SoK: A Tale of Reduction, Security, and Correctness - Evaluating Program Debloating Paradigms and Their Compositions

Muaz Ali[1], Muhammad Muzammil[2], Faraz Karim[3], Ayesha Naeem[4], Rukhshan Haroon[5], Muhammad Haris[6], Huzaifah Nadeem[7], Waseem Sabir[6], Fahad Shaon[8], Fareed Zaffar[6], Vinod Yegneswaran[9], Ashish Gehani[9], and Sazzadur Rahaman[1]

[1] University of Arizona, [2] Stony Brook University, [3] Georgia Institute of Technology, [4] Boston University, [5] Tufts University, [6] LUMS, [7] University of Pittsburgh, [8] Data Security Technologies, [9] SRI International

Abstract. Automated software debloating of program source or binary code has tremendous potential to improve both application performance and security. Unfortunately, measuring and comparing the effectiveness of various debloating methods is challenging due to the absence of a universal benchmarking platform that can accommodate diverse approaches. In this paper, we first present DEBLOATBENCHA[1], an extensible and sustainable benchmarking platform that enables comparison of different research techniques. Then, we perform a holistic comparison of the techniques to assess the current progress.

In the current version, we integrated four software debloating research tools: CHISEL, OCCAM, RAZOR, and PIECE-WISE. Each tool is representative of a different class of debloaters: program source, compiler intermediate representation, executable binary, and external library. Our evaluation revealed interesting insights (i.e., hidden and explicit tradeoffs) about existing techniques, which might inspire future research. For example, all the binaries produced by OCCAM and PIECE-WISE were correct, while CHISEL significantly outperformed others in binary size and Gadget class reductions. In a first-of-its-kind composition, we also combined multiple debloaters to debloat a single binary. Our performance evaluation showed that, in both ASLR-proof and Turing-complete gadget expressively cases, several compositions (e.g., CHISEL-OCCAM, CHISEL-OCCAM-RAZOR) significantly outperformed the best-performing single tool (i.e., CHISEL).

Keywords: Program Debloating, Debloating Comparison, Benchmark

1 Introduction

With the growing success of the software industry, and the availability of several competing platforms to support, modern software has become bloated [23]. Since most of the software are monolithic by design, they may contain functionalities

[1] Debloating benchmark for applications.

that end-users do not need. This extra functionality can not only cause performance issues but also become a security risk [11, 38, 53]. To avoid these issues, BusyBox [1] offers manually debloated, resource-optimized clones of software with minimal functionality, which are specially suited for embedded systems.

There has been an extensive research to automate this process [4–9, 13, 16, 17, 22, 26, 29, 31, 35–37, 39–42, 44–46, 50, 52]. Given, a *target program* and *its deployment context*, an automatic program debloating (a.k.a *software specialization*) tool debloats the program itself [5–9, 13, 16, 17, 22, 26, 31, 33, 35–37, 39, 40, 42, 44–46, 52] or its execution environment (e.g., Kernel [4, 29, 41], Firmware [50], etc.) *automatically*. While the main goal of all the automated software debloating tools is to remove as much unnecessary code as possible while attempting to preserve the intended functionality, it still remains challenging to properly evaluate the correctness and compare their performance. The primary reason behind this is – due to the lack of a unified benchmark platform, path to setting up diverse tools for analysis is unclear and seems expensive [39].

Platform for evaluation. This paper takes the first step towards creating an extensible and sustainable benchmarking framework named DebloatBench$_A$, for automated software debloating. We designed DebloatBench$_A$ to evaluate program debloaters targeting software written in C/C++, since most of the efforts in software debloating [5–8, 13, 22, 31, 36, 37, 39, 42, 44–46, 52] are dedicated to C/C++ domain. This design choice also enables the broadest impact, as applications written in C/C++ are more widely used [25], and offer more attack surface than others [10, 12, 43].

While our DebloatBench$_A$ framework is more general, the current version integrates a set of software debloating research tools that represent four different classes: Chisel [22] (source code), Occam [31, 33] (compiler intermediate representation), Razor [37] (executable binary) and Piece-wise [39] (external library). Our current target program suite contains 10 coreutils programs from ChiselBench [22], 3 graphical user interface (GUI)-based, and 2 network-facing programs. Note that, DebloatBench$_A$ is designed to provide an *easy-to-use* command-line interface to run the different debloating tools that are integrated into it. DebloatBench$_A$ can be downloaded and run with as few as 3 commands. For wide applicability, the ease of adding new tools or enhancing the target application suite is a core requirement. So among other things, DebloatBench$_A$ is customizable and extensible by design.

Evaluating program debloating methods. To inform and inspire future research, we conducted a thorough study with the set of debloaters integrated with DebloatBench$_A$. In particular, we examined the correctness, and changes in memory usage, on-disk size, security-relevant gadgets, and running time of the binaries produced by the debloaters. We found the tools based on static analysis (e.g., Occam, Piece-Wise) produced binaries that passed all tests while binaries from debloaters that used dynamic analysis (e.g., Chisel, Razor) failed an increasing number of tests as the aggressiveness of debloating was raised. Chisel failed to generate binaries for all non-coreutils programs. Razor failed on 3 out of 5 non-coreutils binaries too. This indicates a fundamental limita-

tion of test-case-driven debloaters to handle GUI-based or network-facing programs. Our gadget analysis focused on ASLR-proof attack bootstrapping and Turing-complete categories of micro-gadget classes [24]. There was an inverse relationship between the average correctness of a tool's output and its debloating effectiveness (measured by in-memory and on-disk resource usage) and gadget reduction in the binaries derived. A surprising finding was that PIECE-WISE increases binary size while unable to reduce any gadget classes. RAZOR increases binary size in general and could not reduce any gadget classes for non-coreutils programs. Three of the four tools ran quickly enough that they could be integrated into a software staging workflow, while the fourth took several orders of magnitude longer, making it impractical for most use cases. Leveraging this insight, we created the first-of-its-kind compositions of multiple tools to debloat a single binary. Our experimental evaluation on 10 coreutils indicates that compositions can achieve better reductions of gadget classes than the best single tool. It is worth noting that, DEBLOATBENCH$_A$ is an outcome of several years of a group of junior and senior researchers' effort. As part of the development of DEBLOATBENCH$_A$ and evaluation of program debloating tools, we directly fixed and reported several bugs in CHISEL and OCCAM and invented a new technique to measure ROP gadgets for PIECE-WISE.

Our contributions can be summarized as follows:

- We develop a new *easy-to-extend* framework named DEBLOATBENCH$_A$ to evaluate software debloating techniques. We created a set of 82 different variants of 10 unix programs/coreutils, 3 GUI-based, and 2 network-facing applications (total 15) for robust analysis. We integrated four different tools (i.e., CHISEL, OCCAM, RAZOR and PIECE-WISE) covering four different classes of debloaters. We are in the process of open-sourcing DEBLOATBENCH$_A$.
- We perform a holistic comparative analysis of these four debloaters under various metrics. Our evaluation shows that all the binaries produced by OCCAM and PIECE-WISE were correct. In contrast, CHISEL significantly outperformed others in binary size and Gadget class reductions, while failing to produce correct binaries for all non-coreutils programs.
- To leverage the strength of multiple debloaters, we performed a novel composition analysis in which we created several pipelines to use multiple tools to debloat a single binary. Our performance evaluation of tool composition showed that, in both ASLR-proof and Turing-complete gadgets expressively cases, several compositions (e.g., CHISEL-OCCAM, CHISEL-OCCAM-RAZOR) significantly outperformed the best single tool (CHISEL).

