# SoK: Software Debloating Landscape and Future Directions

Mohannad Alhanahnah
mohannad@cs.wisc.edu
University of Wisconsin-Madison
Computer Sciences
USA

Yazan Boshmaf
yboshmaf@hbku.edu.qa
Hamad Bin Khalifa University
Qatar Computing Research Institute
Qatar

Ashish Gehani
ashish.gehani@sri.com
SRI
USA

## ABSTRACT

Software debloating seeks to mitigate security risks and improve performance by eliminating unnecessary code. In recent years, a plethora of debloating tools have been developed, creating a dense and varied landscape. Several studies have delved into the literature, focusing on comparative analysis of these tools. To build upon these efforts, this paper presents a comprehensive systematization of knowledge (SoK) of the software debloating landscape. We conceptualize the software debloating workflow, which serves as the basis for developing a multilevel taxonomy. This framework classifies debloating tools according to their input/output artifacts, debloating strategies, and evaluation criteria. Lastly, we apply the taxonomy to pinpoint open problems in the field, which, together with the SoK, provide a foundational reference for researchers aiming to improve software security and efficiency through debloating.

## CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; **Software security engineering**.

## KEYWORDS

Systematization of Knowledge, Software Debloating, Software Security, Taxonomy, SDLC, SBOM

**ACM Reference Format:**
Mohannad Alhanahnah, Yazan Boshmaf, and Ashish Gehani. 2024. SoK: Software Debloating Landscape and Future Directions. In *Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

## 1 INTRODUCTION

Modern software development is heavily dependent on third-party libraries to accelerate development and improve functionality [41]. However, this practice introduces significant complexity and increases the attack surface of applications due to the integration of various components, each with its own set of dependencies and vulnerabilities [17]. The increased complexity increases security risks and leads to code bloat, adversely affecting performance.

Software debloating [39, 52], the process of removing unnecessary code from applications, is a promising approach to address these issues. By eliminating extraneous features, debloating can significantly reduce the attack surface, enhance performance, and improve maintainability. This technique complements other security measures, such as Control-Flow Integrity (CFI) [35] and Address Space Layout Randomization (ASLR) [53], by minimizing the amount of code that needs protection. Software debloating has gained renewed momentum, in part due to cyber defense initiatives, such as the US Navy's Total Platform Cyber Protection (TPCP) program [2]. Subsequently, numerous debloating tools were introduced, leading to various studies [10, 12, 23] that examine the literature on software debloating and perform comparative analyses of the prototyped tools. While these studies are thorough, their primary objective is to empirically compare specific aspects, such as resulting binary size or gadget count, of particular types of debloating tools, such as those that target C/C++ programs or containers. The limited scope restricts the influence of these studies to a subset of debloating tools, rather than providing a systematic, comprehensive, and wide-ranging examination of the entire debloating domain, which encompasses a diverse array of tools and evaluation criteria. As such, there is a significant need to augment previous research with a holistic and systematic study of the complete software debloating landscape, thereby enabling more extensive and inclusive conclusions about open issues and challenges in this domain.

To bridge this gap, this paper systematizes the current knowledge on software debloating, providing a multilevel taxonomy that divides the current landscape into three main categories corresponding to the three main stages of the debloating workflow. We also highlight open problems in the field, calling for more practical, usable, and secure debloating solutions that can be integrated seamlessly into modern development workflows.

## 2 SOFTWARE DEBLOATING WORKFLOW

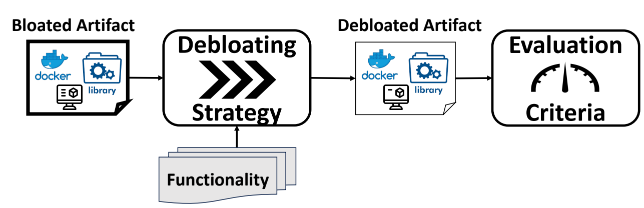


**Figure 1: Typical debloating workflow.**

Software bloat refers to unnecessary functionalities and their corresponding software dependencies and components [39, 52]. Figure 1 depicts the typical workflow used by debloating tools. To


