D-Linker: Debloating Shared Libraries by Relinking From Object Files

Jiatai He, Pengpeng Hou, Jiageng Yu, Ji Qi, Ying Sun, Lijuan Li, Ruilin Zhao, and Yanjun Wu

***Abstract*—Shared libraries are widely used in software development to execute third-party functions. However, the size and complexity of shared libraries tend to increase with the need to support more features, resulting in bloated shared libraries. This leads to resource waste and security issues as a significant amount of generic functionality is included unnecessarily in most scenarios, especially in embedded systems. To address this issue, previous works attempt to debloat shared libraries through binary rewriting or recompilation. However, these works face a tradeoff between flexibility in usage (needs recompilation and runtime support) and the effectiveness of debloating (binary rewriting achieves insufficient file size reduction). We propose D-Linker, a tool that debloats shared libraries by reducing both code and data sections in link-time at the object level without recompilation. Our key insight is that object-level shared library debloating is especially suitable for embedded systems because it strikes a balance of flexibility and efficiency. D-Linker identifies the required ELF object files of the shared libraries in an application and relinks them to produce a debloated shared library with better-debloating effectiveness by avoiding the data reference analysis. Our approach achieves over 70% of gadgets reduction as a security benefit and an average size reduction of 49.6% for a stripped libc of coreutils. The results also indicate that D-Linker improves debloating effectiveness by approximately 30% compared to binary-level shared library debloating and incurs a 5% decrease in code gadgets reduction compared to source-code-level shared library debloating.**

***Index Terms*—Binary debloating, embedded system, shared library.**

I. Introduction

SHARED libraries play a crucial role in modern software development by enabling code reusability and modularity [1], [2], [3]. As opposed to static libraries, which are susceptible to space wastage and complicated updates, shared libraries provide several benefits: 1) sharing a single module among multiple programs eliminates the need to embed the module in each program, thus minimizing space wastage and 2) shared libraries only require an interface, without the need to recompile the program during program updating, which simplifies the process and increases module independence.

However, the proliferation of shared libraries in diverse applications [4] increases their size and complexity, because of supporting for new features while maintaining the old ones. For example, the shared library referenced by the CUDA toolkit has grown almost five times in size from 2012 to 2021 [5]. These bloated portions lead to shared library bloating and cause considerable waste of memory and disk space and security risks, as loading a library involves mapping the entire library into memory, including numerous unused functions and data. A study conducted on a diverse set of 2016 programs across various domains demonstrated that 95% of glibc code is never used [6].

Shared library bloating is a major concern, particularly in scenarios with limited memory and disk storage capabilities like embedded systems [7] or containers. For instance, the size of the Alpine lightweight image that employs musllibc is only 5.59MB, while shared libraries account for 3.9 MB [8]. The bloating of shared libraries not only leads to the waste of storage but also to the inefficient use of memory. Lee et al. [9] demonstrated that loading an entire library, including its unused components, into memory can lead to 10% RAM memory wastage and 35% flush memory wastage in non-MMU embedded systems due to the single address space. Even systems equipped with an MMU can face memory waste due to the 4KB page size and the read-ahead [10] mechanism, which often results in the entire shared library being loaded into physical memory. From the security perspective, failure to address code bloat in shared libraries also frequently leads to return-oriented programming (ROP) gadgets [11] that can be stitched together by the attacker to create malicious attacks. Certain critical vulnerabilities remain exploitable by attackers despite the broadly deployed defenses [12]. To mitigate these issues, the debloating of shared libraries of resource-constrained scenarios like embedded systems and container images is essential.

