ProdeBench: A Comprehensive Program Debloating Benchmark

Anonymous Author(s)

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 the diverse nature of the approaches employed. We present ProdeBench[1], an extensible and sustainable benchmarking platform that enables comparison of different research techniques. 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. We perform a holistic comparison of the techniques and explore the various hidden and explicit tradeoffs in using them. Our evaluation shows that all the binaries produced by Occam and Piece-Wise were correct, while Chisel significantly outperformed others in binary size and Gadget class reductions. We also composed 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).

1 INTRODUCTION

With the growing success of the software industry, and the availability of several competing platforms to support, modern software has become bloated [21]. Since software is designed to be one-size-fits-all, it may contain several functionalities that the end-user does not need. This extra functionality can not only cause performance issues but also become a security risk [12, 32, 43]. 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 [6–10, 14, 17, 18, 20, 23, 25, 27, 29–31, 33–36, 38, 39, 41, 42]. Given, a *target program* and *its deployment context*, an automatic program debloating (a.k.a *software specialization*) tool debloats the program itself [7–10, 14, 17, 18, 20, 23, 27, 29–31, 33, 34, 36, 38, 39, 42] or its execution environment (e.g., Kernel [6, 25, 35], Firmware [41], 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 an unified benchmark platform, path to setting up diverse tools for analysis is unclear and seems expensive [33].

This paper takes the first step towards creating an extensible and sustainable benchmark framework named ProdeBench, for automated software debloating tools. We designed ProdeBench to evaluate program debloaters targeting software written in C/C++, since most of the efforts in software debloating [7–9, 14, 20, 27, 30, 31, 33, 36, 38, 39, 42] are dedicated to C/C++ domain. This design choice also enable the broadest impact, as applications written in C/C++ are more widely used, run in resource constrained environments [22] and offer more attack surface than others [11, 13, 37].

The landscape of existing C/C++ program debloating tools is diverse in nature. Based on the target types, existing C/C++ program debloating tools can be broadly categorized as *i*) application-level (e.g., [20, 27, 31]), *ii*) library-level (e.g., [30, 33]) and *iii*) environment-level (e.g., [6, 25]) program debloaters. Application-level program debloating tools can be categorized as follows: source-level [20], intermediate representation (IR)-level [27] and binary-level [31] specialization. While our ProdeBench framework is more general, the current version integrates a set of software debloating research tools that represent four different classes: Chisel [20] (source code), Occam [27] (compiler intermediate representation), Razor [31] (executable binary) and Piece-wise [33] (external library).

Efforts to create an unified framework to evaluate a diverse set of software debloaters are significantly hindered by two main challenges – i.e., handling the diversity of *design* and *execution environment*. We accommodated design level diversity while choosing the target program suite, selecting the deployment context and test-cases to ensure fair treatment during performance comparison. We used container-based software isolation with Docker to handle the diversity of execution environments. ProdeBench is designed to provide an *easy-to-use* command-line interface to run the different debloating tools that are integrated into it. ProdeBench can be downloaded and run with as few as 3 commands. For wide applicability and adoption, the ease of adding new tools or enhancing the target application suite is a core requirement. So among other things, ProdeBench is customizable and extensible by design.

To evaluate the ProdeBench framework, we conducted a pilot study with the set of above-mentioned debloaters. The exercise allowed us to pose and answer research questions about the selected suite of tools. 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. Our gadget analysis focused on ASLR-proof attack bootstrapping and Turing-complete categories of micro-gadget classes [5]. 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. Three of the four tools ran quickly enough that they could be integrated

[1] Prode means *brave* in Italian. Although, some of the decorated computer scientists might assume that ProdeBench is the short form of program debloating benchmark.

Debloating Class	Tools	Target		Type			Input			Analysis		Automated?	Opensourced?
		Application	Library	Source	LLVM IR	Binary	Configuration	Testcases	Annotation	Static	Dynamic		
Source	Chisel [20]	✓	-	✓	-	-	-	✓	-	✓	-	●	✓
	C-Reduce [36]	✓	-	✓	-	-	-	✓	-	✓	-	●	✓
	Perses [39]	✓	-	✓	-	-	-	✓	-	✓	-	●	✓
	DomGad [42]	✓	-	✓	-	-	✓	-	-	✓	-	●	✓
LLVM IR	Occam [27]	✓	✓*	-	✓	✓	✓	-	-	✓	-	●	✓
	Trimmer [8]	✓	-	-	✓	-	✓	-	-	✓	-	●	-
	LLPE [38]	✓	-	-	✓	-	✓	-	✓	✓	-	◐	✓
	LMCAS [9]	✓	-	-	✓	-	✓	-	-	✓	-	●	-
Binary	Razor [31]	✓	✓*	-	-	✓	-	✓	-	-	✓	●	✓
	Ancile [14]	✓	✓	-	-	✓	✓	✓	-	-	✓	●	-
Library	Piece-Wise [33]	-	✓	✓	-	-	-	-	-	✓	-	●	✓
	BlankIt [30]	-	✓	-	✓	-	-	-	-	-	✓	●	-
	Nibbler [7]	-	✓	-	-	✓	-	-	-	✓	-	●	-

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

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 compositions of multiple tools to debloat a single binary. Our experimental evaluation indicates showed that compositions can achieve better reductions of gadget classes than the best single tool.

Our contributions can be summarized as follows:

- We develop a new *easy-to-extend* benchmarking framework named ProdeBench to evaluate software debloating techniques. We created a set of 60 different variants of 10 coreutils application for robust analysis. In the current version, 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 ProdeBench.
- 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.
- We also 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 gadget expressively cases, several compositions (e.g., Chisel-Occam, Chisel-Occam-Razor) significantly outperformed the best single tool (Chisel).

2 DEBLOATING METHODS

ProdeBench 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 representatives that are included in our benchmark.

2.1 Application-level program debloating.

Application-level program debloating tools can be categorized as follows: source-level [20], intermediate representation (IR)-level [27] and binary-level [31] debloaters.

Source-level program debloating. Most of the source-level program debloating methods (e.g., Chisel [20], C-Reduce [36], and Perses [39]) 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 [42]. To address this issue, Xin *et al.* proposed DomGad [42] for *sub-domain aware* program specialization. *However, as acknowledged by Xin et al. [42],* Chisel *[20] represents the state-of-the-art, in terms of debloating 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 [27] 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 [8] , LLPE [38], LMCAS [9] 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 for executable binaries rely on execution tracing, triggered by a carefully chosen testcases (e.g., Razor [31]) or fuzzing (e.g., Ancile [14]). Razor has three modules (*Tracer*, *PathFinder* and *Generator*) that work collectively to generate a debloated binary supporting all required features. To achieve this, 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 order to support more inputs of the same functionalities, *PathFinder* is used to expand the CFG based on some control-flow heuristics. Finally, with the expanded CFG, the *Generator* module rewrites the original binary to produce a debloated binary. In addition to debloating context (i.e., intended functionalities), Ancile [14] requires a set of testcases to seed the fuzzer. *Because of being opensource, 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 [7], *ii*) load-time [33] and *iii*) runtime-debloating [30]. Given a set of applications, static debloating tools (e.g., Nibbler [7]) debloats dynamically linked libraries statically, which replaces the original set of libraries permanently. Load-time debloaters redact (e.g., Piece-wise[33]) functions while loading the target library into the memory. Runtime debloaters load (e.g., BlankIt [30]) certain functions only if they are required at runtime. *Since,* Piece-wise *is the only open-sourced tool in this domain, we included it as the representative for library-level program debloating tools.*

3 CHALLENGES FOR BENCHMARKING

We discuss various challenges in creating a program debloating benchmark including dealing with tool diversity while preserving fair comparison and bugs in prototype implementations.

3.1 Dealing with Program Diversity

Design-level diversity of existing debloating tools makes direct comparisons harder. Following we discuss some of the important challenges we address to handle them.