*FEAST '24, October 14–18, 2024, Salt Lake City, UT, USA.*

ACM ISBN 979-8-4007-1233-3/24/10
https://doi.org/10.1145/XXXXXX.XXXXXX

**Table 1: Selected publications on software debloating landscape.**

| Tool | Venue | In/Out Artifacts | Removal Granularity | | | | Analysis | | | Functionality | | | Evaluation Criteria | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | File | Library | BB | Stmt | Static | Dynamic | ML-Assisted | Config | Test cases | Annotation | Performance | Security | Robustness | Usability | Integration | Sustainability |
| Hacksaw [26] | CCS'23 | S2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| C2C [22] | CCS'22 | S2P | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| Slimium [47] | CCS'20 | S2B | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| NA [19] | CCS'19 | B2B | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| CHISEL [24] | CCS'18 | S2S | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| PacJam [43] | ASIACCS '22 | S2S | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| LightBlue [60] | USENIX Sec'21 | S2B, B2B | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Temporal Specal. [21] | USENIX Sec'20 | S2P | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| RAZOR [46] | USENIX Sec'19 | B2B | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| Piece-Wise [48] | USENIX Sec'18 | S2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| IRQDebloat [25] | S&P'22 | B2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| LMCAS [8] | EuroS&P'22 | S2B | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| Saffire [40] | EuroS&P'20 | S2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| Mininode [30] | RAID'20 | S2S | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| CONFINE [20] | RAID'20 | C2P | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| CARVE [13] | FEAST'19 | S2S | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| BinRec [32] | FEAST'18 | B2B | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Nibbler [4] | ACSAC'19 | B2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| JShrink [14] | FSE'20 | B2B | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| JReduce [29] | FSE'19 | B2B | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| Cimplifier [49] | FSE'17 | C2C | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| Picup [58] | FSE'23 | S2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Minimon [37] | ICSE'24 | B2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Perses [56] | ICSE'18 | S2S | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| AutoDebloater [36] | ASE'23 | B2B | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| DomGad [61] | ASE'20 | S2S | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| BlankIt [45] | PLDI'20 | B2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| C-Reduce [51] | PLDI'12 | S2S | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Decker [44] | ASPLOS'23 | S2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| $\mu$Trimmer [66] | ASPLOS'22 | B2B | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Trimmer [6] | TSE'22 | S2B | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| XDebloat [57] | TSE'21 | B2B | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| NA [18] | TSE'21 | S2S | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| BLADE [9] | SecDev'23 | S2S | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| JDBL [54] | Trans. SE. Meth.'23 | S2B | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| OCCAM [42] | Commun. ACM'23 | S2B | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| Ancile [11] | CODASPY '21 | S2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| JSLIM [63] | EISA 2021 | S2S | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| PRAT [59] | TOSEM'21 | S2B | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| DEPCLEAN [55] | Empir SE'21 | D2D | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| DECAF [15] | ICSE-SEIP'20 | B2B | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| NA [31] | EuroSec'19 | S2B | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| DeepOCCAM [33] | MLforSystems'19 | S2B | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| BINTRIMMER [50] | LNSC'19 | B2B | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| RedDroid [27] | ISSRE'18 | B2B | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SPEAKER [34] | DIMVA'17 | C2P | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Jred [28] | COMPSAC'16 | B2B | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| NA [16] | ISLPED '01 | S2S | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ |

identify bloat and eliminate it, developers use existing tools that take a *bloated artifact*, such as an application, container, or firmware, often coupled with a *deployment context*, and then produce a *debloated artifact* utilizing a particular *debloating strategy* applied by the tool. After that, the quality of the output artifact is assessed using various evaluation criteria. As depicted in Figure 1, in addition to *bloated artifact*, *debloating strategy* may receive additional input (that is, in the form of annotation or instrumentation) to indicate the required functionality that should be preserved in the output artifact. The next section discusses the details of this debloating workflow in the context of the reviewed literature and our proposed taxonomy.

## 3 MULTI-LEVEL TAXONOMY

Our goal is to study and contrast existing software debloating tools and techniques. To achieve this, we first surveyed related research covering all papers published in top-tier security conferences, namely IEEE S&P, USENIX Security, ACM CCS, and NDSS from 2000 to March 2024. We also selected papers from top academic conferences and journals broadly related to software debloating. This process yielded 48 publications that are summarized in Table 1.

Figure 2 shows the multilevel taxonomy we designed to categorize the software debloating landscape.

In this taxonomy, the top level outlines the three main stages of the workflow. Lower levels categorize specific aspects of the debloating landscape, based on the publications listed in Table 1, under each stage of the workflow.

### 3.1 Input/Output Artifacts

Debloating tools require an input to generate an output. These inputs and outputs are referred to as artifacts and can come in various formats, such as source code, binaries, and containerized applications. The output resulting from debloating can also take any of these forms, or might even be a policy. Figure 3 shows the number of publications with proposed tools that use one or more of the following type mappings between input and output artifacts:

- **Source-to-Source (S2S).** In this workflow, the debloating operation is applied to the given source code, resulting in a minimized source code output. CHISEL [24] and Mininode [30] execute their debloating procedures for C/C++ and JavaScript programs, respectively.
- **Source-to-Binary (S2B).** The workflow starts with the source code and transforms it into an Intermediate Representation (IR). The debloating process then operates on the IR code. Ultimately, the debloated program is produced in binary format. For instance, LMCAS [8] debloats C/C++ programs by first