In resource-constrained embedded systems, different systems support diverse functionalities, while each specific system, such as those in the automotive or avionics industries, typically requires only a subset of the functions provided by shared libraries [13]. This necessitates the debloating of shared libraries to address two primary considerations: 1) due to the limited set of functions required by each specific

Manuscript received 12 August 2024; accepted 12 August 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB4503905, and in part by the HKU-CAS Joint Laboratory for Intelligent System Software. This article was presented at the International Conference on Embedded Software (EMSOFT) 2024 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor S. Dailey. *(Jiatai He and Pengpeng Hou contributed equally to this work.) (Corresponding author: Ji Qi.)*

Jiatai He and Ruilin Zhao are with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 101408, China (e-mail: jiatai21@iscas.ac.cn; zhaoruilin22@mails.ucas.ac.cn).

Pengpeng Hou, Jiageng Yu, Ji Qi, Ying Sun, Lijuan Li, and Yanjun Wu are with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China (e-mail: pengpeng@iscas.ac.cn; jiageng08@iscas.ac.cn; qiji@iscas.ac.cn; sunying@nj.iscas.ac.cn; lilijuan@iscas.ac.cn; yanjun@iscas.ac.cn).

Digital Object Identifier 10.1109/TCAD.2024.3446712

system, redundancy in shared libraries becomes significant, necessitating shared library debloating and 2) due to the diverse requirements of different types of embedded systems, the effort and computational cost associated with customizing and debloating code for each specific system are substantial. This necessitates efficient mechanisms for binary debloating.

Existing work on shared library debloating can be categorized into two types: 1) source-code-level debloating [6], [14], [15], [16] and 2) binary debloating [13], [17], [18].

The source-code-level debloating involves shared library debloating based on the source code, offering precise debloating but incurring the need of source code and associated costs of compilation stage (e.g., lack of flexibility, need source code and runtime support). For example, piecewise compilation and loading (PCL) [6] embeds call graph information extracted from source code in binaries and uses a custom loader to rewrite unnecessary functions with invalid instructions at runtime, whereas BlankIt [14] and Trimmer [15] leverage additional analysis during compilation and load only necessary functions into the program's memory at runtime based on the current execution point.

The binary-level debloating does not need the source code, achieving high flexibility but at the cost of debloating effectiveness. These works use a variety of transformation approaches, including creating specialized libraries via rewriting, to remove or blank unused functions with no-ops or invalid instructions [13], [17], fragmenting libraries [19], relocating symbols to statically link library functions [20], and replacing unnecessary library functions [21]. However, these works often face limitations in control flow analysis at the binary level [22], [23], [24], leading to the need of additional information (e.g., debug information [17] or specific ISAs [18]) and the insufficient reduction in file size.

In this study, our key insight is that object-level debloating is especially suitable for embedded system shared library debloating, because it strikes a balance of flexibility and debloating effectiveness.

First, object-level debloating offers higher flexibility compared to source-code-level debloating. In embedded environments, there is often a large amount of invalid or outdated source code that cannot support additional compilation and runtime mechanisms. In such cases, object-level debloating does not face these issues: 1) static library files are often more readily available than source code, allowing object-level debloating to use static libraries to generate debloated shared libraries and 2) object-level debloating is typically more efficient, as the cost of relinking is much lower than the cost of recompilation.

Second, object-level debloating can erase more unused content compared to binary-level debloating, resulting in better-debloating effectiveness. As binary rewriting cannot resolve data references [22], [23], [24], binary-level debloating only erases the functions in the code section (`.text` section) and cannot resolve the data reference for erasing the unused data in the data section [18]. Moreover, binary overwriting erases the code section by memory page granularity, so code erasing is limited to 4 KiB page size, which would leave some unused code in the library and affect the file size of the debloated library [13]. Compared to binary-level (e.g., Nibbler [17], ELFtailor [13], μTrimmer [18]), debloating shared library by object-level granularity removes unused ELF object files rather than erases binary code, which reducing not only the code sections [25] (e.g., `.text`) but also the data sections (e.g., `.data`) and other sections (e.g., `.rela.dyn`), resulting in further size reduction effectiveness improvement.