2 Debloating Methods

DEBLOATBENCH$_A$ supports application- and library-level software debloating. Since, evaluating kernel-level debloating would require a different set of machinery than application- and library-level debloating, we exclude it from our benchmark. Table 1 summarizes various application- and library-level program debloaters highlighting the tools that are included in our benchmark.

Debloating Class	Tools	Target		Type			Input			Analysis		Automated?	Opensourced?
		Application	Library	Source	LLVM IR	Binary	Configuration	Testcases	Annotation	Static	Dynamic		
Source	CHISEL [22]	✓	-	✓	-	-	-	✓	-	✓	-	●	✓
	C-Reduce [42]	✓	-	✓	-	-	-	✓	-	✓	-	●	✓
	Perses [46]	✓	-	✓	-	-	-	✓	-	✓	-	●	✓
	DomGad [52]	✓	-	✓	-	-	✓	-	-	✓	-	●	✓
	DeBop [51]	✓	-	✓	-	-	✓	-	-	✓	-	●	✓
LLVM IR	Occam [31, 33]	✓	✓*	-	✓	✓	✓	-	-	✓	-	●	✓
	Trimmer [6, 7, 44]	✓	-	-	✓	-	✓	-	-	✓	-	●	-
	LLPE [45]	✓	-	-	✓	-	✓	-	✓	✓	-	◐	✓
	LMCAS [8]	✓	-	-	✓	-	✓	-	-	✓	-	●	-
Binary	Razor [37]	✓	✓*	-	-	✓	-	✓	-	-	✓	●	✓
	Ancile [13]	✓	✓	-	-	✓	✓	✓	-	-	✓	●	-
Library	Piece-Wise [39]	-	✓	✓	-	-	-	-	-	✓	-	●	✓
	BlankIt [36]	-	✓	-	✓	-	-	-	-	-	✓	●	-
	Nibbler [5]	-	✓	-	-	✓	-	-	-	✓	-	●	-

Table 1. Comparison of different classes of program debloaters in terms of their target, analysis method, level of automation, and the availability. Here, "✓" indicates *yes* or *supported*, "✓*" indicates *experimental feature*, "-" indicates *not supported* or *no*. "●" indicates *fully automated* and "◐" indicates *partially automated.*

2.1 Application-level program debloating.

Application-level program debloaters can be categorized as follows: source-level [22], intermediate representation (IR)-level [31, 33] and binary-level [37] debloaters.

Source-level program debloating. Most of the source-level program debloating methods (e.g., CHISEL [22], C-REDUCE [42], and PERSES [46]) use variants of the delta-debugging algorithm for debloating. Delta-debugging uses a set of testcases to encompass the usage profile of the program after debloating, which has a potential to over-fit [52]. To address this issue, Xin *et al.* proposed more conservative methods (e.g., DOMGAD [52], DEBOP [51]) to preserve functional generality. *However, as acknowledged by Xin et al. [52],* CHISEL *[22] represents the state-of-the-art, in terms of code reduction performance. Therefore, we included* CHISEL *in our benchmark as the candidate representative for source-level program debloating tools.*

IR-level program debloating. Existing IR-based program debloating tools operate on LLVM bitcode and leverages partial evaluation for code reduction. For example, OCCAM [31, 33] combines partial evaluation and type thoery to remove unnecessary code. It supports cross-module analysis with multiple passes – first, summarizing cross-module dependencies and then using it for specialization. Similarly, TRIMMER [6, 7, 44] , LLPE [45], LMCAS [8] also uses partial evaluation as the core technique for specialization. *As* OCCAM *is the only open-sourced tool supporting automated analysis, we included* OCCAM *as the representative for IR-level program specialization tools.*

Binary-level program debloating. Program debloating methods for executable binaries rely on execution tracing, triggered by carefully chosen testcases (e.g., Razor [37]) or fuzzing (e.g., Ancile [13]). For example, Razor first runs the binary with the given test cases and uses *Tracer* to collect execution traces. It then decodes the traces to construct the program's CFG, which contains only the executed instructions. In addition to debloating context (i.e., intended functionalities), Ancile [13] requires a set of testcases to seed the fuzzer. *Because of being open source, we included* Razor *in our benchmark to represent binary-level program debloaters.*

2.2 Library-level program debloating.

Library-level program debloating has three flavors – *i*) static [5], *ii*) load-time [39] and *iii*) runtime-debloating [36]. Given a set of applications, static debloating tools (e.g., Nibbler [5]) debloats dynamically linked libraries statically, which replaces the original set of libraries permanently. Load-time debloaters redact (e.g., Piece-wise[39]) functions while loading the target library into the memory. Runtime debloaters load (e.g., BlankIt [36]) certain functions only if they are required at runtime. *For our evaluation, we chose* Piece-wise*, since its the only tool whose code is opensourced.*

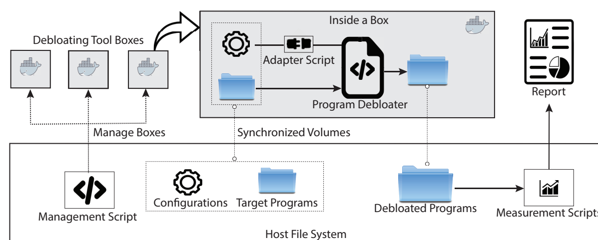


Fig. 1. DebloatBench$_A$ framework overview.

3 Components of the DebloatBench$_A$ Framework

In this section, we describe the components of the DebloatBench$_A$ framework.

Framework Overview. Figure 1 provides an overview of our DebloatBench$_A$ framework. Our design followed the open-close principle [32] to ensure extensibility without affecting usability and sustainability. We adopted a container-based approach to building DebloatBench$_A$ framework to provide isolation of environments across different debloaters. We created separate containers for each of them. We use a command-line tool-based management system (known as *orchestrator*) to build and manage the life cycle of these containers. Each input program in DebloatBench$_A$ has a corresponding configuration file describing various metadata (e.g., testcase location, build script location, etc.) about the program. Different program specialization tools use different formats for input program metadata. Each container has a corresponding adapter script to convert DebloatBench$_A$'s input program configuration files into its own format.

Table 2. Target application suite and their deployment contexts considered in DebloatBench$_A$. All the cells with different flags represents *the set of deployment contexts* named as variants (82 in total). The first 10 applications are coreutils, next 3 applications are GUI-based and the rest of the 2 are network based applications.