Dealing with partial evaluation. Partial evaluation-based tool, Occam uses a set of *static* and *dynamic* arguments for producing debloated programs. In the debloated version static arguments become constants. Let's consider the following scenario: If someone executes `sort` as `sort -g`, it performs *general numeric sort*, but `sort -h` performs human numeric sort. However, if someone executes `sort -hg`, it shows, "options 'gh' are incompatible". This means, if *g* and *h* both are used jointly as static arguments in Occam, the specialized binary will behave such that both options were used simultaneously. More generally, let's assume, a program *P* supports two exclusive functionality with two arguments arg_x and $arg_{x'}$ respectively, *we call* arg_x *and* $arg_{x'}$ *as conflicting arguments.* Then if someone uses arg_x and $arg_{x'}$ simultaneously as static arguments to specialize *P*, the specialized binary will behave like arg_x and $arg_{x'}$ are hardcoded into it and *will not perform either functionality.* To avoid this issue, it is required to ensure the compatibility of static arguments. However, Chisel, Razor and Piece-wise do not suffer from this problem. Thus, a meaningful performance comparison of Occam with others becomes challenging. **Resolution:** To overcome this challenge, we specialized programs with either one argument or a set of compatible arguments for all the specialization tools in our benchmark.

Dealing with load-time debloaters. Quach *et al.* [33] used ROPgadget [3] tool to measure the ROP gadget reduction after library specialization with Piece-wise. Piece-wise loader specializes libraries at load-time, so the specialized version only exists in the memory while the static version of the library retains to its original form. However, ROPgadget performs static analysis on the static binaries, so ROPgadget cannot be directly used to analyze *in-memory* specialized libraries produced with Piece-wise. **Resolution:** To solve this issue, we invented a new method to enable ROP gadget counting for Piece-wise at runtime with ROPgadget [3], which can be of an independent interest.

3.2 Ensuring Comparison Fairness

The goal of this study is to illuminate strengths and weaknesses of different program specialization paradigms. It required careful consideration to choose the best performing configuration of the tools and to design the testcases to ensure the fairness in comparison. In the following we discuss some of the issues we address to facilitate comparison fairness.

Choosing best performing configuration. Razor can debloat one program in five different ways using *no heuristic*, *zCode*, *zCall*, *zFunc*, and *zLib* heuristics. Our analysis has revealed that no one of the heuristics produces optimal results for all of the testing metrics. For instance, employing *zLib* almost always yields a smaller decrease in the number of ROP gadgets than *zFunc*. However, *zLib*'s correctness results are much better than others, where *no heuristic*, *zCode*, *zCall* give a lower performance than the rest (Figure 16). This means that there is an additional overhead of comparing Razor with other tools, where an optimal heuristic has to be separately decided for each argument after careful analysis. **Resolution:** To reslove this issue, we collected binaries debloated with each of the heuristics and only consider the variant with better correctness results. Similarly, after performing a sanity check on the performance of different configurations of Occam, we observed that while all the configuration produces correct binaries, *onlyonce* provided the best debloating output in terms of size decrease. So we used this configuration for performance comparison with other tools.

Improving the quality of testcases. Chisel and Razor's performance and the quality of the specialized binaries are largely dependent on the quality of testcases. We observed that the testcases presented inside the docker container of ChiselBench[2] were inadequate and resulted in defected binaries for several applications. For example, we noticed that the testcases for specializing for `uniq` were built with small files, so the specialized binary would pass testcases built around small files, but all the testcases involving large files failed. However, Razor did not face such issues since the testcases Razor used for specialization already considered large files. However, building testcases with large files for Chisel would

significantly increase its run time, which would be disadvantageous to CHISEL (i.e., it might not terminate within the maximum allocated time interval). **Resolution:** We used a mixture of both very large and small files for train cases to optimize the correctness of the produced binaries and the training time.

3.3 Prototype Implementation Issues

Implementation bugs in the tools made our performance measurement challenging. Since CHISEL operates on the source code, it produces debloated `C files`. In our testing, we observed that some of them contained syntax errors. Among many, we identified and patched CHISEL to fix two of the most prevalent cases of such errors.

One type of error was that the debloated files contained non-ASCII characters at arbitrary locations. We found that non-ASCII characters in its comments affected the debloating process, sometimes causing syntax errors by retaining non-ASCII characters without their comment containers(`//` or `/* */`). Another type of error was the accidental removal of curly braces from the code. We discovered that CHISEL's Dead Code Elimination functionality occasionally removes useful curly braces that are part of the syntax from the code. **Resolution:** We fixed both of the issues and opened pull requests in the CHISEL's Github repository master branch. Both of the solutions were approved by the authors and a pull request, which were merged into the CHISEL's Github repository. This not only enables a fairer comparison but also made CHISEL more robust.

We also figured during our analysis that RAZOR fails to make the original code read-only in the debloated binary for core-utils build for GNU/Linux 3.2.0 (these were compiled using clang version 7.0.1-8+deb10u2). However, this was fixed in the latest update made to RAZOR's repository by its authors. Additionally, some of debloated binaries in OCCAM, resulted in segmentation faults that we reported to the developers. All these implementation issues required extensive debugging and diagnosis efforts, in order to either fix them or minimize their impact on our evaluation.

4 PRODEBENCH BENCHMARK

We describe the goals, the design, and individual components of the PRODEBENCH framework.

4.1 Benchmark Goals

G1. Fairness. Providing a fair platform for comparison is an intrinsic requirement for any benchmark frameworks. As explained in Section 3.2, PRODEBENCH's design concurs with the requirement. Additionally, in the interest of fairness for future tools, users can add their own metrics, target applications, when they add a new framework to PRODEBENCH.

G2. Reproducibility. Ensuring the reproducibility of experimental results is also an important requirement. The target application, debloating contexts and testcases are well defined in PRODEBENCH, which would always lead to deterministic output. Thus, outputs produced from the framework are reproducible.

G3. Extensibility. Extensibility is essential for the continuous integration of new tools into the benchmark. PRODEBENCH aims to offer easy integration of new program specialization tools into it. Another design goal is to facilitate the extensibility of the input programs without affecting the existing setup.

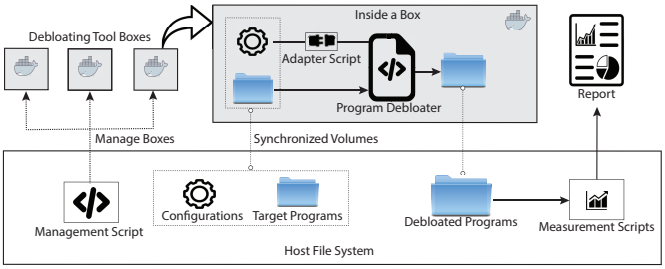

Figure 1: PRODEBENCH framework overview.

G4. Usability. Usability is also an important attribute for wide adoption. PRODEBENCH framework can be downloaded and run with as few as 3 commands. Any specialization tool, input program or measurement tool can easily be excluded for experimentation. Extending the benchmark is made easy with reusable templates and configuration files.

G5. Isolation. This is a complementary goal to support diverse execution environments required by different tools. For example, OCCAM and CHISEL both depend on llvm, but OCCAM requires version `10.0.1` and CHISEL requires `7.0.1`. The use of separate *non-interacting* Docker containers for each of the debloaters in our benchmark inherently provides isolation.

G6. Sustainability. Sustainability is an important attribute of any scientific benchmarking framework. Software written today, if not maintained well, can easily be outdated and become fragile due to the ever-changing nature of its supporting environment (e.g., libraries, software dependencies, OS). While the sustainability of PRODEBENCH will largely depend on the sustainability of the program specialization tools, the use of Docker containerization furthers packaging and redistribution toward this goal.

4.2 Framework Overview

Figure 1 provides an overview of our PRODEBENCH framework. We adopted a container-based approach to build PRODEBENCH framework. We created separate containers for each program specialization tools. 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 PRODEBENCH has a corresponding configuration file describing various metadata (e.g., testcase location, build script location, etc.) about the program. Different program specialization tools uses different formats for input program metadata. Each container has a corresponding adapter script to convert PRODEBENCH's input program configuration files into its own format. All the orchestrator scripts, input program suites with their configurations, performance measurement scripts reside in the host file system. Individual containers can directly access input programs and their configurations from the host file system. After specialization, the output binaries are copied into the host file system for analysis.