Finally, in practical scenarios, when faced with version modifications or functional updates, object-level debloating presents several advantages over both binary-level and source-code-level debloating. Unlike source-code-level debloating, object-level debloating does not require source code or additional runtime mechanisms when updating system functionalities, significantly reducing the cost of updates. Additionally, binary-level debloating needs modifications to the code sections and ELF segment structures, which compromise the maintainability of the shared library by increasing the difficulty in debugging and introducing uncertainties in the ELF format. In contrast, shared libraries produced by object-level debloating are indistinguishable in format from native shared libraries, minimizing additional maintenance costs.

Our insight leads to D-Linker, a tool which debloats shared libraries in link-time at object-level by relinking the ELF object files to the shared libraries. The debloating process of D-Linker has two modes: 1) normal debloating mode and 2) in-depth debloating mode. In normal debloating mode, D-Linker performs binary analysis on the target binary file to generate its function call graph (FCG). Based on the FCG, D-Linker identifies the used object files and then relinks them to create the debloated shared library file with both unused code and data sections eliminated. At the same time, if certain fixed functionalities of the usage scenario are determined, a better-debloating effectiveness can be achieved. Therefore, we propose an in-depth debloating approach, which involves dynamic tracking of shared libraries based on a specific set of user-defined functional tests to achieve a deeper debloating and ultimately obtain a better-debloating effect.

D-Linker faces two challenges: 1) how to handle data dependencies and 2) the completeness of symbol dependency. These issues are typically resolved in normal debloating through inherent symbol dependencies. However, in the case of in-depth debloating, due to the inability to track data symbols dynamically and the incompleteness of symbols during linking, additional mechanisms are required to address these challenges. We have adopted an object data dependency analysis method to handle the data dependency issue in in-depth debloating and use binary modification to address symbol incompleteness in in-depth debloating.

We summarize our major contributions as follows.

1) We introduce D-Linker, an innovative technique for debloating shared libraries by relinking from ELF object files. Our approach offers the ability to reduce the size of not only the code section but also the data and other sections, achieving high-reduction effectiveness.
2) D-Linker efficiently leverages static libraries to extract object files during the debloating process of most shared libraries, resulting in no need of compilation and source

code (with a few exceptions, as elaborated in Section V), achieving high flexibility.
3) We evaluated D-Linker and prior methods for debloating shared libraries, focusing on key aspects, such as debloating effectiveness (i.e., file size reduction), runtime overhead, and security analysis. The results indicate that D-Linker improved debloating effectiveness by approximately 30% compared to the binary-level debloating and incurred only a 5% decrease of code gadgets reduction compared to the source-code-level debloating.

II. Background and Related Work

Binary debloating has been a hot topic in security and embedded systems. Mulliner and Neugschwandtner [26] first proposed the specialization of shared libraries for runtime environments, with the core idea of keeping the functions needed in the shared libraries for different scenarios and removing the functions not needed in the shared libraries at load time. Existing works can be categorized into the following two approaches.

Source-Code-Level Debloating: Quach et al. [6] proposed piecewise using llvm compiler for a compiler and loader-assisted mechanism to reduce executable code loaded by a process. They analyze the indirect points reference by interprocedure static value-flow analysis and saving a full-program dependency graph. Then they load the functions that are present in the dependency graph at load time. This approach worked well in removing useless code, but masking functions at load time incurs additional performance overhead, requiring source code and recompiling. Similar works, BlankIt proposed by Porter et al. [14] and Trimmer by Sharif et al. [15] BlankIt feed the functions that the user needs to call into a linked library using a decision tree predictor to reduce the exposed surface of the code, which reduces the code exposure surface by an average of 96%, but also increases the average runtime overhead by 18%.