	Selected Arguments and Combinations						#Variants (82)
bzip2	-fc	-kc	-ksc	-ksfc	-sc	-sfc	6
chown	-c	-R	-Rc	-Rv	-v		5
mkdir	-m a=r	-m a=rw	-m a=rwx	-mp a=r	-mp a=rw	-mp a=rwx	6
sort	-c	-cf	-cfn	-cfr	-cn	-cr	11
	-f	-fn	-fr	-n	-r		
uniq	-c	-cd	-cdw N	-cu	-cuw N	-cw N	11
	-d	-dw N	-u	-uw N	-w N		
grep	-v	-E	-F	-i	-m		5
gzip	-c	-d	-f	-t			4
tar	-cf	-tvf	-xf				3
date	-d	-u	-r	-Rd	-ud		5
rm	-r	-f	-rf	-i			4
gm	negate	monochrome	flop	flip	contrast		5
vlc	noaudio	loop	fullscreen	starttime	novideo		5
gv	scale	noantialias	fullscreen	color			4
putty	telnet	ssh	m	load			4
nginx	-t	-s	-p	-c			4

3.1 Framework Components

There are three major components in DebloatBench$_A$, i.e., input programs, debloating tools, orchestrators, and measurement scripts. Input programs, orchestrators and measurement scripts reside in the host file system. Debloaters and their corresponding adapter scripts reside inside isolated containers.

Debloating Tools. Each of the debloaters in DebloatBench$_A$ are built within an isolated docker container. Container images freeze the execution environment. While each tool requires an input program and corresponding metadata to perform debloating, they use different means to accept those inputs (Table 1). DebloatBench$_A$ uses a configuration file to collect inputs corresponding to an input program. We created scripts to parse DebloatBench$_A$'s configuration file and generate inputs for individual tools. These scripts are called *adapter scripts*. Adapter script of a tool bridges it with DebloatBench$_A$.

Target Program Suite. In our current version, we selected 10 coreutils/linux utility programs from ChiselBench [22] on which all the selected tools run correctly to produce meaningful results. To evaluate their generalizability, we selected 3 GUI-based and 2 network-facing programs. With the goal of capturing diversity, we chose a diverse of deployment contexts for each of the applications. We term the combination of a target application and a specific deployment context a *variant*. Table 2 summarizes the set of 82 variants that constitute the complete workload.

Testcases. With the target program to debloat, program debloaters also take testcases (i.e., Chisel, Razor, Piece-wise) or a configuration file (i.e., Oc-

CAM) as input. Note that, the number of test cases impacts the training time for test case-dependent tools, hence it is important to pick quality test cases to maximize the coverage without impacting the performance. Given the application's running configuration, generating high-quality test-cases is an active area of research. Since this is an orthogonal problem, such automation is beyond the scope. Therefore, we relied on manually created test cases. To produce binaries with CHISEL, RAZOR, and PIECE-WISE, we created a set of 726 test cases. We also created a total of 1007 number of testcases to check the correctness[2]. They are summarized in Table 8 in Appenfix. While preparing these testcases, we aimed to capture diverse behavior in order to maximize the coverage.

Measurement Scripts. We measure the performance of program debloaters with the following five metrics: *i*) correctness of the debloated binaries *ii*) decrease in binary size, *iii*) Security analysis in the lens of gadgets reduction and *iv*) debloating time. Note that, we did not use CVEs for security evaluation, mostly because CVEs are correlated with the functionalities. Elimination of them are more likely to be influenced by the selection of functionalities than a tool.

In-memory gadget counting. PIECE-WISE debloats external libraries in the unit of functions while loading into the memory. We use *gdb* to find missing *functions* in the debloated version loaded in the memory. After collecting that information, we create a new version of the library by replacing the missing function bodies with *NOP*s. Finally, we use this version of the library to collect ROP gadgets using the ROPgadget tool [3].

4 Experimental Setup in DEBLOATBENCH$_A$

As discussed in Section 2, for performance comparison, we incorporated the following four debloaters into DEBLOATBENCH$_A$, i.e., CHISEL [22], OCCAM [31, 33], RAZOR [37] and PIECE-WISE [39], which covers four different paradigms. Next, we discuss our experimental setup to evaluate them. We conducted two set of experiments to measure the performance of *i*) standalone tools and *ii*) their composition. Finally, we discuss the metrics that we used to compare performance.

4.1 Standalone Mode

Setting up CHISEL: From the CHISEL authors we learned that CIL [34] was used to merge the C files for the input programs in the earlier version of CHISEL. To run CHISEL successfully, we reused the merged C files for 10 coreutils programs from CHISELBENCH. For the other 5 large programs, we leveraged its build system integration functionality.

Setting up OCCAM: A wide range of policies is supported by OCCAM to debloat binaries. Each policy results in a different debloated binary ranging from *aggressive* to *no* specialization. After running a sanity checking experiment to find the best configuration, we selected the *onlyonce* for measuring and comparing OCCAM's performance.

[2] Some of the test cases are taken from Razor Benchmarks. [37]

Setting up Razor**:** Razor's performance is largely dependent on the choice of heuristic used by the *Pathfinder* module. Since, Razor is relatively faster than other tools, for Razor we created multiple version of binaries corresponding to each of the heuristics and selected the version with maximum correctness for performance analysis and comparison with other tools.

Setting up Piece-wise**:** For Piece-wise, we used the pre-built compiler and loader provided with the Docker container. We used `musl-libc` v1.1.15 as the library dependency for each of the input programs in our application suite and then debloated `musl-libc` with Piece-wise. To create non-Piece-wise compiled binaries, we used the same docker container that Piece-wise repository provides and downloaded unmodified LLVM and Clang along with `musl-libc`, with the exact same versions that Piece-wise used.

4.2 Composition Mode

Since various specialization tools in $\text{DebloatBench}_{\text{A}}$ operate on different forms of application code (i.e., source, IR, binary or library), it is possible to run multiple tools to debloat a single program. For example, Chisel debloats at the source code level, and the resulting binary can be further debloated using Razor, which performs debloating at the binary level. Bulding upon this idea, we formulate the following 4 unique compositions of tools and use them to debloat the $\text{DebloatBench}_{\text{A}}$'s input program suite: *i)* Chisel to Occam, *ii)* Chisel to Occam to Razor, *iii)* Chisel to Razor and *iv)* Occam to Razor.

As Piece-wise requires both source code and the binary to perform debloating, it can only be composed with Chisel. We also tried Piece-wise to Chisel pipeline with limited success that we discuss in Section 5.3. For a given metric, we compare the performance of compositions with the best-performing individual tools.

5 Evaluation of Debloaters

Research Questions: To understand the utility of software debloating tools, we considered the following issues. **RQ1:** Does a debloating approach adversely impact the correctness of target applications? **RQ2:** How effective is each debloater at reducing the size of individual programs? **RQ3:** What is the effect of debloating on the gadget-related security of target programs? **RQ4:** How usable are each of the debloating approaches in practice? **RQ5:** Does composing debloaters offer any further improvement?