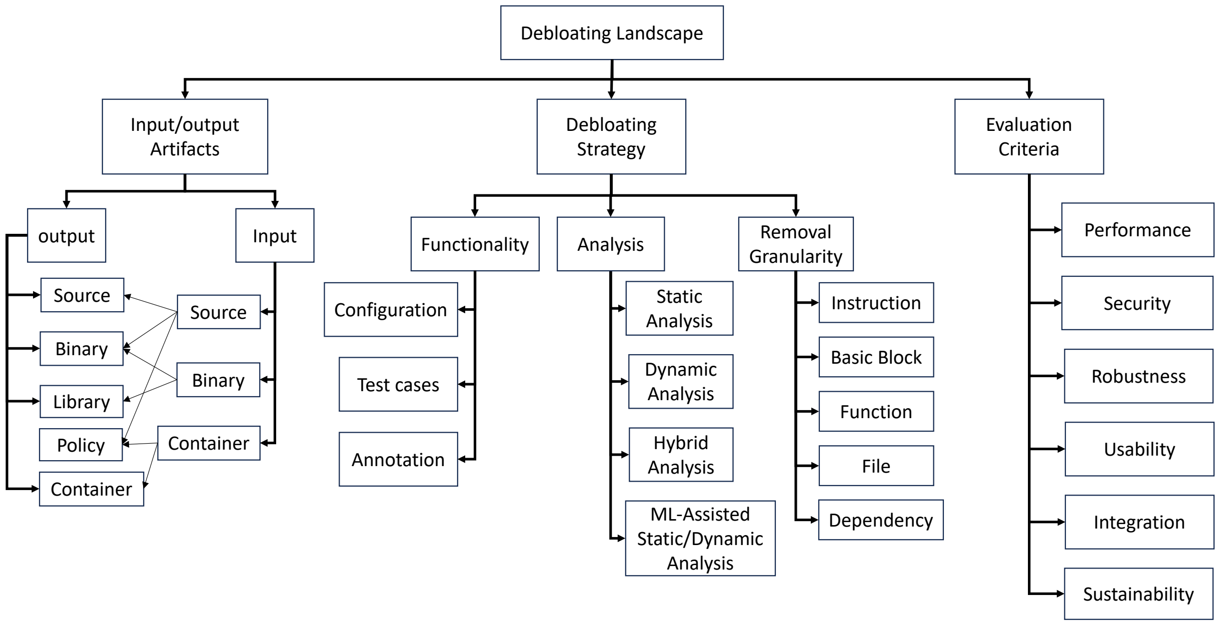


**Figure 2: Taxonomy of software debloating landscape.**

converting them into LLVM IR, resulting in executable output. Tools utilizing this debloating workflow have been applied to platforms such as firmware, as seen with PRAT [59]. Other tools in this category focus on trimming shared libraries, an approach exemplified by Piece-Wise[48]. Certain tools implementing this workflow extend beyond trimming by incorporating additional checks, such as Saffire [40].

- **Artifact-to-Policy (S2P or C2P).** This workflow generates a policy (i.e. *seccomp()*) that limits the program's behavior at run-time. As observed in the reviewed literature, the input artifact for this process can be either source code (S2P) or a containerized application (C2P), as exemplified in debloating tools such as temporal-specialization [21] and Confine [20]. Generally, these debloating methods do not involve actual trimming but focus on minimizing the use of unnecessary resources, such as syscalls.

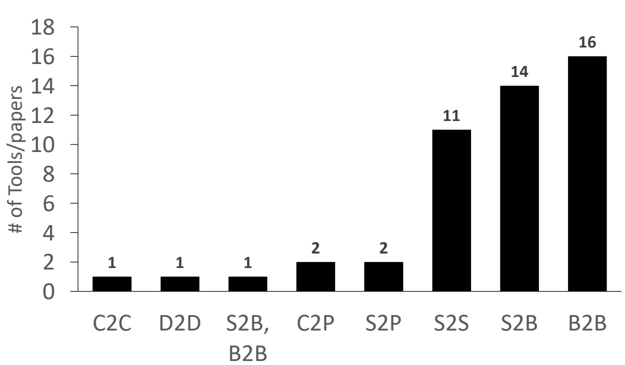


**Figure 3: I/O artifacts type mappings across tools.**

- **Binary-to-Binary (B2B).** The workflow begins with a binary file and results in a debloated program, also in binary format. Similar to S2B tools that apply additional checks, certain tools implementing this workflow extend beyond trimming, such as Razor [46], and incorporate extra checks, like those of binary control-flow trimming [19], to safeguard CFI. Consequently, the size of the debloated programs may increase in some instances. This debloating approach has been applied to various platforms, including Android (e.g., XDebloat [57], RedDroid [27]) and firmware (e.g., IRQDebloat [25], DECAF [15]). A different group of tools focus exclusively on debloating shared libraries, such as BlankIt [4] and Nibbler [4]. Likewise, tools such as $\mu$Trimmer [66] are designed to debloat shared libraries, but specifically within the context of firmware images.
- **Container-to-Container/s (C2C).** In this workflow, the debloating operation takes a container as input and produces a debloated version of the same container or divides it into multiple containers, each with a portion of the application from the original container. For instance, Cimplifier [49] can function in two modes: either by trimming the container or partitioning it into smaller segments. MMLB [64] builds on the trimming feature of Cimplifier to empirically investigate bloat in machine learning (ML) containers.
- **Dependency-to-Dependency (D2D).** This workflow accepts inputs consisting of dependency and build management files, like the Project Object Model (POM), where developers outline details about the project, its dependencies, and the build process. The output is a debloated version of the dependency management file(s). An example of this is DepClean [55], which specializes in debloating POM files in Java projects.