Binary-Level Debloating: The benefit of debloating shared libraries by binary analysis and erasing is to remove unused code without source code and recompilation, which can be used in more scenarios than debloating during compilation. Ziegler et al. [13] proposed ELFtailor, which debloats shared libraries in two steps: 1) running binary static analysis using capstone [27] and dynamic analysis using uprobes [28] to obtain the FCG of the application and 2) overwriting and erasing the functions not in the FCG and reorganizing the shared libraries. A similar work is proposed by Ioannis et al. Nibbler [17], which obtains FCG of shared library unstripped statically. Then, Zhang et al. [18] proposed an MIPS-based shared library debloating tool μTrimmer. Due to the MIPS instruction set, indirect pointer calls to functions can be thoroughly analyzed statically in shared libraries stripped, which is a big step forward in binary analysis. μTrimmer almost reaches the upper limit of the code debloating of shared libraries. At the same time, these works also mention that although the code section is debloated, they still cannot resolve data reference, which means binary rewriting cannot debloat the data section of the shared library.

III. Overview of D-Linker

A. Overview

We present D-Linker, a tool that aims to debloat shared libraries by relinking the ELF object files.

Debloating Modes: Our debloating process has two modes: 1) normal debloating mode and 2) in-depth debloating mode. In the normal debloating mode, the set of object files that are used is selected based on the dependencies between them, and then relinked to create the debloated shared library with both unused code and data sections eliminated. However, during the debloating process, there may still be unused object files in the set due to the presence of redundant dependencies. In order to further improve the debloating effect, we introduce the in-depth debloating mode. This mode involves leveraging the test suite of the target program and employing binary rewriting techniques to identify and eliminate unused object files. By employing this approach, we aim to achieve more effective debloating effectiveness. We will conduct a more detailed analysis of this in following sections.

Challenges: To achieve the above design, we encountered two challenges, which will be discussed in Section IV-B.

Challenge 1 (Data Reference): In the process of in-depth debloating, we need to erase the references to the unused object files identified by dynamic analysis. However, dynamic analysis is inherently limited to detecting called functions and does not adequately address data references. This limitation presents a significant challenge: How can we precisely identify object files that contain critical data references and therefore cannot be removed during in-depth debloating? To address this challenge, we employ a method to handle data reference when removing object files.

Challenge 2 (The Completeness of Symbol Dependency): Another challenge is to address the issue of symbol dependency completeness, which primarily needs to be resolved in in-depth debloating. Relinking the object files tracked by dynamic analysis requires breaking dependencies between some object files to achieve a better-debloating effectiveness. To address this issue, we employ binary rewriting to modify symbol attributes, thereby breaking dependencies between certain objects.

Overview: Our methodology consists of three main steps. First, we generate the FCG by conducting static and dynamic binary analysis, with dynamic analysis performed for in-depth debloating mode. Second, we identify the used object files and their dependencies. Finally, we modify and relink the necessary object files to generate the debloated library. Fig. 1 provides a visual representation of our approach, which comprises three components: 1) FCG reconstruction; 2) identifying object files; and 3) relinking. Together, these components involve 7 steps, with steps 3 and 6 only applicable in cases where in-depth debloating is necessary.

FCG Reconstruction—Steps 1–4: To reconstruct the FCG of the application and shared library, we employ a four-step process. First, we analyze the dependency between shared