We term the combination of a target application and a specific deployment context a *variant*. Table 2 summarizes the set of 82 variants that constitute the complete workload. Each variant gives rise to a different debloated binary. In the analyses below, a debloater is applied to all variants of a program, with the average result reported. *During our evaluation, we observed that the debloaters significantly failed to produce results for meaningful comparison on non-coreutils programs. Hence, we report their results separately for RQ1 to RQ4 and we only use 10 coreutils programs to answer RQ5.*

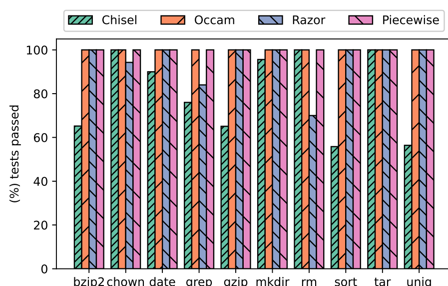


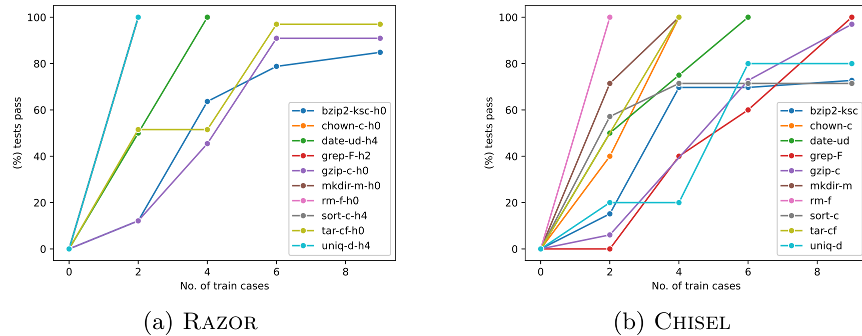
Fig. 2. Each debloater was applied to all variants of a target program. The average fraction of tests passed for each target is reported here.

5.1 Evaluation results on 10 coreutil programs

RQ1: Tool Correctness. We use testcases to measure the correctness of a given debloater, which implies that whether a specific debloated binary is *correct* is an under-estimate. Figure 2 reports the results of our correctness evaluation. The debloating approaches that employ static analysis – i.e., OCCAM and PIECE-WISE – passed 100% of the tests. In contrast, the debloaters that rely on dynamic analysis did not – CHISEL passed 80.4% of the tests, while RAZOR produced correct results for 94.8% of the cases. OCCAM produces the best correctness results because of its static partial evaluation-based approach that conservatively retains all the functionality for a given argument. CHISEL performs worse because of its overly-reliance on the provided testscripts.

We undertook the exercise of augmenting the training cases provided with each debloater for the target applications (Figure 3). Our experience indicated that a debloated binary created with more training cases retains more behavioral diversity, allowing it to pass more correctness tests. However, the level of improvement varied significantly from one target application to another. To quantify this, we report on the fraction of tests that passed for each of the targets as a function of the number of training cases utilized. The results for RAZOR and CHISEL are shown in Figure 3. These results are based on one variant of a given target program that represent the main functionality (e.g. chown's -c variant) as opposed to the average of all the variants. The general trend says that more training cases yielded increased debloating correctness.

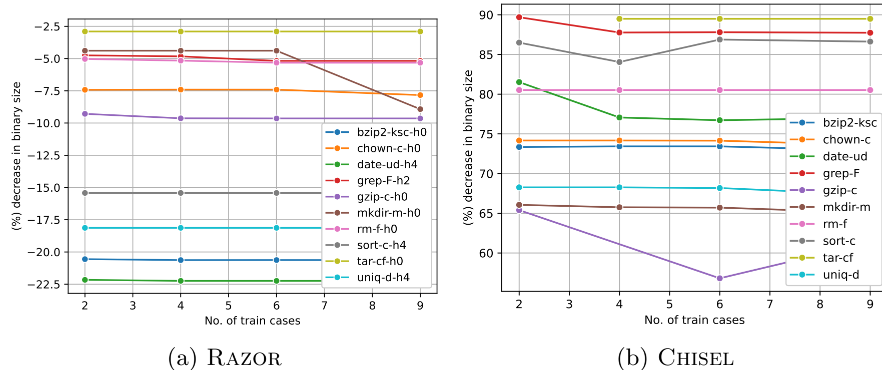
RQ2: Size Reduction. A primary goal of debloating a target application is to reduce its size by eliminating code that will not be used in a particular deployment. The effect is on the binary size on disk. CHISEL and OCCAM eliminate code at the source and compiler intermediate representation levels, respectively. This usually reduces the size of the resulting binaries. In contrast, RAZOR retains the original binary and extends it with transformed code, while PIECE-WISE adds metadata representing the program's control flow graph to the binary. These effects are easily observed in Figure 6(b). Since OCCAM's partial evaluation can increase the number of functions (when both unspecialized and specialized versions are retained), it occasionally increases code size. We also measure the effect



(a) RAZOR (b) CHISEL

Fig. 3. (a) The correctness of RAZOR's debloating is a function of the training cases used. The variant and heuristic level used is denoted in the legend. h0 indicates no heuristic used. Note that sort, uniq, rm, mkdir, and chown hit 100% correctness on two train cases. (b) The correctness of CHISEL's debloating is a function of the training cases provided to its oracle test script.

on binary size of varying the number of train cases for RAZOR and CHISEL as shown in Figure 4. We note that increasing the number of train cases can sometimes lead to a relative increase in size reduction for CHISEL (gzip, sort).



(a) RAZOR (b) CHISEL

Fig. 4. (a) The reduction in binary size on disk by RAZOR vs. the number of training cases used. (b) The reduction in binary size on disk by CHISEL vs. the number of training cases used. Note that positive values indicate reductions, while negative ones are size increases. For this experiment, the same data is used from Figure 3.

RQ3: Gadget Expressivity. Raw ROP gadget count and code size is not a reliable metric for estimating the vulnerability of a binary [14, 15]. Homescu *et al.* [24] argued that gadgets can be categorized into classes (based on the type of functionality provided), with just a single member from each class sufficing for the assembly of specific categories of attacks. They constructed classes of "micro-gadgets" (restricted to maximum lengths of 3 bytes) that provide the basis for each category. We report on the effect of debloating on the changes in the two categories of ASLR-proof and Turing-complete expressivity. These are reported in Figure 5. In both categories, CHISEL yields the highest reduction (of 28.2%

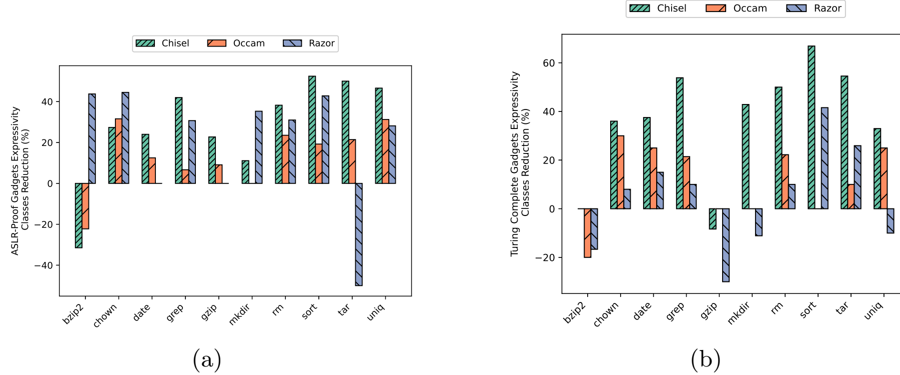


Fig. 5. (a) ASLR-proof ROP gadget expressivity and (b) Turing-complete ROP gadget expressivity reduction.

and 36.6% on average, respectively). For the Turing-complete category, Occam (11.4%) is more effective than Razor (4.3%). In contrast, Razor (20.6%) yields more reduction for the ASLR-proof category than Occam (13.3%). Finally, we applied Piece-wise to `musl libc`. In both the Turing-complete and ASLR-proof categories, there was no reduction in the number of classes, i.e., gadgets for 16 of 17 Turing-complete classes and 34 of 35 ASLR-proof classes were present.

