile() {
  cd $DIR
  case $COV in
  1) CFLAGS="-w -fprofile-arcs -ftest-coverage --coverage $BIN_CFLAGS" ;;
  *) CFLAGS="-w $1 $BIN_CFLAGS" ;;
  esac
  $CC $C_FILE $CFLAGS -o $REDUCED_BINARY >&$LOG || exit 1
  return 0
}
```

Program	Arguments
bzip2-1.0.5	-d -fc -t
sort-8.16	-d -s -r ; flagless
uniq-8.16	-i -u -d ; flagless
chown-8.2	-v -R ; flagless
gzip-1.2.4	-c -d
rm-8.4	-r -f
grep-2.19	-F -E -v -i ; flagless
mkdir-5.2.1	-m -p ; flagless
date-8.21	–date -d -rfc -r -f ; flagless

TABLE VI: Arguments tested in the test scripts written for our evaluation. *flagless* indicates test script included default argument-less functionality.

```
file1: ASCII text, with CRLF line terminators
file2: Unicode text, UTF-8 (with BOM) text, with CRLF line terminators
file3: ASCII text, with very long lines (65536), with no line terminators
file4: a.out little-endian 32-bit demand paged pure executable
file5: bzip2 compressed data, block size = 100k
file6: Lisp/Scheme program, ISO-8859 text, with CRLF line terminators
file7: ASCII text, with CRLF line terminators
file8: HTML document, ASCII text, with very long lines (17001)
file9: ASCII text
file10: data
file11: troff or preprocessor input, ASCII text
file12: C source, ASCII text
file13: JPEG image data, JFIF standard 1.01
file14: POSIX shell script, ASCII text executable
```

Fig. 13: The different file types used in the test oracle to support high coverage for generality.