4.3 Framework Components

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

Debloating Tools. As discussed in Section 2, the current version of ProdeBench contains the following four program specialization tools, Chisel [20], Occam [27], Razor [31] and Piece-wise [33]. Each of them 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 specialization, they use different means to accept those inputs (Table 1). ProdeBench uses a configuration file to collect inputs corresponding to an input program. We created scripts to parse ProdeBench's configuration file and generate inputs for individual tools. These scripts are called *adapter scripts*. Adapter script of a tool bridges it with ProdeBench. The directory containing input programs and corresponding configurations are shared across all the containers from the host. After debloating, the outputs binaries are copied back to the host for analysis.

Target Program Suite. To evaluate the tools, we chose 10 coreutils programs from ChiselBench. The primary reason to choose applications from ChiselBench is to sidestep the issues of preparing input programs for Chisel. From the Chisel authors we learned that CIL [28] was used to merge the C files for the input programs in ChiselBench in the earlier version of Chisel. However, using CIL to merge files is manual and error-prone. While using CIL from scratch, we observed that dependencies for each of the programs are required to handle individually, which require significant manual efforts. Even after that the success of merging all the files is not guaranteed (resulting syntax errors). The recent version of Chisel is integrated with the build system of input applications. Because of implementation bugs, we could not successfully use it either. These bugs causes syntax errors in the debloated version of the source code. Since, we could not setup Chisel to work on programs other than the ones given in the ChiselBench, we only included them in our input program suite. 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 60 variants that constitute the complete workload. Note that, while combining multiple arguments in a given debloating context, we avoided choosing values with exclusive functionalities. As discussed in Section 3, specialization with arguments supporting exclusive functionalities in Occam does not produce binary with meaningful functionalities.

Tool Orchestrator. In ProdeBench, all management scripts are called *orchestrators*. The commands in orchestrator scripts conform to the following command-line convention: `./pdbench tool_name [args]`. For example, the command for building Chisel container is `./pdbench chisel build`. We implemented three categories of commands in for all the tools, *i*) container lifecycle management commands (such as build, start, stop, and remove) for creating and managing containers, *ii*) specialization commands for building and specializing input programs, *iii*) measurement commands for measuring different matrices. To specialize a program with a certain tool in ProdeBench we start by building and starting the corresponding container. Then we perform the specialization of the input program inside containers. Finally, we collect the specialized program by copying them into the host file system and then along with measurement metrics. All these steps are unified in orchestrators and can be performed from the host.

Measurement Scripts. Our measurement scripts unified the overall data collection procedure for all of them. We measure the performance of program specialization techniques with the following five metrics: *i*) correctness of the debloated binaries *ii*) decrease in binary size and memory usage, *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 mostly correlated with the functionalities. Elimination of them are more likely to be influenced by the selection of functionalities than a tool. Next, we discuss the *in-memory* gadget counting method, we used to evaluate load-time program debloaters.

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].

4.4 Extending ProdeBench

ProdeBench can be extended in the following two directions, *i*) adding new program specialization tools and *ii*) adding new input programs. Next, we briefly discuss the processes to accomplish that.

Adding new tools. In order to add a new program specialization tool to ProdeBench, users need to create a Docker container following the ProdeBench guideline. After that they need create a corresponding *orchestrator* by using the guidelines and template codes provided within ProdeBench. The users also need to provide a specification mentioning the required inputs to run and a corresponding adapter script to parse those inputs from ProdeBench configuration files for input programs.

Adding new input programs. Each tool in ProdeBench comes with a specification of required inputs and how to provide them in a JSON configuration file. Each tool has its own field, within which users need to provide corresponding information. Figure 18 in Appendix shows an example configuration for Chisel. By creating a JSON configuration file and corresponding inputs, users can test their target program against the specialization tool of their choice. In the current version, users can collect results from up to four specialization tools included in ProdeBench by filling their corresponding sections in the configuration file. Each tool provides template files and validation scripts to make the integration more user-friendly.

5 EXPERIMENTAL SETUP

We discuss our experimental setup to evaluate the performance of program specialization tools in ProdeBench. We conducted two set of experiments to measure the performance of *i*) standalone

Table 2: Target application suite and their deployment contexts considered in ProdeBench. All the cells with single or a combination of flags in a row represents *the set of deployment contexts* corresponding to an application.

	Selected Arguments and Combinations											Total Variants (60)
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	-f	-fn	-fr	-n	-r	11
uniq	-c	-cd	-cdw N	-cu	-cuw N	-cw N	-d	-dw N	-u	-uw N	-w N	11
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

tools and *ii*) their composition. Finally, we discuss the metrics that we used to compare performance.

5.1 Standalone Mode

Testcase Preparation. With the target program to debloat, program specialization tools in our ProdeBench also take testcases (i.e., Chisel, Razor, Piece-wise) or a configuration file (i.e., Occam) as input. To produce binaries with Chisel, Razor and Piece-wise, we created a set of 325 testcases for all the target applications (Table 3 in Appendix). To produce variants with Occam, we used the corresponding selected set of arguments (Table 2) as static. We also created a total 710 number of testcases to check the correctness of the specialization tools. While preparing these testcases, we aimed to capture diverse behavior in order to maximize the coverage.

Setting up Chisel. requires merging all the codes for an input program into a *C file*. However, as explained in Section 4.3, merging codes is an error-prone process and requires non-trivial efforts. To avoid that, during our experiment for all the input programs we reused the merged *C files* from ChiselBench.

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.

Setting up Razor. There are three main components in Razor [31], namely *Tracer*, *PathFinder* and *Generator*. *Tracer* executes the binary with train cases, and then records the execution traces. These traces are used to create a partial CFG of the input program. *Pathfinder* uses four heuristics (i.e., *no heuristic*, *zCode*, *zCall*, *zFunc*, and *zLib*) to collect codes corresponding to the CFG. *Generator* then uses the results of the path finder to rebuild the binary for the program. Razor 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.

5.2 Composition Mode

Since various specialization tools in ProdeBench 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 ProdeBench's input program suite:

(1) Chisel to Occam
(2) Chisel to Occam to Razor
(3) Chisel to Razor
(4) 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 7.2. Our selection of variants, training cases and correctness testcases to analyze the results of the aforementioned compositions is identical to individual tools. For a given metric, we compare the performance of compositions with the best performing individual tools.

6 EVALUATION

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 60 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.

6.1 RQ1: Tool Correctness

Software debloaters may be used by developers as a productivity aid or end users that need to deploy applications in resource-constrained or high-risk environments. In either case, there is an expectation that a target application will continue exhibit correct behavior within the planned deployment context.

Our definition for whether a specific debloated binary is *correct* is an under-estimate. More specifically, we implement a test that runs the program with a particular input and checks if it produces an expected output. A different such test must be created for each of the 60 variants. This set of test scripts is included in the distribution of the benchmark.

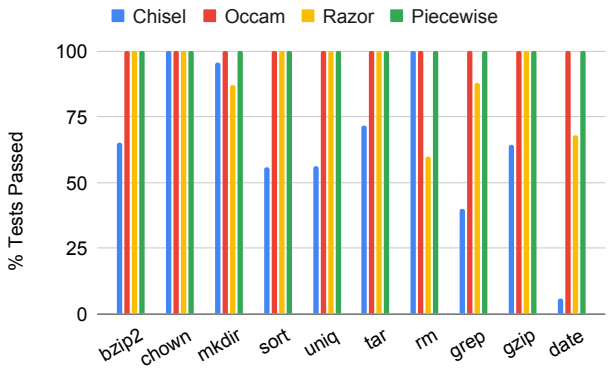

Figure 2: Each debloater was applied to all variants of a target. The average fraction of tests passed for each target is reported here.

Figure 2 reports the results of applying the correctness tests to all 60 variants. The debloating approaches that employ static analysis – that is, Occam and Piece-wise – passed 100% of the tests. In contrast, the debloaters that rely on dynamic analysis did not – Chisel only passed 65.5% of the tests, while Razor produced correct results for 90.3% of the cases.

We undertook the exercise of augmenting the training cases provided with each debloater for the target applications. Our experience indicated that more training yielded increased debloating correctness. In particular, 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 variation, 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 Figures 3 and 4, respectively.

Summary: Correctness

Chisel: 65.5%, Occam: 100%, Razor: 90.3%, Piece-Wise: 100%. Static analysis-based debloaters produce more correct binaries than dynamic analysis-based debloaters.