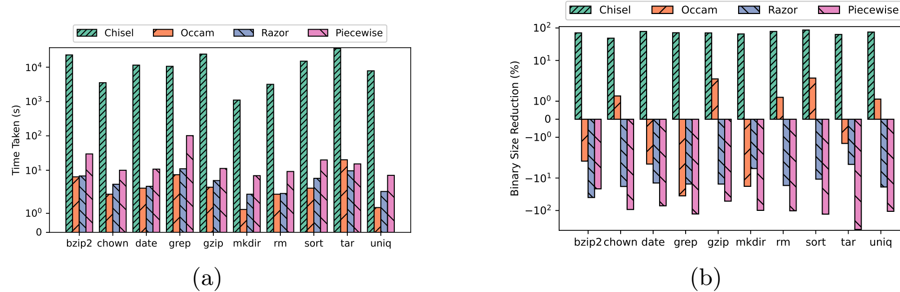


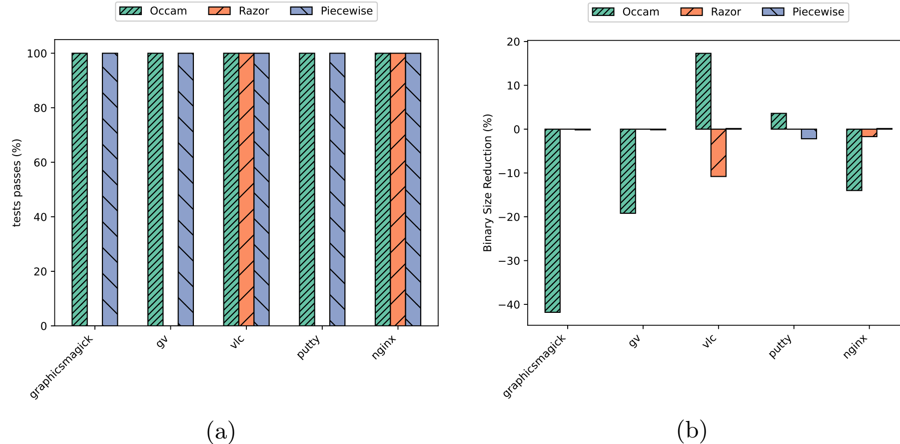
Fig. 6. (a) Average time to debloat all variants of a target application with each debloater. Note that the y-axis is logarithmic to accommodate the large differences in time taken. (b) Effect on binary size on disk after applying different debloaters to each target program. Note that positive decreases indicate binary size reduction.

RQ4: Tool Usability. To gain insight into the settings where each debloater could potentially be deployed, the time it takes to run on all the variants in our workload was measured. The results are in Figure 6 (a). Chisel takes several orders of magnitude (with an average of around 13,000 seconds) more time than the other debloaters. This limits the settings in which Chisel could be practically utilized. In contrast, the average time taken by Piece-wise, Occam, and Razor are 22.1, 4.9, and 5.2 seconds, respectively. This makes them usable in traditional optimization workflows. Piece-wise, Occam, and Razor take significantly less time than Chisel because they mainly rely on static analysis,

whereas CHISEL relies on Markov Decision Processes to find a minimal subset of statements satisfying the provided testcases.

Summary: Evaluation results on coreutils programs

- **Correctness:** CHISEL*: 80.4%,* OCCAM*: 100%, Razor: 94.8%,* PIECE-WISE*: 100%. Static analysis-based debloaters produce more correct binaries than dynamic analysis-based debloaters.*
- **Size on disk:** CHISEL *and* OCCAM *cause reductions. However, by extending the original binary,* RAZOR *and* PIECE-WISE *increases the size.*
- **Gadget Expressivity:** CHISEL *outperformed others in both categories: ASLR-proof (28.2%), and Turing-complete (36.6%).*
- **Debloat Time:** CHISEL *takes several orders of magnitude (3.75 hours) more than others. The average time taken by* PIECE-WISE*,* OCCAM*, and* RAZOR *are 22.1, 4.9, and 5.2 seconds, respectively.*



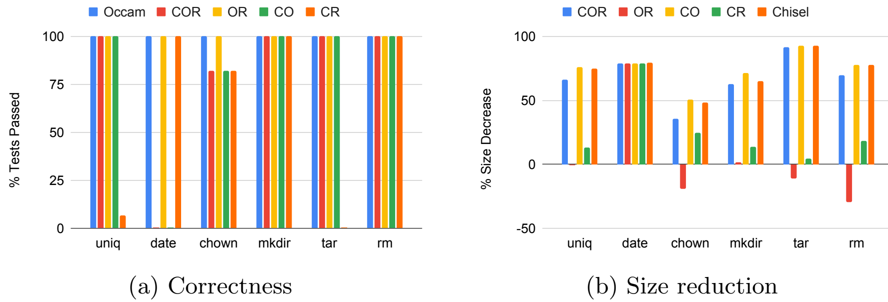
(a) (b)

Fig. 7. For non-coreutils programs: (a) Average fraction of tests passed. (b) Average binary size reduction.

5.2 Evaluation results on 5 non-coreutils programs.

Here, we present the evaluation results on non-coreutils programs. To evaluate the correctness **(RQ1)** we used 247 test cases for 22 variants of 5 non-coreutils programs. We debloated these applications inside containers. For their correctness testing, we used the host system because of the difficulty of handling GUI applications inside the containers. Our evaluation shows that only OCCAM and PIECE-WISE produced correct binaries that passed all the test cases. RAZOR produced correct binaries for two of them (vlc and nginx), while failing to debloat others. Finally, CHISEL failed on all of the target programs. Surprisingly, all of the debloaters increased binary size to some extent **(RQ2)**. Correctness and binary size reduction results are summarized in Figure 7.

The maximum average debloating time **(RQ3)** for PIECE-WISE was 129.67 seconds (on Graphicsmagick) and the minimum was 13.66 seconds (on Gvpdf).