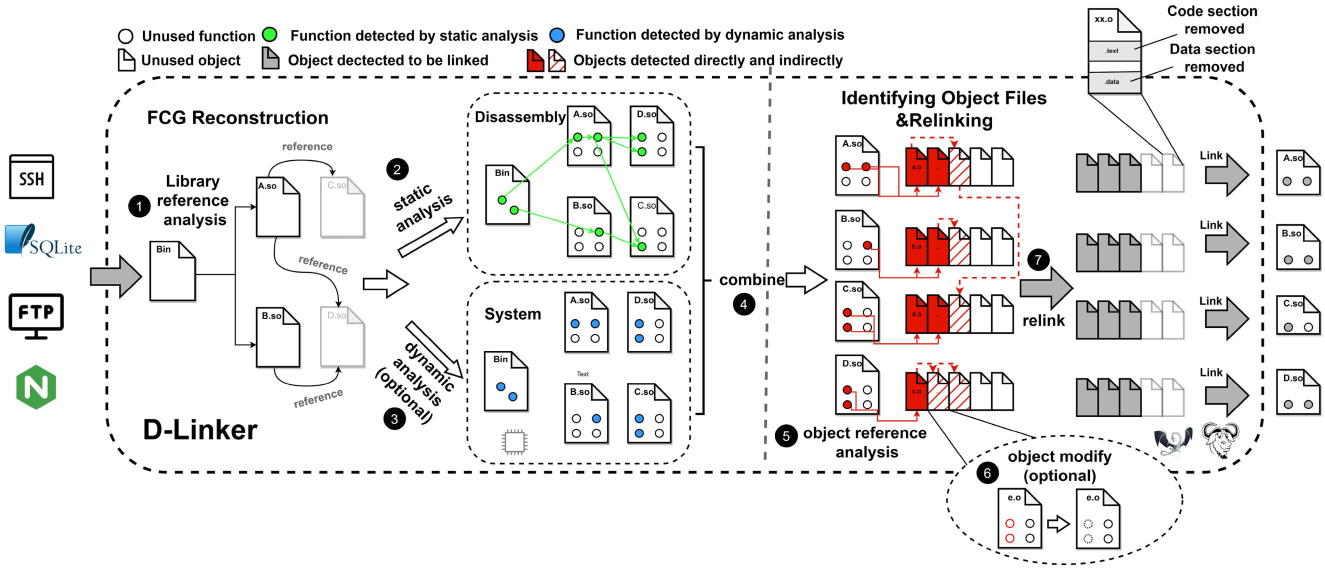

Fig. 1. Overview of D-Linker.

libraries by identifying the required shared libraries and their dependencies based on the `.dynamic` section of binary (step 1, Fig. 1 ①). Second, we perform static binary analysis and disassemble the application and shared libraries to generate the FCG of the shared libraries (step 2, Fig. 1 ②). Third, if in-depth debloating is required by user, we validate the used functions of the shared libraries through dynamic analysis using uprobes (step 3, Fig. 1 ③). Finally, we merge the results from Step 2 and Step 3 (if applicable) to obtain a comprehensive FCG (step 4, Fig. 1 ④).

Identifying Object Files—Step 5: After establishing the FCG of the shared libraries in Step 4, the subsequent step is to identify the used object files based on the guaranteed FCG. In cases where in-depth debloating is required, we also pinpoint the redundant symbol dependency between object files (step 5, Fig. 1 ⑤).

Relinking—Steps 6 and 7: The relinking process consists of two steps. First, if in-depth debloating is necessary, we modify the object files identified in Step 5 and eliminate any redundant symbol dependencies (step 6, Fig. 1 ⑥). Finally, we relink the used object files to generate debloated shared libraries, discarding unused object files that contain both unused data and code (step 7, Fig. 1 ⑦).

Debloating Shared Libraries of the Total System: Debloating shared libraries within an total system (e.g., IoT devices, containers) is a common use case of D-Linker, particularly in the context of embedded systems that encompass numerous binaries. D-Linker takes all the binaries of the embedded system as input, following a procedure analogous to the debloating of a single binary. The resultant debloated shared libraries are relinked using the union of the object files requisite for all binaries. In Section VI, we demonstrate the efficacy of D-Linker in such scenarios by debloating the shared libraries of an Alpine Linux system running vsftpd, showcasing its significant effectiveness improvements in this context.

IV. Detailed Design of D-Linker

A. FCG Reconstruction

The reconstruction of the FCG involves three steps. First, a shared libraries dependency analysis is performed to discover all required shared libraries and their dependencies relevant to the target application. This step ensures a comprehensive understanding of the libraries required. Second, a static binary analysis is conducted using disassembly tools to establish the FCG of the shared libraries. However, it should be noted that static analysis tools may not be able to detect all used interfaces. Thus, further runtime dynamic analysis is necessary when in-depth debloating is required and the result of dynamic analysis would be combined with the FCG established by static analysis.