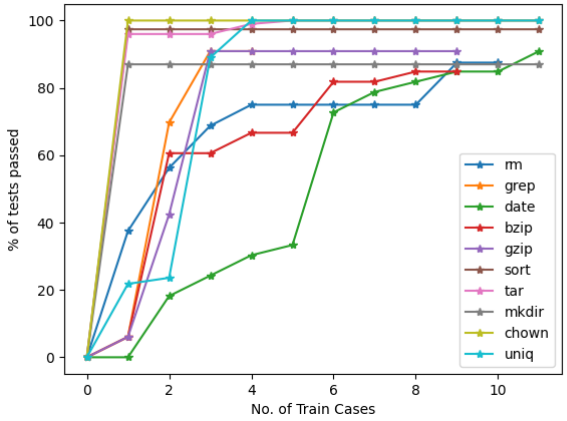

Figure 3: The correctness of Razor's debloating is a function of the training cases used.

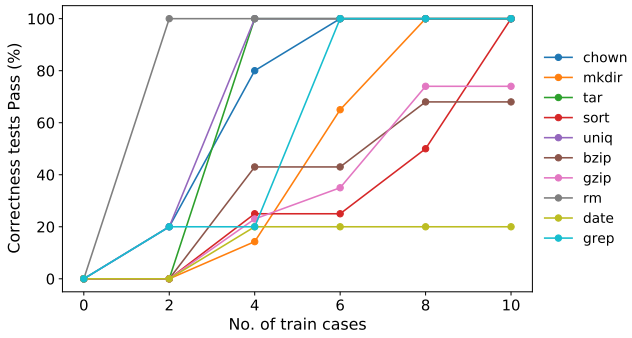

Figure 4: The correctness of Chisel's debloating is a function of the training cases provided to its oracle test script.

6.2 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. There are two levels in the memory hierarchy where this has a notable effect. The first is the memory used in the address space of the process created from the program binary. This resource is of particular interest for embedded applications, where lower memory usage can translate to lower manufacturing or operational costs. Valgrind's Massif [4] was used to measure execution of the original as well as debloated versions of each program. The resulting

reduction in memory usage is reported in Figure 5. At runtime, the debloated binaries produced by Chisel take less memory while those from Occam and Razor take more memory than the originals. This derives from the change in the size of the binary on disk, as described below. (A larger binary takes more memory when loaded in the process address space.)

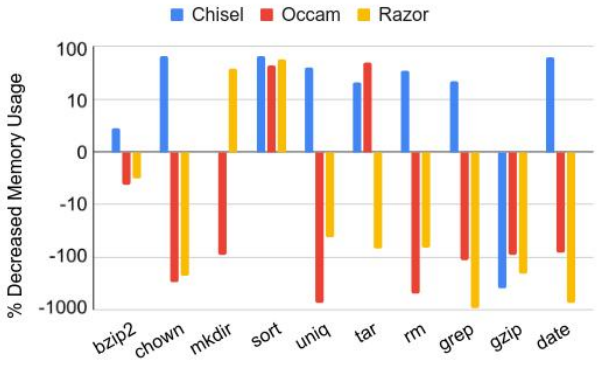

Figure 5: The memory used by the process created from the debloated binary. Each bar is the average for all variants derived from that application. Of particular note, using different debloaters on the same target produces significant variation in the memory footprint.

The second effect is on the binary size on disk. While storage costs continue to decrease, they matter when operating at scale. For example, debloating may be applied to a collection of applications, such as an entire operating system distribution in an embedded device, a cloud virtual machine, or a container runtime. Upon investigation, we found that debloaters employ diverse strategies with respect to how they transform the binary.

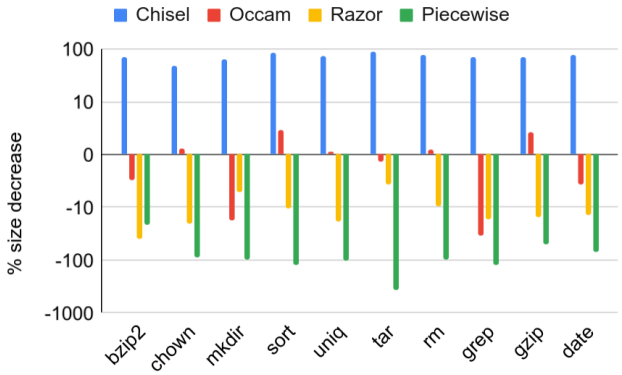

Figure 6: Effect on binary size on disk after applying different debloaters to each target application. Note that positive decreases indicate binary size reduction. Negative values mean that the binary size grew.

The approaches fall into two broad classes. The first typically reduces the code on disk, while the second significantly adds to it. In the first class, Chisel and Occam eliminate code at the source and compiler intermediate representation levels, respectively. This

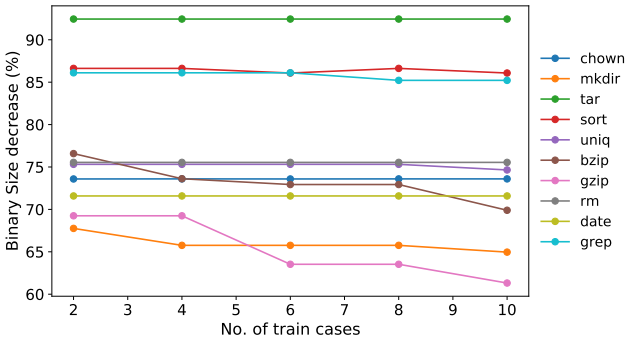

Figure 7: The reduction in binary size on disk by Chisel as a function of the number of training cases used.

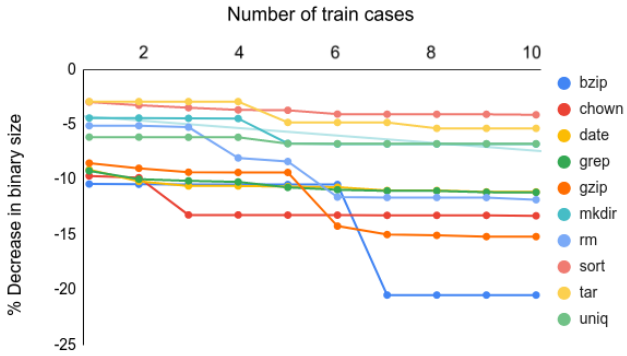

Figure 8: The reduction in binary size on disk by Razor as a function of the number of training cases used. Note that positive values indicate reductions, while negative ones are size increases.

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. Since Occam's partial evaluation can result in an increase in the number functions (when both unspecialized and specialized versions are retained), it is seen to occasionally increase code size.

In the case of the debloaters that use dynamic analysis, the extent of the binary size reduction depends on the number of training cases that are utilized. With more training, the resulting debloated binary becomes more robust. However, this arises through retention of more functionality, which reduces the extent to which the binary can be debloated. To quantify the effect, we report the binary size reduction as a function of the number of training cases. Figures 7 and 8 show this for Chisel and Razor, respectively. In the case of Chisel, as the number of training cases increase, the binary size decrease reduces in some cases while remaining stable in others. With Razor, the pattern is analogous but opposite. This is because Razor retains the original binary code and extends it. In some cases, the binary grows slowly with the increase in training cases while in others it increases in size significantly.

Summary: Size Reduction

In terms of reducing memory footprints, Chisel *outperformed others. In terms of size on disk,* Chisel *and* Occam *effect reductions. However, due to extending the original binary,* Razor *and* Piece-Wise *cause an increase in size.*

6.3 RQ3: Gadget Expressivity

Raw ROP gadget count and code size is not a reliable metric for estimating the vulnerability of a binary, as explained below. Homescu *et al.* [5] 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.

The simplest *practical* category involves 11 classes, but is not robust to the *address space layout randomization (ASLR)* functionality that is present in modern commodity operating systems. The category is also not Turing-complete but suffices for bootstrapping an attack. An ASLR-proof variant employs 35 classes. Homescu *et al.* also constructed a Turing-complete category with 17 classes.