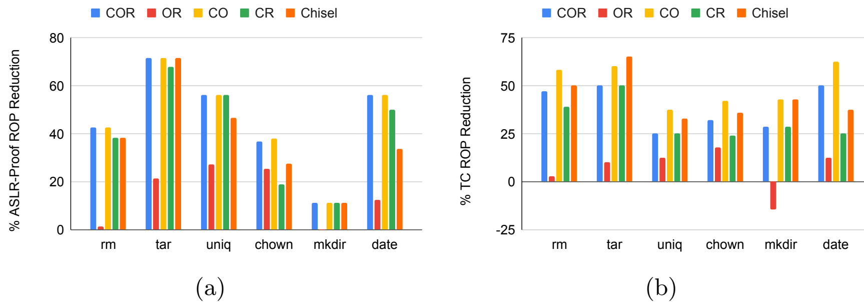
(a) Correctness (b) Size reduction

Fig. 8. (a) Average fraction of tests passed after applying a debloater composition to a target application. (b) Average binary size reduction using debloater composition.

Occam took an average maximum of 95.6 seconds (on Graphicsmagick) and a minimum of 1.3 seconds (on Vlc). Razor took an average maximum of 120 seconds. As summarized in Figure 10(c) and (d) in Appendix, only Occam removed gadgets classes for both ASLR-proof and Turing-complete categories **(RQ4)**, while Razor increased the number of Turing-complete gadget classes.

Summary: Evaluation on non-coreutils programs

- **Correctness:** *Dictated by static analysis-based tools (*Occam *and* PieceWise*).*
- **Size:** *All of them increased the binary size (except* Occam *on Vlc).*
- **Gadget Expressivity:** Occam *significantly outperformed others.*



(a) (b)

Fig. 9. (a) ASLR-proof ROP gadget expressivity reduction using debloater composition. (b) Turing-complete ROP gadget expressivity reduction using debloater composition.

5.3 RQ5: Debloater Composition

For meaningful comparison, we only used coreutils programs to evaluate debloater compositions. In Figures 8, and 9, the three-stage pipeline is referred to as COR to denote Chisel to Occam to Razor. Further, each pair of these three debloaters can be composed without the third one. We evaluated these three com-

binations as well. In the figures, they are denoted by OR for Occam to Razor, CO for Chisel to Occam, and CR for Chisel to Razor. After applying two or more debloaters in succession, the resulting binary must function correctly. Figure 8(a) reports the average fraction of tests passed after each combination of debloaters is applied to a target application. The correctness testing of individual debloaters (reported in Section 5.1) found that Chisel was the most likely to produce a binary that failed a check. These variants were eliminated when testing the composition of debloaters. On the other hand, Occam is used as a baseline since binaries derived from its output passed all tests.

The maximum average binary size reduction by any single tool is 70.4% (from Chisel). The combination of Chisel followed by Occam slightly outperforms it with an average reduction of 74.6%. (The combination of all three debloaters only yields an average reduction of 67.5%.) For the Turing-complete gadget expressivity case, the single tool with the maximum average improvement was again Chisel with 36.6%. However, in this case, the Chisel-Occam-Razor composition yielded a slightly better 38.8% while the Chisel-Occam combination provided a substantially better reduction of 50.5%. In the ASLR-proof gadget expressivity case, the single tool with the highest reduction was Chisel with 28.2% (Figure 9). Here, Chisel-Occam, Chisel-Razor, and Chisel-Occam-Razor combinations all outperformed with reductions of 45.9%, 40.4%, and 45.8%, respectively.

Chisel to Piece-wise Pipeline. After debloating with Chisel, out of 10 target programs, we were successful in compiling five of them with Piece-wise compiler. Among these programs, Piece-wise was not able to compile all the variants of `grep`, `date`, and `tar`. This is due to the incompatibility between `musl-libc` and `glibc`, where Chisel uses `glibc` and Piece-wise uses `musl-libc` for compilation.

Summary: Composition of Debloaters

- **Correctness:** *Produces correct binaries for non-*Chisel *pipelines.*
- **Size:** Chisel-Occam *pipeline outperformed the best individual tool (*Chisel*).*
- **Gadgets:** *In both ASLR-proof and Turing-complete gadget expressively cases, several compositions (e.g.,* Chisel-Occam*,* Chisel-Occam-Razor*) significantly outperformed the best tool (*Chisel*).*

6 Discussion

We first discuss the impact of design choices on the performance and the usability of program specialization tools in the light of our evaluation. Then, we discuss the limitations of this study.

6.1 Impact of Design Choices

Dependency on test-cases in Chisel: Our evaluation revealed that, in terms of the correctness of the debloated binary, Chisel is the weakest of all the

techniques. This is mostly because of CHISEL's strong reliance on test scripts that it uses to guide the debloating. Also, these scripts can be tricky to get right and CHISEL can misbehave sometimes even when the scripts are seemingly correct. In our experiments, on average, over 96% of the debloating time is spent in running the property test script.

Partial evaluation in OCCAM: Our experience shows that the partial evaluation [27] significantly reduced the usability of OCCAM. It only allows non-conflicting flags to be present in a debloated binary. We call two flags to be non-conflicing, if both of them can be used simultaneously in the same execution. So for two conflicting flags, one needs to create two variants. However, it is worth noting that configuration-based program debloating enabled by partial evaluation in OCCAM are easy to setup. This is because it does not require careful and tedious use of testcases, where the quality of the tests impacts the overall usability of the debloated binary.

Tracing-based reduction in RAZOR: A dominant trend in the analysis of RAZOR debloating was that the number of correctness tests passed would remain low for several heuristics (i.e., *no heuristic*, *zCode*, and *zCall*) and high for others (i.e., *zFunc* and *zLib*). This implies that the benefits of using different heuristics need to be fully assessed to choose the heuristic that produces the correct binary to retain reasonable functionality. Figure 10(a) illustrates the overall relationship between heuristic levels and the percentage of test cases passed in each coreutils program. Moreover, RAZOR's training time is dependent on the number of train cases provided to it. Figure 10(b) in Appendix shows how the number of train cases impacts the time taken to train RAZOR for each of the 10 target programs.

Load-time reduction in PIECE-WISE: At compile time PIECE-WISE computes the program dependency graph and appends it in the `.dep` section in the ELF header, which is used for reduction in load-time. This significantly increases the size. For some applications, the large increase in size can outweigh the benefit.

6.2 Limitations of this Study.

In the current version, DEBLOATBENCH$_{\text{A}}$ chose a single tool per category and provides an in-depth analysis. However, tool coverage can easily be extended. The current choice of selecting target applications was hindered by existing debloaters' capabilities, which can be extended too. We created an extensive number of test cases to maximize the coverage for training and testing, however, it is hard to guarantee.

7 Related Work

C/C++ program specialization. There are three broad classes of program specialization, i.e., source-level (e.g., CHISEL [22], C-REDUCE [42], and PERSES [46] and DOMGAD [52]), IR-level (e.g., TRIMMER [6, 7, 44] , LLPE [45], LMCAS [8], OCCAM [31, 33]) and binary-level (e.g., RAZOR [37]). Performance comparisons in most of these tools are done with either the state-of-the-art tools in their category or none. Library specialization tools (e.g., PIECE-WISE [39], BlankIt [36],

Nibbler [5]) also followed a similar trend. Razor [37] and LMCAS [8] are two exceptions. Razor compared its performance with Chisel on runtime, binary correctness as well as code, ROP gadget and CVE reduction. LMCAS [8] compared the runtime with Occam, Chisel and Razor, while the performance on other metrics were compared with Occam. To the best of our knowledge, DebloatBench_A is the first benchmark to systematically scrutinize tools across all the categories to underscore the strengths and weaknesses of each of the methods. Our evaluation also highlights that a composition of multiple methods has a great potential to achieve better performance than any of the individual tools.