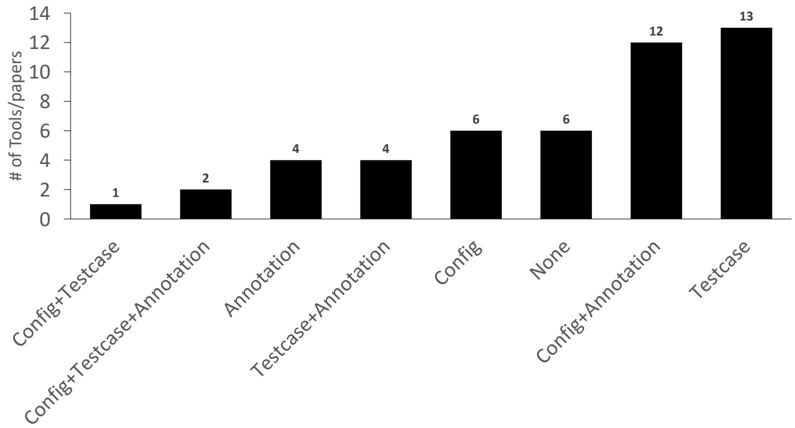


**Figure 4: Strategies to identify functionality across tools.**

Some tools adopt a more comprehensive approach to debloat various layers of the software stack, thereby combining multiple type mappings for input/output artifacts. For instance, LightBlue [60] debloats the Bluetooth stack, specifically focusing on debloating applications (S2B) and firmware (B2B).

## 3.2 Debloating Strategies

This stage of the workflow outlines the methods used by developers to determine unnecessary functionalities, pinpoint their associated dependencies, and remove them. As shown in Figure 2, this stage is divided into three main components, as follows:

*3.2.1 Functionality.* This component presents three strategies to identify unneeded functionalities at a high level.

- **Configuration.** In this strategy, the debloating workflow receives program configurations as input, which are to be preserved in the debloated output. These configurations may also specify particular points of interest, such as specific functions and libraries. For example, LMCAS [8] requires configurations via command-line arguments or a configuration file, mirroring the program's standard execution approach. Conversely, tools like OCCAM [42] and Trimmer [5, 6] use a template format to input the required configurations. Other tools, like temporal-specialization [21], anticipate the configuration in the form of a list of key functions from the input artifact.
- **Test cases.** This debloating strategy requires a collection of test cases to represent the program's usage profile post-debloating. Tools like Chisel [24] and Razor [46] use test cases supplied by the developers as input. Other tools, such as Ancile [11], employ fuzzing techniques to generate these test cases. Hacksaw [26] utilizes hardware probing to identify necessary device drivers to perform kernel debloating.
- **Annotation.** In this strategy, the input program is augmented with specific logic. This addition is either to gather particular information during dynamic analysis, such as profiling, or to initiate different actions. For instance, LMCAS [8] marks specific locations in the program to signal the completion of the profiling process. Conversely, Slimium [47] employs binary instrumentation to track functions that are called during runtime.

Six tools [4, 21, 27, 28, 30, 63] (under the none category in Figure 4) depend solely on static analysis techniques to pinpoint unneeded functionalities, eliminating the need for explicit expression of these functionalities. In particular, all these tools use only static analysis and identify unused code by performing a reachability analysis on call graphs [4, 27] or dependency graphs [30]. This indicates that the functionality can be further classified into two categories: unreachable content and feature removal, where the latter pertains to reachable but non-essential content.

*3.2.2 Analysis.* This component describes program analysis techniques that have been utilized by various software debloating tools.

- **Static Analysis.** This analysis focuses on building various types of graphs, such as call graphs, Control Flow Graphs (CFGs), and dependency graphs, to identify dependencies at multiple levels of granularity. C2C [22] generates a CFG and performs data flow analysis during its analysis. Additionally, an important aspect of static analysis is the optimization and elimination of unnecessary dependencies. For example, LMCAS [8], OCCAM [42], and Trimmer [6] implement LLVM passes to simplify and remove unneeded code.
- **Dynamic Analysis.** In this analysis technique, run-time data is collected to identify essential dependencies that must be preserved. This technique typically involves instrumenting the application before execution. Various tools have been used to aid in dynamic analysis. For example, LMCAS [8] and LightBlue [60] employ symbolic execution, whereas other tools such as Slimium [47] have developed their own dynamic analysis methods.