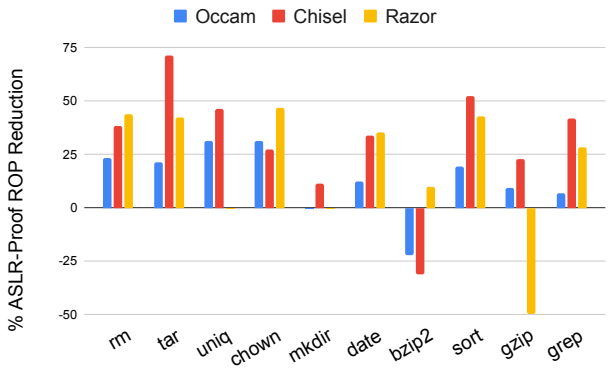

Figure 9: ASLR-proof ROP gadget expressivity reduction.

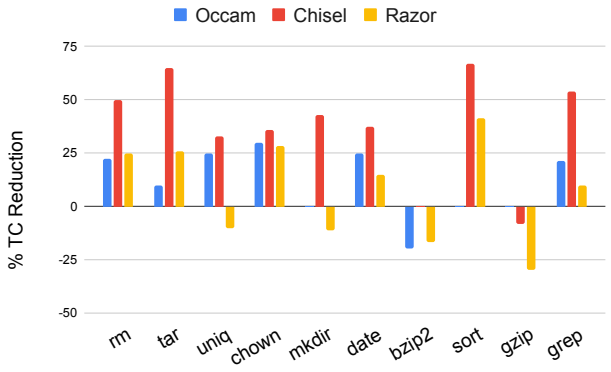

Figure 10: Turing-complete ROP gadget expressivity reduction.

Brown *et al.* [15, 16] observed that code size reduction may lead to increased gadget-based vulnerability. This is because even though many gadgets of some classes may be removed, it is possible that a few gadgets in new classes may be introduced. Consequently, 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 Figures 9 and 10, respectively. In both categories, Chisel yields the highest reduction (of 31.4% and 37.7% on average, respectively). For the Turing-complete category, Occam (11.4%) is more effective than Razor (7.8%). In contrast, Razor (19.9%) 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. More specifically, gadgets for 16 of 17 Turing-complete classes and 34 of 35 ASLR-proof classes were present.

Summary: Gadget Expressivity

Chisel *outperformed others (by around 2X) in both categories: ASLR-proof (31.4%), Turing complete expressivity (37.7%). For the second position,* Occam *and* Razor *were indistinguishable. However,* Piece-Wise *failed to remove any gadget classes.*

6.4 RQ4: Tool Usability

A significant factor that makes a debloating tool usable is the time it takes to run. The budget permitted depends on the workflow in which it is employed. If it is to be utilized as a compiler transformation, the latency of execution must be low. If it is to deployed in a continuous integration system, a longer run may be feasible. If it is only used in the final step of staging a major software release, a significantly longer run may be acceptable.

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 11. Chisel takes several orders of magnitude (with an average of 11,453 seconds) more time than the other debloaters. This limits the settings in which could be practically utilized. In contrast, the average time taken by Piece-wise, Occam, and Razor are 4.1, 4.9, and 5.2 seconds, respectively. This makes them usable in traditional optimization workflows.

Summary: Debloat Time

Chisel *takes several orders of magnitude (3.18 hours) more than others. The average time taken by* Piece-wise, Occam, *and* Razor *are 4.1, 4.9, and 5.2 seconds, respectively.*

6.5 RQ5: Debloater Composition

The debloaters were selected as representatives of tools that operate on source, compiler intermediate representation, and binary formats. This allows the debloaters to be composed into pipelines. In particular, a target program's source can be fed to Chisel, which emits debloated source. This can be compiled into LLVM bitcode and fed to Occam, which emits specialized bitcode. This in turn can be compiled into a binary that can then be fed to Razor, which emits a reduced binary.

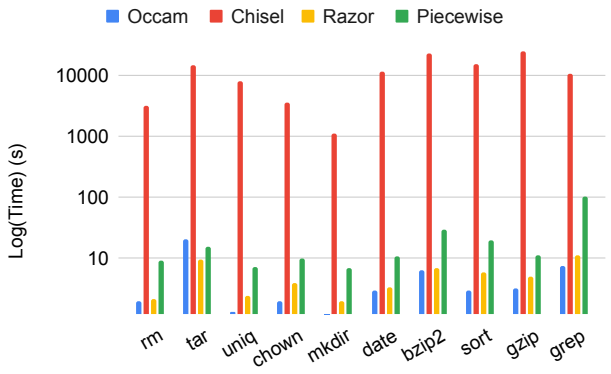

Figure 11: Average time to debloat all variants of a target application with each debloater. Note that the y-axis is logarithmic to accommodate the large differences in time taken.

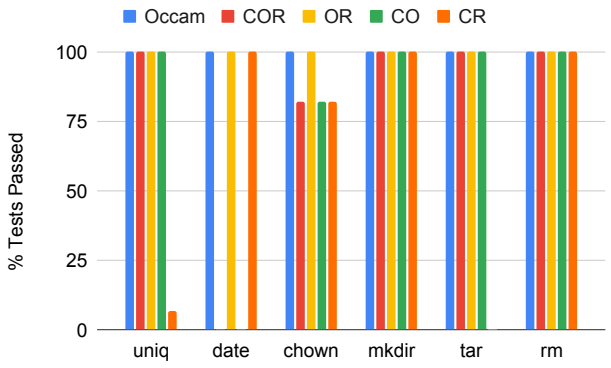

Figure 12: Average fraction of tests passed after applying a debloater composition to a target application.

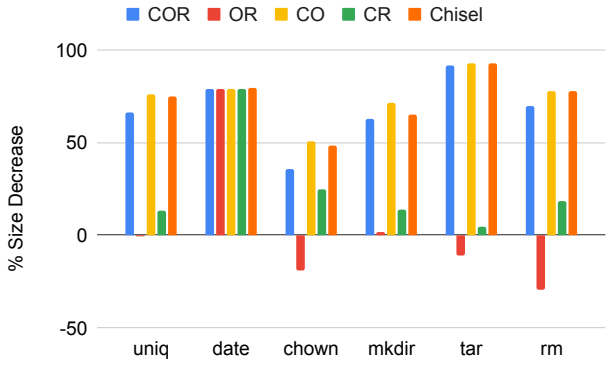

Figure 13: Average binary size reduction using debloater composition.

In Figures 13, 14, and 15, this 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 combinations 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 12 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 6.1) 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.

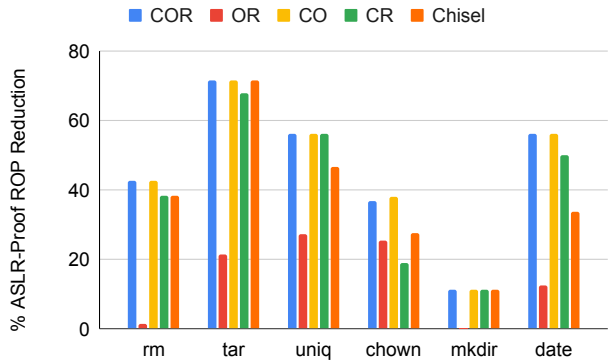

Figure 14: ASLR-proof ROP gadget expressivity reduction using debloater composition.

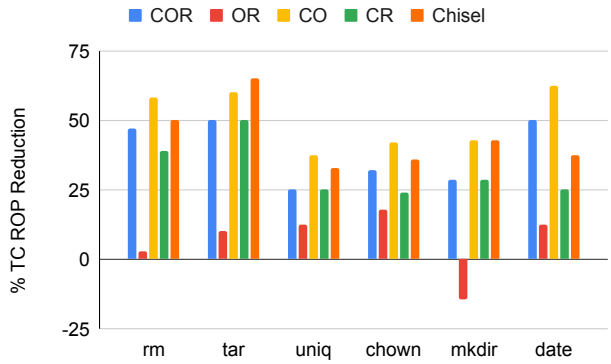

Figure 15: Turing-complete ROP gadget expressivity reduction using debloater composition.

The maximum average binary size reduction by any single tool is 73.6% (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 37.7%. 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 31.4%. Here, Chisel-Occam, Chisel-Razor, and Chisel-Occam-Razor combinations all outperformed with reductions of 45.9%, 40.4%, and 45.8%, respectively.

Summary: Composition of Debloaters

- **Correctness:** *Dictated by* Chisel. *Produces correct binaries for non-*Chisel *pipelines.*
- **Size:** Chisel *to* Occam *pipeline slightly outperformed the best 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*).*

7 DISCUSSION AND LIMITATIONS

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.