Shared Libraries Dependency Analysis: The method used for shared libraries dependency analysis involves two steps. First, after loading the application, D-Linker detects the dependent shared libraries based on the information stored in `DT_NEEDED`, which is an item in the `.dynamic` section. This step ensures that the directly dependent shared libraries are identified. Second, the shared libraries identified in the previous step recursively search for their own dependent shared libraries using the same method. This recursive process continues until a complete shared library dependency graph is established, encompassing all the dependencies.

Once the shared library dependency graph is obtained, D-Linker proceeds to detect the undefined symbols in both the shared libraries and the application. This step involves identifying symbols that are referenced but not defined within the code. By identifying these undefined symbols, D-Linker is able to gather detailed reference information about how the shared libraries rely on each other. This information helps in understanding the relationships and interactions between the shared libraries, which is crucial for the debloating process.

Static Binary Analysis: During static binary analysis, the initial step involves disassembling both the applications and

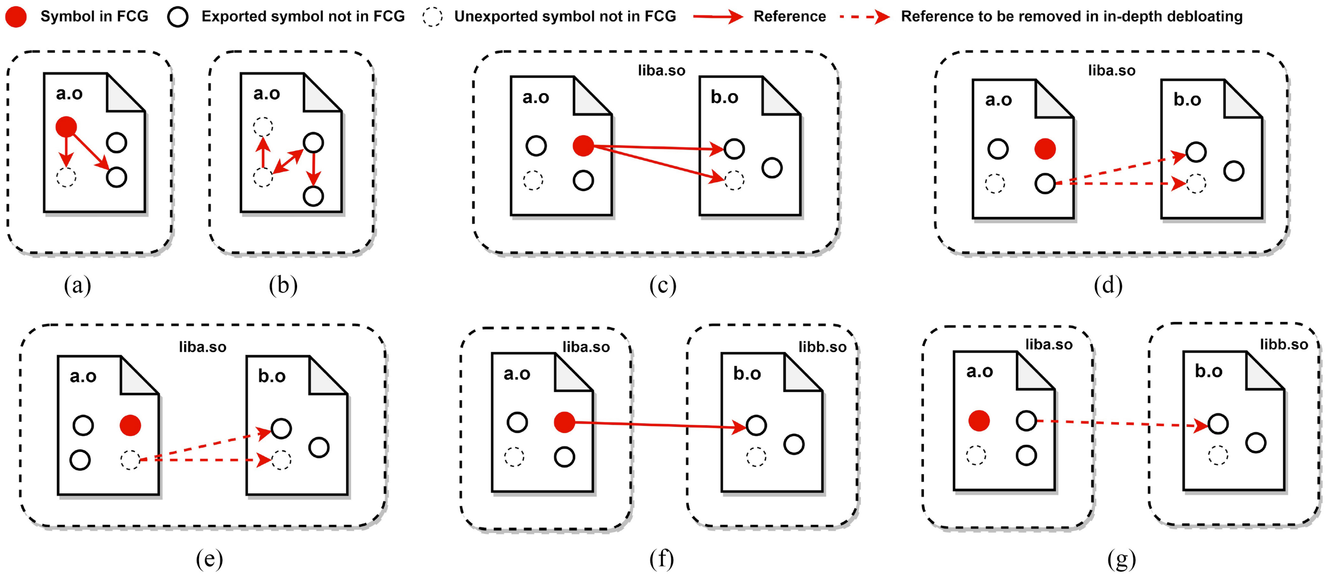

Fig. 2. Reference of object files. (a) Symbols in FCG require symbols in same object. (b) Symbols not in FCG require symbols in same object. (c) Symbols in FCG require symbols in another object but in the same shared library. (d) Default global system not in FCG require symbols in another object but in the same shared library. (e) Hidden global symbol not in FCG require symbols in another object but in the same shared library. (f) Symbols in FCG require symbols in another shared library. (g) Symbols not in FCG require symbols in another shared library.

shared libraries. Subsequently, the call relationships within the execution path of the binary file are determined by examining its disassembly code and symbolic information. These findings are then utilized to reconstruct the FCG. It is important to note that the FCG solely encompasses global symbols from the shared libraries, as the handling of local symbol references is performed by our object-level debloating technique.