Machine Learning (ML) is often employed in conjunction with program analysis. An example of this is Chisel [24], which combines delta debugging with reinforcement learning. Various tools have utilized a blend of static and dynamic analyses, sometimes supplemented with machine learning (ML). For instance, Confine [20]

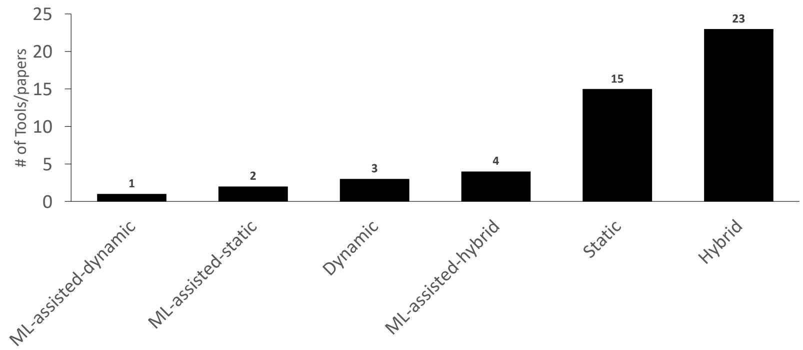


**Figure 5: Analysis techniques across tools.**

and Piece-Wise [48] employ hybrid analysis techniques for debloating containers and libraries. BlankIt [45], another hybrid analysis tool, focuses on debloating shared libraries and incorporates ML, specifically decision trees, to predict the functions required at a particular call site during execution. Figure 5 presents the number of debloating tools that fall under the different analysis categories.

*3.2.3 Removal Granularity.* Software debloating tools aim to eliminate unnecessary code and dependencies, but they do so at different levels of granularity. As shown in Figure 2, there are four distinct levels of removal granularity in the context of software debloating: (1) instruction or statement, (2) basic block, (3) function or library, and (4) file, including class or dependency management. Notably, some tools, such as Confine [20], temporal-specialization [21], and SPEAKER [34], primarily aim to reduce syscalls rather than directly removing code elements.

## 3.3 Evaluation Criteria

In this stage of the workflow, measurable metrics are applied to the artifact before and after debloating to assess its effectiveness from multiple perspectives. As shown in Figure 6, the following are the main evaluation criteria used in the reviewed literature:

- **Performance.** This metric evaluates the performance of the debloated program in terms of its memory usage, CPU utilization, bandwidth, and runtime.
- **Security.** Tools for software debloating, particularly those created by the security community, are designed primarily to improve security and minimize potential attack vectors. Their security assessment predominantly revolves around quantifying the count of Common Vulnerability Exposures (CVEs) and gadgets.
- **Robustness.** This metric is analyzed from various viewpoints: *correctness* and *generality*. The latter evaluates how accurately a debloated program functions with inputs that were not part of the original usage profile [62]. Methods like fuzzing and test cases are used to assess the correctness. Tools like LMCAS [8] and Razor [46] also examine for undesirable behaviors, including incorrect operations, infinite loops, crashes, and missing output.
- **Usability.** This metric focuses on assessing the resources needed by the debloating tool (not the debloated artifact), examined from the perspective of runtime and functionality requirements. For example, Chisel [24] utilizes reinforcement learning along with delta debugging, thus increasing the overhead of running it.
- **Integration.** BLADE [9] views software debloating as essential for ecosystems such as clouds, requiring rapid analysis to support integration with continuous integration and continuous delivery (CI/CD) infrastructures. Consequently, this metric evaluates the capacity of debloating tools to integrate with established ecosystem infrastructures. Despite BLADE's vision, its evaluation did not encompass demonstrating integration capabilities.
- **Sustainability.** This metric evaluates the quality of debloated programs based on carbon footprint and energy reduction. We found only one debloating tool [16] that primarily targets energy reduction and thus focuses on only evaluating this factor.

# 4 FUTURE RESEARCH

This section presents open problems in software debloating and calls for solutions that are practical, usable, and secure.

## 4.1 Software Robustness

Software debloating tools typically prioritize the preservation of error-free paths by utilizing test cases that reflect the intended behavior or providing accurate configurations. As a result, event handler procedures can be removed from the debloated programs, affecting the reliability and robustness of the application. Ancile [11] includes the reachable exception handlers in the final binary. Carve [13] avoids introducing vulnerabilities by replacing debloated code with replacement code that preserves high-level program properties. In some cases, during the debloating process, Carve replaces the *switch* block with exception handling code that traps execution before code blocks that become vulnerable after debloating. However, more work is needed to balance robustness and removal [65].