7.1 Impact of Design Choices

Delta debugging in Chisel. Since Chisel is a delta debugging-based [44] iterative algorithm, it takes orders of maginitude more time than its peers (Figure 11). Since Chisel is a delta-debugging-based iterative algorithm, it takes orders of maginitude more time than its peers. It is important to note that Chisel's debloating objective allows for an entirely different set of debloated binaries that would not be possible with a tool like Occam and while Razor provides a similar guarantee to Chisel in terms of the functionality of the debloated binary, our experiments show that the ROP gadget and binary size reduction are better in Chisel.

Our evaluation also 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 property test scripts that it uses to guide the debloating. In our experience, these scripts can be tricky to get right and Chisel can misbehave some times even when the scripts are seemingly correct. In particular, we found ensuring correctness for sort to be very challenging. Most of the variants of sort fail the majority of the correctness tests despite repeated attempts to fix it. The original script released by the authors also produces a low quality binary for sort. In short, ensuring binary correctness in Chisel can be a very tedious process, costing quite some time. Part of the reason why writing these scripts can be tedious is that, while you may choose to write a very rigorous test script, Chisel test scripts are a trade-off between correctness and time to debloat. In our experiments, on an average, over 96% of the debloating time is spent in running the property test script. So the time one saves in writing the scripts may translate to extremely long debloating times. Finding the sweet spot may become a time consuming endeavour.

Partial evaluation in Occam. Our experience, shows that the use of partial evaluation [24] significantly reduced the usability of Occam. It only allows non-conflicting arguments to be present in a debloated binary. So if two conflicting arguments are required, one needs to create two variants. However, it is worth noting that configuration-based program debloating enabled by partial evaluation in Occam seems more natural from a usability point of view. This is because it does not require careful and tedious use of test-cases, where the quality of the tests impacts the overall usability of the debloated binary.

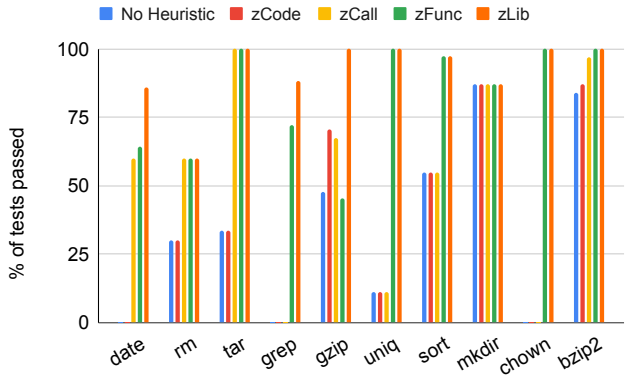

Figure 16: Average fractions of test passed for different heuristics in Razor**.**

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 benefits of using different heuristics cannot be fully assessed because one needs to choose the heuristic that produces correct binary to retain reasonable functionality. For instance, debloating using *no heuristic*, *zCode*, and *zCall* pass no tests for the `-F` argument of `grep` but *zFunc* and *zLib* pass 90% of the correctness tests. *zFunc* and *zLib* would include more of the code from the original binary and hence would have a higher number of ROP gadgets despite passing more testcases. Figure 16 illustrates the overall relationship between heuristic levels and the percentage of test cases passed in each application. Moreover, Razor's training time is dependent on the number of train cases provided to it. Figure 17 shows how the number of train cases impact the time taken to train Razor for each of the 10 target applications. While the overhead is relatively small for some target applications, other applications such as `bzip2` and `gzip` suffer from a much higher overhead. Hence, in addition to the number of train cases, the overhead is also largely dependent on the type of application being debloated.

Additionally, in some situations it may be possible that the effect of train cases is completely overridden by the heuristic used. For example, the binaries obtained without using any heuristic and using the `zCode`, `zCall` and `zLib` heuristics for the `-f` flag of `gzip` consistently failed all 33 test cases. On the contrary, the binary obtained using these same train cases and the `zFunc` heuristic passed all the 33 correctness tests. This is because `zFunc` allows including non-executed external functions, minimizing code debloat and preserving most of the functionality of the original binary. In such scenarios, the role of train cases becomes trivial as a similar success rate may be achieved using the same heuristic but with a much smaller number of train cases.

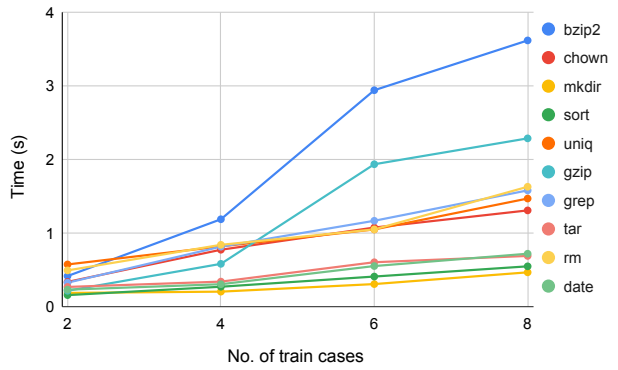

Figure 17: Relationship between the time taken to train RAZOR and the number of train cases.

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 code reduction in load-time. This significantly increases the size of the binary. For most of the applications we tested, `.dep` section constituted about half of their binary size. We observed the largest increase in `tar`, which is about five times higher than the original size. When using PIECE-WISE for debloating a program, this cost of increased binary size needs to be taken into account. For some applications the large increase in size can outweigh the benefit. Intuitively, loadtime program reduction also increases the loading time of a binary.

7.2 CHISEL to PIECE-WISE Pipeline

After debloating with CHISEL, out of 10 target programs, we were successful to compile 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. Our analysis on the binaries debloated with CHISEL to PIECE-WISE pipeline shows the correctness and ROP decrease of the debloated was dictated by CHISEL, while the size in disk was dictated PIECE-WISE. The overall results are summarized in Table 8 in Appendix.

7.3 Limitations of this Study

In the current version, PRODEBENCH chose a single tool per category and provides an in-depth analysis. However, as explained in Section 4.4, tool coverage can easily be extended. To build the target program suite one needs to consider the following criteria. First, *programs* need to be mature and well-maintained to ensure that they will be available in the future for continued use by the benchmark. Second, they need to general enough that they can be utilized with new debloating approaches and tools. Third, in addition to the programs, a set of runtime *contexts* need to be defined. These need to represent realistic deployments. However, preserving all the criteria at the same time might be hindered by practical limitations of the tools are testing. As discussed in Section 4.3, the choice of selecting target applications was hindered by a practical limitation of CHISEL's prototype. Also, the choice of functionalities in the debloated variants of the original target applications were influenced by OCCAM. We created extensive number of testcases to maximize the coverage, however it is hard to guarantee. Finally, it is worth noting that, as part of the development of PRODEBENCH 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.

8 RELATED WORK

C/C++ program specialization. There are three broad classes of program specialization, i.e., source-level (e.g., CHISEL [20], C-REDUCE [36], and PERSES [39] and DOMGAD [42]), IR-level (e.g., TRIMMER [8] , LLPE [38], LMCAS [9], OCCAM [27]) and binary-level (e.g., RAZOR [31]). Performance comparison 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 [33], BlankIt [30], Nibbler [7]) also followed a similar trend. RAZOR [31] and LMCS [9] are two exceptions. RAZOR compared its performance with CHISEL on runtime, binary correctness as well as code, ROP gadget and CVE reduction. LMCS [9] compared the runtime with OCCAM, CHISEL and RAZOR, while the performance on other metrics were compared with OCCAM. To the best of our knowledge, PRODEBENCH is the first benchmark to systematically scrutinize tools across all the categorizes to underscore the strength 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 [25] and SHARD [6] offers application-specific kernel-level debloating. CIMPLIFIER [35] uses dynamic analysis to detect logically distinct applications inside a container and automatically breaks it into smaller containers. LIGHTBLUE [41] leverages static analysis to perform application-guided firmware debloating. CDE [19] leverages execution tracing to identify the dependencies of an application for seamless porting.