Environment/OS-level debloating MultiK [29] and shard [4] offers application specific kernel-level debloating. Cimplifier [41] uses dynamic analysis to detect logically distinct applications inside a container and automatically breaks it into smaller containers. LightBlue [50] leverages static analysis to perform application-guided firmware debloating. CDE [20] leverages execution tracing to identify the dependencies of an application for seamless porting. In a concurrent work, recently, Hassan *et al.* developed a framework named DebloatBench_C to evaluate container debloaters [21].

Program specialization for other languages. Researchers have also explored program debloating for other languages. For example, Piranha [40] targets Objective-C. JShrink [16]; JSCleaner [17], Lacuna [35], Muzeel [30], Stubbifier [48] and [49] target JavaScript; JRed [26], JAX [47], BloatLid [18] Depclean [2] and [9] target Java- and PHP-based applications, respectively. A body of work exists on byte code reduction as well [19, 28].

8 Conclusion

We presented DebloatBench_A, an extensible and sustainable benchmarking framework for rigorous evaluation of program debloaters. We integrated Chisel, Occam, Razor and Piece-wise into the framework and performed a holistic comparative study. Our analysis shows that conservative static analysis tools produce correct binaries (e.g., Occam, Piece-wise), while aggressive dynamic analysis tools (e.g., Chisel) perform better in reducing size and gadget classes. A surprising finding was Piece-Wise failed to reduce any gadget classes while increasing the binary size. Also, test-cased-driven tools performed worse on non-coreutils programs. Our analysis of multi-tool composition at different stages opens up avenues for future explorations.

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, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR. We thank Muhammad Hassan, Abdullah Naveed, Talha Tahir, Muhammad Farrukh, and Ahsan Amin for their help in preparing and testing the large application suite.

Bibliography

[1] Busy box. `https://busybox.net/`

[2] Depclean. `https://github.com/castor-software/depclean`

[3] Ropgadget tool. `https://github.com/JonathanSalwan/ROPgadget`

[4] Abubakar, M., Ahmad, A., Fonseca, P., Xu, D.: Shard: Fine-grained kernel specialization with context-aware hardening. USENIX Security Symposium **28th** (2019)

[5] Agadakos, I., Jin, D., Williams-King, D., Kemerlis, V.P., Portokalidis, G.: Nibbler: debloating binary shared libraries. In: ACSAC. pp. 70–83 (2019)

[6] Ahmad, A., Anwar, M., Sharif, H., Gehani, A., Zaffar, F.: Trimmer: Context-Specific Code Reduction. 37th IEEE/ACM Conference on Automated Software Engineering (ASE) (2022)

[7] Ahmad, A., Noor, R., Sharif, H., Hameed, U., Asif, S., Anwar, M., Gehani, A., Zaffar, F., Siddiqui, J.: Trimmer: An Automated System For Configuration-Based Software Debloating. IEEE Transactions on Software Engineering (TSE) **48(9)** (2022)

[8] Alhanahnah, M., Jain, R., Rastogi, V., Jha, S., Reps, T.: Lightweight, multi-stage, compiler-assisted application specialization. In: 7th European Symposium on Security and Privacy. IEEE (2022)

[9] Azad, B.A., Laperdrix, P., Nikiforakis, N.: Less is more: Quantifying the security benefits of debloating web applications. USENIX Security Symposium **28th** (2019)

[10] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C., Kamsky, A., McPeak, S., Engler, D.R.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (2010)

[11] Bhattacharya, S., Rajamani, K., Gopinath, K., Gupta, M.: The interplay of software bloat, hardware energy proportionality and system bottlenecks. In: Hot-Power'11. pp. 1–5 (2011)

[12] Bierbaumer, B., Kirsch, J., Kittel, T., Francillon, A., Zarras, A.: Smashing the stack protector for fun and profit. In: Janczewski, L.J., Kutylowski, M. (eds.) ICT Systems Security and Privacy Protection. IFIP, vol. 529, pp. 293–306. Springer (2018)

[13] Biswas, P., Burow, N., Payer, M.: Code specialization through dynamic feature observation. In: Joshi, A., Carminati, B., Verma, R.M. (eds.) CODASPY '21. pp. 257–268 (2021)

[14] Brown, M.D., Pande, S.: Is less really more? towards better metrics for measuring security improvements realized through software debloating. In: 12th USENIX Workshop (CSET 19) (2019)

[15] Brown, M.D., Pruett, M., Bigelow, R., Mururu, G., Pande, S.: Not so fast: Understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. Proc. ACM Program. Lang. **5**(OOPSLA) (2021)

[16] Bruce, B.R., Zhang, T., Arora, J., Xu, G.H., Kim, M.: Jshrink: in-depth investigation into debloating modern java applications. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE. pp. 135–146. ACM (2020)

[17] Chaqfeh, M., Zaki, Y., Hu, J., Subramanian, L.: Jscleaner: De-cluttering mobile webpages through javascript cleanup. In: Huang, Y., King, I., Liu, T., van Steen, M. (eds.) WWW. pp. 763–773. ACM / IW3C2 (2020)

[18] Dewan, A., Rao, P., Sodhi, B., Kapur, R.: Bloatlibd: Detecting bloat libraries in java applications. In: 16th Conference on the Evaluation of Novel Approaches to Software Engineering (2021)

[19] GuardSquare: Proguard. `https://github.com/Guardsquare/proguard`

[20] Guo, P.J., Engler, D.R.: CDE: using system call interposition to automatically create portable software packages. In: Nieh, J., Waldspurger, C.A. (eds.) USENIX ATC (2011)

[21] Hassan, M., Tahir, T., Farrukh, M., Naveed, A., Naeem, A., Shaon, F., Zaffar, F., Gehani, A., Rahaman, S.: Evaluating container debloaters. In: IEEE Secure Development Conference, SecDev 2023, Atlanta, GA, USA, October 18-20, 2023. IEEE (2023)

[22] Heo, K., Lee, W., Pashakhanloo, P., Naik, M.: Effective program debloating via reinforcement learning. In: 2018 ACM CCS. pp. 380–394 (2018)

[23] Holzmann, G.J.: Code inflation. IEEE Softw. **32**(2), 10–13 (2015)

[24] Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., , Franz, M.: Microgadgets: Size does matter in Turing-Complete Return-Oriented programming. In: USENIX WOOT '12) (2012)

[25] Javed, F., Afzal, M.K., Sharif, M., Kim, B.S.: Internet of things (iot) operating systems support, networking technologies, applications, and challenges: A comparative review. IEEE CS&T **20**(3), 2062–2100 (2018)