## 4.2 SBOM Generation

The generation of Software Bills of Materials (SBOMs) has gained significant importance as regulatory bodies like the US National

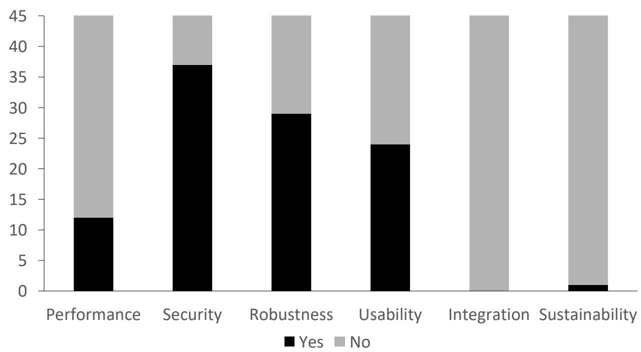


**Figure 6: Evaluation criteria across tools.**

Telecommunications and Information Administration (NTIA) mandate the disclosure of primary and transitive dependencies, thereby documenting the entire code provenance [3]. MMLB [64] constructs dependency trees for ML containers to investigate the impact of debloating on the number of direct and transitive dependencies. Recent dependency management approaches, such as DepsRAG, advocate the use of large language models (LLMs) and knowledge graphs (KGs) to support the generation of SBOMs [7]. Identifying software dependencies constitutes a fundamental aspect of the debloating process, positioning it as a potential facilitator for SBOM generation. The intersection highlights the necessity for further research in this domain.

### 4.3 ML for Debloating

Our investigation indicates that only a limited number of tools (7 out of 48) utilize machine learning (ML) to support debloating. Given the widespread adoption of ML, particularly LLMs, in tasks such as code generation and program repair, there is a compelling need to explore how LLMs can enhance the debloating process.

### 4.4 Debloating Impact on Sustainability

In our literature review, we found only one debloating tool [16] specifically designed to reduce energy consumption. This underscores the need for increased focus and effort in this area. Consequently, we consider this to be an open problem that is worth investigating, especially if new debloating methods can significantly decrease energy use and, as a result, cut down on carbon emissions. Subsequently, researchers might investigate the creation of debloating-driven methods aimed at eliminating software dependencies to achieve energy savings.

### 4.5 CI/CD Integration

Software debloating has often been approached in a siloed manner, which has limited its widespread adoption in real-world scenarios. In today's Software Development Lifecycle (SDLC) and software supply chains, there is a focus on transparency and automation, incorporating practices like CI/CD. CI involves regularly merging code changes from various developers into a central repository, often multiple times per day. CD ensures that the code in the repository is always ready for release, having passed automated tests and quality assessments. Consequently, there are several challenges to address for integrating software debloating tools into CI/CD pipelines [9]. For example, key considerations include determining which test cases should validate a release that includes a debloated version of the application, as well as deciding the necessary security analyses.

Software accreditation presents a significant challenge in integrating software debloating into the CI/CD pipeline. For example, the formal Common Criteria certification process involved independent validation of claims about specific properties of each target of evaluation [1]. Typically, accreditation is performed prior to deployment [38]. Consequently, various approaches can be adopted for CI/CD integration. If debloating occurs post-deployment, as in the case of RAZOR [46], the accreditation process must be repeated. Conversely, if debloating is performed before the software's shipment, accreditation is required only once.

## 5 CONCLUSION

Software debloating is an essential dependency management approach for enhancing both security and performance by removing unnecessary code from applications. Our SoK highlights the diverse techniques and tools available, identifies significant advancements, and points out continuing challenges. We provide a foundational reference, aiming to guide future research and improvements in software debloating.

## ACKNOWLEDGMENTS

This material is based on 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 N00014-24-1-2049. 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.

## REFERENCES

[1] [n. d.]. Common Criteria Publications. https://www.commoncriteriaportal.org/cc/index.cfm.
[2] 2017. U.S. Navy Program Guide. https://media.defense.gov/2020/May/18/2002302043/-1/-1/1/NPG17.PDF. [Accessed 15-06-2024].
[3] 2021. The Minimum Elements For a Software Bill of Materials. https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf. [Accessed 15-06-2024].
[4] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/3359789.3359823
[5] Aatira Ahmad, Mubashir Anwar, Hashim Sharif, Ashish Gehani, and Fareed Zaffar. 2022. Trimmer: Context-Specific Code Reduction. *37th IEEE/ACM Conference on Automated Software Engineering (ASE)* (2022).
[6] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Fareed Zaffar, and Junaid Haroon Siddiqui. 2022. Trimmer: An Automated System for Configuration-Based Software Debloating. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3485–3505. https://doi.org/10.1109/TSE.2021.3095716
[7] Mohannad Alhanahnah, Yazan Boshmaf, and Benoit Baudry. 2024. DepesRAG: Towards Managing Software Dependencies using Large Language Models. *arXiv preprint arXiv:2405.20455* (2024).
[8] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. 2022. Lightweight, Multi-Stage, Compiler-Assisted Application Specialization. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. 251–269. https://doi.org/10.1109/EuroSP53844.2022.00024

[9] Muaz Ali, Rumaisa Habib, Ashish Gehani, Sazzadur Rahaman, and Zartash Uzmi. 2023. Blade: Scalable Source Code Debloating Framework. In *2023 IEEE Secure Development Conference (SecDev)*.