Program specialization for other languages. Researchers have also explored program specialization for other languages. For example, Piranha [34] targets Objective-C. JSHRINK [17]; JSCLEANER [18], LACUNA [29], Muzeel [26] and [40] target JavaScript; JRed [23] and [10] target Java- and PHP-based applications, respectively.

9 CONCLUSION

We presented PRODEBENCH, 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. Our analysis of multi-tool composition at different stages opens up avenues for future explorations.

REFERENCES

[1] Busy box. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2021-06-25.

[2] Chisel benchmarks. https://github.com/aspire-project/chisel-bench.

[3] Ropgadget tool. https://github.com/JonathanSalwan/ROPgadget. Accessed: 2022-05-02.

[4] Valgrind Massif. https://valgrind.org/docs/manual/ms-manual.html.

[5] Microgadgets: Size does matter in Turing-Complete Return-Oriented programming. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, Bellevue, WA, August 2012. USENIX Association.

[6] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. Shard: Fine-grained kernel specialization with context-aware hardening. *USENIX Security Symposium*, 28th, 2019.

[7] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 70–83, 2019.

[8] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid Haroon Siddiqui, and Fareed M Zaffar. Trimmer: An automated system for configuration-based software debloating. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[9] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, multi-stage,compiler-assisted application specialization. *arXiv:2109.02775v1 [cs.SE]*, 6 Sep, 2021.

[10] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. *USENIX Security Symposium*, 28th, 2019.

[11] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[12] Suparna Bhattacharya, Karthick Rajamani, K Gopinath, and Manish Gupta. The interplay of software bloat, hardware energy proportionality and system bottlenecks. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, pages 1–5, 2011.

[13] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. Smashing the stack protector for fun and profit. In Lech Jan Janczewski and Miroslaw Kutylowski, editors, *ICT Systems Security and Privacy Protection - 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings*, volume 529 of *IFIP Advances in Information and Communication Technology*, pages 293–306. Springer, 2018.

[14] Priyam Biswas, Nathan Burow, and Mathias Payer. Code specialization through dynamic feature observation. In Anupam Joshi, Barbara Carminati, and Rakesh M. Verma, editors, *CODASPY '21: Eleventh ACM Conference on Data and Application Security and Privacy, Virtual Event, USA, April 26-28, 2021*, pages 257–268. ACM, 2021.

[15] Michael D. Brown and Santosh Pande. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[16] Michael D. Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. Not so fast: Understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[17] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: in-depth investigation into debloating modern java applications. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 135–146. ACM, 2020.

[18] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. Jscleaner: De-cluttering mobile webpages through javascript cleanup. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 763–773. ACM / IW3C2, 2020.

[19] Philip J. Guo and Dawson R. Engler. CDE: using system call interposition to automatically create portable software packages. In Jason Nieh and Carl A. Waldspurger, editors, *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*. USENIX Association, 2011.

[20] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 380–394, 2018.

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

[22] Farhana Javed, Muhamamd Khalil Afzal, Muhammad Sharif, and Byung-Seo Kim. Internet of things (iot) operating systems support, networking technologies, applications, and challenges: A comparative review. *IEEE Communications Surveys & Tutorials*, 20(3):2062–2100, 2018.

[23] Yufei Jiang, Dinghao Wu, and Peng Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016*, pages 12–21. IEEE Computer Society, 2016.

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

[25] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B. Bobba, David Lie, and Jesse Walker. Multik: A framework for orchestrating multiple specialized kernels. *CoRR*, abs/1903.06889, 2019.

[26] Tofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, Waleed Hashmi, Lakshmi Subramanian, and Yasir Zaki. Muzeel: A dynamic javascript analyzer for dead code elimination in today's web. *arXiv preprint arXiv:2106.08948*, 2021.

[27] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1504–1511, 2015.

[28] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[29] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. An extensible approach for taming the challenges of javascript dead code elimination. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 391–401. IEEE Computer Society, 2018.

[30] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 164–180, 2020.

[31] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. Razor: A framework for post-deployment software debloating. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[32] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments. In Taesoo Kim, Cliff Wang, and Dinghao Wu, editors, *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017*, pages 65–70. ACM, 2017.

[33] Anh Quach, Aravind Prakash, and Lok-Kwong Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 869–886, 2018.

[34] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. Piranha: reducing feature flag debt at uber. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, pages 221–230. ACM, 2020.

[35] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick D. McDaniel. Cimplifier: automatically debloating containers. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 476–486. ACM, 2017.

[36] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346, 2012.

[37] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 552–561. ACM, 2007.

[38] Christopher S.F. Smowton. I/O Optimisation and elimination via partial evaluation. Technical Report UCAM-CL-TR-865, University of Cambridge, Computer Laboratory, December 2014.

[39] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 361–371, 2018.

[40] Hernán Ceferino Vázquez, Alexandre Bergel, Santiago A. Vidal, Jorge Andrés Díaz Pace, and Claudia A. Marcos. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Inf. Softw. Technol.*, 107:18–29, 2019.

[41] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. Lightblue: Automatic profile-aware debloating of bluetooth stacks. *USENIX Security Symposium*, 30th, 2021.

[42] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. Subdomain-based generality-aware debloating. *IEEE/ACM*, 35th, 2020.

[43] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 421–426, 2010.

[44] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999.

APPENDIX

Figure 18: CHISEL's main field in the JSON configuration file

```
{
   "Chisel": { "Oracle":{"url":"", "file":""},
               "Program":{"tarball_url":"",
                          "git_repo_url":"",
                          "file":""},
               "Compilation":{
                   "compiler":"", "flags":""
                  },
               "BSI":{
                  "enable":"False",
                  "Init_cmds":"",
                  "Make_cmd":"",
                  "binary_output_loc":""
               },
               "Chisel_flags":"",
               "Dependencies":[]
               }
}
```

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

Applications	#Train cases	#Correctness cases
bzip2	42	198
chown	8	35
mkdir	10	69
sort	55	77
uniq	55	55
grep	15	25
gzip	28	132
tar	21	99
date	33	50
rm	16	20
(Total)	325	760

Table 4: Metrics for debloated binaries generated by Chisel

Program	Size Decrease (%) min/avg/max	ROP Gadget Decrease (%) min/avg/max	Memory Used Decrease (%) min/avg/max	ASLR ROP dec. (%) avg	Turing-complete ROP dec. (%) avg	Correctness Tests Passes (%) min/avg/max	Debloating Time (s) min/avg/max
bzip2-1.05	69.9/70.4/73.0	40.0/40.6/43.3	-3.4/2.8/5.8	-31.5	0	57.7/65.2/72.8	20249/22847/24386
chown-8.2	0.9/48.3/78.6	3.9/40.4/62.0	-9.3/64.9/78.9	27.4	36	100.0/100.0/100.0	1984/3538/6979
mkdir-5.2.1	65.0/65.3/65.6	67.2/69.7/73.6	0.1/0.1/0.1	11.1	42.8	91.7/95.6/100.0	782/1105/1991
sort-8.16	81.1/86.2/89.0	71.7/78.8/84.8	33.3/64.9/78.3	52.4	66.9	42.9/55.8/85.7	14280/15021/15626
uniq-8.16	74.4/74.7/74.9	62.7/65.5/67.0	21.6/39.4/67.3	46.6	32.9	0.0/56.4/100.0	4309/7864/10236
tar-1.14	92.4/92.7/93.0	89.3/91.3/93.2	-8.1/20.6/69.9	71.4	65	12.1/71.7/100.0	10862/14674/19304
rm-8.4	75.7/77.9/80.0	48.6/63.6/70.6	0.0/35.1/61.9	38.2	50	100.0/100.0/100.0	2048/3177/4131
grep-2.19	0.0/71.0/87.8	3.6/57.8/73.0	-22.2/21.6/85.8	42.0	53.8	0.0/40.0/100.0	4734/10632/17770
gzip-1.2.4	61.3/70.2/77.2	54.0/62.3/69.0	-1030.7/-415.4/56.9	22.7	-8.3	15.2/64.4/94.0	10986/24135/56778
date-8.21	71.6/79.5/82.0	67.6/75.5/84.4	55.0/64.5/72.4	33.8	37.5	0.0/6.0/100.0	8838/11534/16653