The FCG described here is adequate for generating necessary object files to produce normal debloating shared libraries, thanks to the symbolic dependences. However, in cases where in-depth debloating is needed, dynamic analysis becomes necessary.

Dynamic Binary Analysis: Dynamic binary analysis serves as a complementary counterpart to static binary analysis by detecting all used functions through runtime dynamic analysis using uprobes. To detect functions called by function pointers and virtual functions in C++ that cannot be found through static analysis, uprobes are deployed to all the global symbols of shared libraries required by the application. The uprobes are triggered upon executing the application's test suite, resulting in a list of called functions. This list is then merged with the FCG obtained through static binary analysis to yield the final FCG for the shared libraries. To obtain a precise representation of the target use case, it is necessary for the dynamic analysis test suite to cover all functionalities essential in the final system. This goal can be achieved by adhering to strictly defined procedures for interacting with the system and utilizing extensive test suites, as required for certification purposes. This requirement is particularly relevant in industries, such as automotive engineering.

Dynamic analysis is employed to facilitate in-depth debloating, which aims to eliminate unnecessary runtime dependencies and improve the effectiveness of debloating outcomes. However, it is important to note that dynamic analysis relies on the coverage provided by the test suite. Consequently, it cannot guarantee the proper functioning of functions that are not covered by the tests. This limitation can result in certain stability compromises.

B. Relinking

After obtaining the FCG in shared libraries, we select the used object files based on the FCG to relink the debloated shared libraries. In this section, we will discuss how D-Linker handles the reference between object files especially the data reference (Challenge 1) and relinks the object files with incomplete symbol dependency (Challenge 2).

Symbols in Object Files: Symbols within object files are classified into two types: 1) symbols included in the FCG and 2) symbols not included in the FCG, as shown in Fig. 2.

Resolve Challenge 1: We employ a method to generate a set of used object files with safe data reference. The method for generating a set of used object files is outlined as follows:

First, we categorize the dependencies into two types: 1) data dependencies, which refer to references to data type symbols and 2) function dependencies, which refer to references to function type symbols.

Second, we generate a full dependency set, encompassing both function and data dependencies, and a data dependency set for each object file. This process is described in Algorithm 1 as follows: for every object file, we add its dependent object files to the dependency set, continuing this process for all object files in the set until there are no more references to external object files. When constructing the full dependency set, we consider both function and data dependencies. However, when forming the data dependency set, we focus exclusively on data dependencies.

Algorithm 1 Get Full Dependency Set and Data Dependency Set

Require: Object files of a shared library O;
Ensure: full dependency set F and data dependency set D;

function DFSFINDDEPENDENCY(o, *dependency*)
  if *dependency* == *full* **then**
    $F(o).add(o)$
    /* x is a object file referenced by o */
    for $x \in fullreference(o)$ **do**
      if $x \notin F(o)$ **then**
        DFSFINDDEPENDENCY(x, *full*)
        $F(o).add(F(x))$
      end if
    end for
  else if *dependency* == *data* **then**
    $D(o).add(o)$
    /* x is a object file data referenced by o */
    for $x \in datareference(o)$ **do**
      if $x \notin D(o)$ **then**
        DFSFINDDEPENDENCY(x, *full*)
        $D(o).add(D(x))$
      end if
    end for
  end if
end function
for o in O **do**
  DFSFINDDEPENDENCY(o, *full*)
  DFSFINDDEPENDENCY(o, *data*)
end for

Finally, the used object set in normal debloating consists of the object files that contain the symbols appearing in the FCG and their associated full dependency set. In the context of in-depth debloating, the used object set comprises the object files for all symbols present in the FCG, obtained from both static and dynamic analysis, as well as their corresponding data dependency sets.