[26] Jiang, Y., Wu, D., Liu, P.: Jred: Program customization and bloatware mitigation based on static analysis. In: IEEE COMPSAC. pp. 12–21 (2016)

[27] Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. **28**(3), 480–503 (1996)

[28] Kalhauge, C.G., Palsberg, J.: Logical bytecode reduction. In: ACM SIGPLAN PLDI. p. 1003–1016. ACM (2021)

[29] Kuo, H., Gunasekaran, A., Jang, Y., Mohan, S., Bobba, R.B., Lie, D., Walker, J.: Multik: A framework for orchestrating multiple specialized kernels. CoRR **abs/1903.06889** (2019)

[30] Kupoluyi, T., Chaqfeh, M., Varvello, M., Hashmi, W., Subramanian, L., Zaki, Y.: Muzeel: A dynamic javascript analyzer for dead code elimination in today's web. arXiv preprint arXiv:2106.08948 (2021)

[31] Malecha, G., Gehani, A., Shankar, N.: Automated Software Winnowing. 30th ACM Symposium on Applied Computing (SAC) (2015)

[32] Martin, R.C.: The open-closed principle. More C++ gems **19**(96) (1996)

[33] Navas, J., Gehani, A.: OCCAMv2: Combining Static and Dynamic Analysis for Effective and Efficient Whole Program Specialization. Communications of the ACM **66(4)** (2023)

[34] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) Conference on Compiler Construction (2002)

[35] Obbink, N.G., Malavolta, I., Scoccia, G.L., Lago, P.: An extensible approach for taming the challenges of javascript dead code elimination. In: Oliveto, R.,

Penta, M.D., Shepherd, D.C. (eds.) Conference on Software Analysis, Evolution and Reengineering (2018)

[36] Porter, C., Mururu, G., Barua, P., Pande, S.: Blankit library debloating: getting what you want instead of cutting what you don't. In: ACM SIGPLAN PLDI. pp. 164–180 (2020)

[37] Qian, C., Hu, H., Alharthi, M., Chung, P.H., Kim, T., Lee, W.: Razor: A framework for post-deployment software debloating. In: USENIX Security (2019)

[38] Quach, A., Erinfolami, R., Demicco, D., Prakash, A.: A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments. In: Kim, T., Wang, C., Wu, D. (eds.) Workshop on Forming an Ecosystem Around Software Transformation (2017)

[39] Quach, A., Prakash, A., Yan, L.: Debloating software through piece-wise compilation and loading. In: USENIX Security. pp. 869–886 (2018)

[40] Ramanathan, M.K., Clapp, L., Barik, R., Sridharan, M.: Piranha: reducing feature flag debt at uber. In: Rothermel, G., Bae, D. (eds.) ICSE-SEIP. pp. 221–230. ACM (2020)

[41] Rastogi, V., Davidson, D., Carli, L.D., Jha, S., McDaniel, P.D.: Cimplifier: automatically debloating containers. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) European Software Engineering Conference / Foundations of Software Engineering (2017)

[42] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: ACM PLDI. pp. 335–346 (2012)

[43] Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) ACM CCS 2007. pp. 552–561. ACM (2007)

[44] Sharif, H., Abubakar, M., Gehani, A., Zaffar, F.: Trimmer: Application Specialization for Code Debloating. 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE) (2018)

[45] Smowton, C.S.: I/O Optimisation and elimination via partial evaluation. Tech. rep., UC, CL (Dec 2014)

[46] Sun, C., Li, Y., Zhang, Q., Gu, T., Su, Z.: Perses: syntax-guided program reduction. In: ICSE 2018. pp. 361–371 (2018)

[47] Tip, F., Laffra, C., Sweeney, P.F., Streeter, D.: Practical experience with an application extractor for java. SIGPLAN Not. **34**(10), 292–305 (oct 1999)

[48] Turcotte, A., Arteca, E., Mishra, A., Alimadadi, S., Tip, F.: Stubbifier: Debloating dynamic server-side javascript applications. CoRR **abs/2110.14162** (2021)

[49] Vázquez, H.C., Bergel, A., Vidal, S.A., Pace, J.A.D., Marcos, C.A.: Slimming javascript applications: An approach for removing unused functions from javascript libraries. Inf. Softw. Technol. **107**, 18–29 (2019)

[50] Wu, J., Wu, R., Antonioli, D., Payer, M., Tippenhauer, N.O., Xu, D., Tian, D.J., Bianchi, A.: Lightblue: Automatic profile-aware debloating of bluetooth stacks. USENIX Security Symposium **30th** (2021)

[51] Xin, Q., Kim, M., Zhang, Q., Orso, A.: Program debloating via stochastic optimization. p. 65–68. ICSE-NIER '20 (2020)

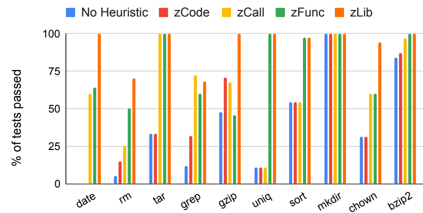
[52] Xin, Q., Kim, M., Zhang, Q., Orso, A.: Subdomain-based generality-aware debloating. IEEE/ACM ASE **35th** (2020)

[53] Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: FSE/SDP. pp. 421–426 (2010)

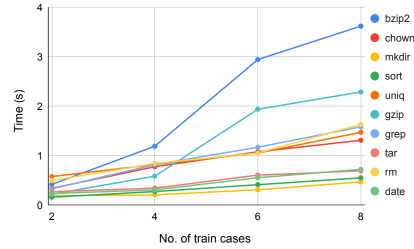
Appendix

Applications	#Train cases	#Correctness cases
bzip2	45	198
chown	20	35
mkdir	10	69
sort	55	77
uniq	55	55
grep	20	25
gzip	36	132
tar	33	99
date	50	50
rm	16	20
gm	125	75
gv	120	75
vlc	125	75
putty	7	10
nginx	9	12
(Total)	726	1007

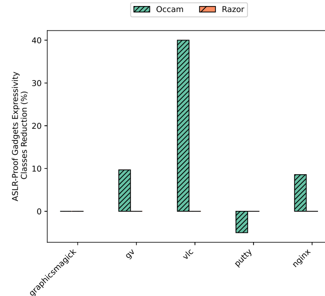
Table 3. The number of train and correctness testcases we used during our experiment for each of the input programs.



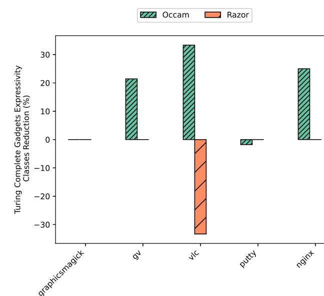
(a)



(b)



(c) ASLR-proof gadgets classes



(d) Turing-complete gadget classes

Fig. 10. For coreutils, (a) average fractions of test passed for different heuristics in Razor and (b) relationship between the time taken to train Razor and the number of train cases. For non-coreutils, (c) ASLR-proof ROP gadget expressivity and (d) Turing-complete ROP gadget expressivity reduction.