Table 5: Metrics for debloated binaries generated by Piecewise

Program	Avg. Size Decrease (%)	Avg. ROP Gadget Decrease	ASLR exp. dec.	Turing-complete exp. dec.	Avg. Correctness Tests Passes (%)	Overhead Added (s) min/avg/max
bzip2-1.05	-21.51	77.57	0%	0%	100%	2.4/2.4/2.4
chown-8.2	-92.81	72.96	0%	0%	100%	0.0/0.0/0.0
mkdir-5.2.1	-98.63	74.46	0%	0%	100%	0.0/0.0/0.0
sort-8.16	-129.15	71.98	0%	0%	100%	0.0/0.0/0.1
uniq-8.16	-105.68	74.41	0%	0%	100%	0.0/0.0/0.0
tar-1.14	-383.09	68.18	0%	0%	100%	6.7/6.8/6.8
rm-8.4	-101.83	73.98	0%	0%	100%	0.0/0.0/0.0
grep-2.19	-127.48	70.69	0%	0%	100%	9.7/9.8/9.8
gzip-1.2.4	-51.37	76.62	0%	0%	100%	0.0/0.0/0.1
date-8.21	-71.54	72.67	0%	0%	100%	0.0/0.0/0.1

Table 6: Metrics for debloated binaries generated by Occam

Program	Size Decrease (%) min/avg/max	ROP Gadget Decrease (%) min/avg/max	Memory Used Decrease (%) min/avg/max	ASLR ROP dec. (%) avg	Turing-complete ROP dec. (%) avg	Correctness Tests Passes (%) min/avg/max	Debloating Time (s) min/avg/max
bzip2-1.05	-3.0/-3.0/-3.0	-16.7/-16.7/-16.7	-91.9/-4.3/78.3	-22.2	-20	100.0/100.0/100.0	6.2/6.4/6.8
chown-8.2	1.3/1.3/1.3	1.8/1.8/1.8	-510.0/-300.2/-150.9	31.6	30.0	100.0/100.0/100.0	1.9/2.0/2.0
mkdir-5.2.1	-18.0/-18.0-18.0	4.8/4.8/4.8	-295.6/-88.7/10.9	0	0	100.0/100.0/100.0	1.2/1.2/1.3
sort-8.16	2.9/2.9/2.9	-3.3/-3.1/-3.0	-40.7/44.3/90.4	19.2	0	100.0/100.0/100.0	2.8/3.0/3.1
uniq-8.16	1.2/1.2/1.2	-2.2/-1.9/-1.6	-1437.3/-722.8/-94.5	31.25	25	100.0/100.0/100.0	1.3/1.3/1.3
tar-1.14	-1.3/-1.3/-1.3	4.3/4.4/4.4	51.2/51.2/51.2	21.4	10	100.0/100.0/100.0	19.9/20.1/20.3
rm-8.4	1.2/1.2/1.2	1.8/1.8/1.8	-581.7/-488.7/-395.9	23.5	22.2	100.0/100.0/100.0	2.0/2.0/2.0
grep-2.19	-35.3/-35.3/-35.3	-8.0/-8.0/-8.0	-112.1/-112.1/-112.1	6.6	21.4	100.0/100.0/100.0	7.4/7.4/7.4
gzip-1.2.4	2.7/2.7/2.7	7.0/7.0/7.0	-182.5/-89.7/-50.2	9.1	0	100.0/100.0/100.0	3.1/3.2/2.4
date-8.21	-3.7/-3.7/-3.7	10.1/10.7/11.3	-82.8/-82.8/-82.8	12.5	25	100.0/100.0/100.0	2.9/3.0/3.2

Table 7: Metrics for debloated binaries generated by Razor

Program	Size Decrease (%) min/avg/max	ROP Gadget Decrease (%) min/avg/max	Memory Used Decrease (%) min/avg/max	ASLR ROP dec. (%) avg	Turing-complete ROP dec. (%) avg	Correctness Tests Passes (%) min/avg/max	Debloating Time (s) min/avg/max
bzip2-1.05	-39.7/-39.7/-39.7	2.1, 2.1, 2.1	-6.5/-3.2/0.2	10	-16.7	100.0/100.0/100.0	6.0/6.8/8.2
chown-8.2	-21.9/-20.5/-18.8	41.8/44.1/47.0	-270.6/-224.9/-161.6	46.7	28.6	100.0/100.0/100.0	2.2/3.9/4.8
mkdir-5.2.1	-6.5/-5.1/-4.4	-11.2/-9.2/-7.5	36.0/39.0/42.0	0	-11.1	75.0/87.0/100.0	1.0/2.0/2.7
sort-8.16	-23.6/-10.8/-3.2	48.8/63.5/79.2	-11.6/57.6/99.2	42.8	41.6	85.7/100.0/100.0	2.0/5.8/7.9
uniq-8.16	-17.8/-18.84/-19.9	-6.4/-4.0/-1.7	-66.4/-41.8/-32.1	0	-10	100.0/100.0/100.0	2.0/2.4/3.3
tar-1.14	-4.6/-3.8/-3.1	67.1/69.4/71.8	-173.7/-68.0/51.1	42.3	25.9	100.0/100.0/100.0	2.1/9.6/13.3
rm-8.4	-11.4/-9.6/-8.1	41.4/50.6/54.9	-116.1/-64.2/-6.6	43.75	25.0	0.0/60.0/100.0	1.3/2.1/4.0
grep-2.19	-21.1/-16.9/-10.9	32.1/39.2/52.2	-1193.3/-927.1/-530.8	28.1	10	100.0/88.0/100.0	8.4/11.0/15.3
gzip-1.2.4	-21.7/-15.3/-11.6	40.5/45.6/54.2	-491.7/207.8/43.5	-50	-30	100.0/100.0/100.0	4.9/5.0/10.4
date-8.21	-28.0	29.5/59.9/78.9	-2349.6/-784.3/38.3	35.3	15	0.0/68.0/75.0	1.3/3.4/7.0

Table 8: Metrics for debloated binaries generated by PIECEWISE from debloated source files provided by CHISEL

Program	Size Decrease (%) min/avg/max	ROP Gadget Decrease (%) min/avg/max	Memory Used Decrease (%) avg	ASLR ROP dec. (%) avg	Turing-complete ROP dec. (%) avg	Correctness Tests Passes (%) min/avg/max	Debloating Time (s) min/avg/max
bzip2-1.05	69.9/70.4/73.0	40.0/40.6/43.3	-3.4/2.8/5.8	-31.5	0	57.7/65.2/72.8	20249/22847/24386
gzip-1.2.4	61.3/70.2/77.2	54.0/62.3/69.0	-1030.7/-415.4/56.9	22.7	-8.3	15.2/64.4/94.0	10986/24135/56778
grep-2.19	0.0/71.0/87.8	3.6/57.8/73.0	-22.2/21.6/85.8	42.0	53.8	0.0/40.0/100.0	4734/10632/17770
date-8.21	71.6/79.5/82.0	67.6/75.5/84.4	55.0/64.5/72.4	33.8	37.5	0.0/6.0/100.0	8838/11534/16653
tar-1.14	92.4/92.7/93.0	89.3/91.3/93.2	-8.1/20.6/69.9	71.4	65	12.1/71.7/100.0	10862/14674/19304

(a) Results from CHISEL

Program	Size Decrease (%) min/avg/max	ROP Gadget Decrease (%) avg	ASLR ROP dec. (%) avg	Turing-complete ROP dec. (%) avg	Correctness Tests Passes (%) min/avg/max	Overhead Added (s) min/avg/max
bzip2-1.05	-44.5/-42.2/-41.4	78.8/78.8/78.8	0	0	57.6/65.1/72.7	0.2/0.3/0.3
gzip-1.2.4	-23.75/-20.6/-13.1	78.8/78.8/78.9	0	0	15.1/65.1/97.0	-0.1/0.2/0.4
grep-2.19	-83.0/-76.8/-70.6	76.9/79.0/81.0	0	0	0.0/0.0/0.0	0.1/0.3/0.5
date-8.21	-45.3/-24.3/-9.9	76.0/76.3/76.4	0	0	0.0/0.0/0.0	0.0/0.2/0.4
tar-1.14	-54.3/-42.6/-30.8	78.9/80.0/81.0	0	0	12.1/56.1/100.0	0.1/0.2/0.3

(a) Results after debloating by PIECEWISE on the output by CHISEL