[10] Muaz Ali, Muhammad Muzammil, Faraz Karim, Ayesha Naeem, Rukhshan Haroon, Muhammad Haris, Huzaifah Nadeem, Waseem Sabir, Fahad Shaon, Fareed Zaffar, et al. 2023. SoK: A Tale of Reduction, Security, and Correctness-Evaluating Program Debloating Paradigms and Their Compositions. ESORICS.

[11] Priyam Biswas, Nathan Burow, and Mathias Payer. 2021. Code Specialization through Dynamic Feature Observation. In *11th ACM Conference on Data and Application Security and Privacy* (Virtual Event, USA) *(CODASPY '21)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/3422337.3447844

[12] Michael D. Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Eric Kilmer, and Ronald Eytchison. 2024. A Broad Comparative Evaluation of Software Debloating Tools. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3927–3943. https://www.usenix.org/conference/usenixsecurity24/presentation/brown

[13] Michael D. Brown and Santosh Pande. 2019. CARVE: Practical Security-Focused Software Debloating Using Simple Feature Set Mappings. In *3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (London, United Kingdom) *(FEAST'19)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3338502.3359764

[14] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/3368089.3409738

[15] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. 2020. DECAF: Automatic, Adaptive De-bloating and Hardening of COTS Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1713–1730.

[16] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. 2001. Automatic Source Code Specialization for Energy Reduction. In *International Symposium on Low Power Electronics and Design* (Huntington Beach, California, USA) *(ISLPED '01)*. Association for Computing Machinery, New York, NY, USA, 80–83. https://doi.org/10.1145/383082.383099

[17] Johannes Düsing and Ben Hermann. 2022. Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories. *Digital Threats* 3, 4, Article 38 (feb 2022), 25 pages. https://doi.org/10.1145/3472811

[18] Fábio de A. Farzat, Márcio de O. Barros, and Guilherme H. Travassos. 2021. Evolving JavaScript Code to Reduce Load Time. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1544–1558. https://doi.org/10.1109/TSE.2019.2928293

[19] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1009–1022. https://doi.org/10.1145/3319535.3345665

[20] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458.

[21] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 99, 18 pages.

[22] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-Grained Configuration-Driven System Call Filtering. In *ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1243–1257. https://doi.org/10.1145/3548606.3559366

[23] Muhammad Hassan, Talha Tahir, Muhammad Farrukh, Abdullah Naveed, Anas Naeem, Fahad Shaon, Fareed Zaffar, Ashish Gehani, and Sazzadur Rahaman. 2023. Evaluating Container Debloaters. *8th IEEE Secure Development Conference (SecDev)* (2023). https://doi.org/10.1109/SecDev56634.2023.00023

[24] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[25] Zhenghao Hu and Brendan Dolan-Gavitt. 2022. IRQDebloat: Reducing Driver Attack Surface in Embedded Devices. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1608–1622. https://doi.org/10.1109/SP46214.2022.9833695

[26] Zhenghao Hu, Sangho Lee, and Marcus Peinado. 2023. Hacksaw: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1994–2008. https://doi.org/10.1145/3576915.3623208

[27] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 189–199. https://doi.org/10.1109/ISSRE.2018.00029

[28] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 12–21. https://doi.org/10.1109/COMPSAC.2016.146

[29] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary Reduction of Dependency Graphs. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 556–566. https://doi.org/10.1145/3338906.3338956

[30] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 121–134.

[31] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *12th European Workshop on Systems Security* (Dresden, Germany) *(EuroSec '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3301417.3312501

[32] Taddeus Kroes, Anil Altinay, Joseph Nash, Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, and Cristiano Giuffrida. 2018. BinRec: Attack Surface Reduction Through Dynamic Binary Recovery. In *Workshop on Forming an Ecosystem Around Software Transformation* (Toronto, Canada) *(FEAST '18)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/3273045.3273050

[33] Nham Le, Ashish Gehani, Arie Gurfinkel, Susmit Jha, and Jorge Navas. 2019. Reinforcement Learning Guided Software Debloating. *2nd Workshop on Machine Learning for Systems* (2019).

[34] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-phase execution of application containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*. Springer, 230–251.

[35] J Ligatti, M Abadi, M Bidiu, and U Erlingsson. 2005. Control Flow integrity. In *12th ACM Conference on Computer and communications security*.

[36] Jiakun Liu, Xing Hu, Ferdian Thung, Shahar Maoz, Eran Toch, Debin Gao, and David Lo. 2023. AutoDebloater: Automated Android App Debloating. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2090–2093.

[37] Jiakun Liu, Zicheng Zhang, Xing Hu, Ferdian Thung, Shahar Maoz, Debin Gao, Eran Toch, Zhipeng Zhao, and David Lo. 2024. MiniMon: Minimizing Android Applications with Intelligent Monitoring-Based Debloating. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 990–990.

[38] Miron Livny, Bart Miller, Jim Basney, Von Welch, Irene Landrum, James A Kupsch, Josef Burger, Jeffery Peterson, and Abe Megahed. 2020. Continuous Software Assurance Through a National Marketplace. Final Technical Report, AFRL-RI-RS-TR-2020-214.

[39] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated Software Winnowing. *30th ACM Symposium on Applied Computing (SAC)* (2015). https://doi.org/10.1145/2695664.2695751

[40] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 17–33. https://doi.org/10.1109/EuroSP48549.2020.00010

[41] Parastoo Mohagheghi, Reidar Conradi, Ole M Killi, and Henrik Schwarz. 2004. An empirical study of software reuse vs. defect-density and stability. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 282–291.

[42] Jorge A. Navas and Ashish Gehani. 2023. OCCAM-v2: Combining Static and Dynamic Analysis for Effective and Efficient Whole-Program Specialization. *Commun. ACM* 66, 4 (mar 2023), 40–47. https://doi.org/10.1145/3583112

[43] Pardis Pashakhanloo, Aravind Machiry, Hyonyoung Choi, Anthony Canino, Kihong Heo, Insup Lee, and Mayur Naik. 2022. PacJam: Securing Dependencies Continuously via Package-Oriented Debloating. In *ACM Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 903–916. https://doi.org/10.1145/3488932.3524054

[44] Chris Porter, Sharjeel Khan, and Santosh Pande. 2023. Decker: Attack Surface Reduction via On-Demand Code Mapping. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 192–206. https://doi.org/10.1145/3575693.3575734

[45] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't.

In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 164–180. https://doi.org/10.1145/3385412.3386017

[46] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750.

[47] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 461–476. https://doi.org/10.1145/3372297.3417866

[48] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886.

[49] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3106237.3106271

[50] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. Bintrimmer: Towards static binary debloating through abstract interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 482–501.

[51] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[52] Natarajan Shankar and Ashish Gehani. 2012. Static Previrtualization. *12th High Confidence Software and Systems Conference (HCSS)* (2012).

[53] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE symposium on security and privacy*. IEEE, 574–588.

[54] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2023. Coverage-Based Debloating for Java Bytecode. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 38 (apr 2023), 34 pages. https://doi.org/10.1145/3546948

[55] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45.

[56] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 361–371. https://doi.org/10.1145/3180155.3180236

[57] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2022. XDebloat: Towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4501–4520. https://doi.org/10.1109/TSE.2021.3120213

[58] Xiaoke Wang, Tao Hui, Lei Zhao, and Yueqiang Cheng. 2023. Input-Driven Dynamic Program Debloating for Code-Reuse Attack Mitigation. In *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 934–946. https://doi.org/10.1145/3611643.3616274

[59] Ryan Williams, Tongwei Ren, Lorenzo De Carli, Long Lu, and Gillian Smith. 2021. Guided Feature Identification and Removal for Resource-Constrained Firmware. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 28 (dec 2021), 25 pages. https://doi.org/10.1145/3487568

[60] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. 2021. LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 339–356.

[61] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2021. Subdomain-Based Generality-Aware Debloating. In *35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 224–236. https://doi.org/10.1145/3324884.3416644

[62] Qi Xin, Qirun Zhang, and Alessandro Orso. 2023. Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating. In *37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 99, 13 pages. https://doi.org/10.1145/3551349.3556970

[63] Renjun Ye, Liang Liu, Simin Hu, Fangzhou Zhu, Jingxiu Yang, and Feng Wang. 2021. JSLIM: Reducing the known vulnerabilities of Javascript application by debloating. In *International Symposium on Emerging Information Security and Applications*. Springer, 128–143.

[64] Huaifeng Zhang, Mohannad Alhanahnah, Fahmi Abdulqadir Ahmed, Dyako Fatih, Philipp Leitner, and Ahmed Ali-Eldin. 2024. Machine Learning Systems are Bloated and Vulnerable. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 1, Article 6 (feb 2024), 30 pages. https://doi.org/10.1145/3639032

[65] Huaifeng Zhang, Mohannad Alhanahnah, and Ahmed Ali-Eldin. 2023. BLAFS: A Bloat Aware File System. (2023). arXiv:2305.04641

[66] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. 2022. One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 255–270. https://doi.org/10.1145/3503222.3507768