It is significant to highlight the dependencies that were eliminated during the debloating process. We specifically consider single-step references between object files, as recursive single-step references can resolve the overall dependency. In Fig. 2, the references to symbols within object files are depicted. Fig. 2(a) and (b) correspond to references within the same object file, but these do not need to be taken into account due to the object-level granularity of debloating. Fig. 2(c) contains two types of symbol references in the FCG: 1) references to symbols exported by the shared library in other objects and 2) references to symbols not exported by the shared library in other objects. These references will be retained after debloating. Additionally, Fig. 2(d) and (e) illustrate additional references of symbols that are not present in the FCG but exist within the same object file as the symbols in the FCG. These references arise due to the object-level debloating granularity and will be kept in normal debloating. In cases where in-depth debloating is required, these references will be eliminated through binary rewriting. When dealing with cross-shared libraries, it is necessary to distinguish between the references from symbols in the FCG to other shared libraries (Fig. 2(f) should not be removed) and the references from symbols not in the FCG to other shared libraries (Fig. 2(g) should be removed). In normal debloating, Fig. 2(g) will also be retained, but this kind of reference will be removed through binary rewriting when in-depth debloating is required.

Resolve Challenge 2: To remove the references in in-depth debloating, we need to perform binary rewriting to change the attribute of undefined symbols which reference symbols in removed object files from "`undefined`" to "`weak`". This is necessary to ensure that we avoid causing "`symbol undefined`" errors when we load the library linked from the object files into memory. After performing the binary rewriting, we can proceed to relink the object files and obtain the debloated library.

TABLE I
REDUCTION OF LIBRARY TOTAL SIZE

Program	File size		
	Baseline	**Tailored**	**Reduction**
Nginx			
libc	612KB	169.8KB	-72.3%
libcrypto	1.8MB	265.7KB	-85.3%
libpcre	153.4KB	120.6KB	-21.4%
libz	117KB	59.2KB	-49.5%
Coreutils			
libc	612KB	325KB	-46.9%
Sqlite			
libc	612KB	170.2KB	-72.1%
libz	117KB	88KB	-25.8%
libreadline	379KB	370.6KB	-2.3%
libncurses	398KB	131.1KB	-66.9%
Openssh			
libc	612KB	198.9KB	-67.5%
libcrypto	1.8MB	1.7MB	-5.6%
libz	117KB	96.8KB	-17.3%
Vsftpd			
libc	612KB	166.1KB	-72.9%
libcrypto	1.8MB	1.5MB	-16.7%
libssl	406KB	375.2KB	-7.6%
Vsftpd(in-depth)			
libc	612KB	166.1KB	-72.9%
libcrypto	1.8MB	1.1MB	-38.9%
libssl	406KB	288.2KB	-29.1%
Alpine(in-depth)			
libc	612KB	358.6KB	-41.5%
libcrypto	1.8MB	1.1MB	-38.9%
libssl	406KB	288.2KB	-29.1%

C. Data Reference (Compared to Binary-Level Debloating)

One notable advantage of D-Linker over previous binary-level debloating methods is that we utilize object files to debloat various sections in binary besides code section, such as data sections and relocation tables. In this section, we will introduce how D-Linker reduces the size of the data sections in shared libraries.

Limitation to Binary Analysis Debloating Tools: Prior debloating tools that rely on binary analysis and rewriting are unable to reduce data sections due to several challenges. One of the primary difficulties lies in the fact that runtime operations often involve references to data and updates to pointers. Existing binary analysis techniques have difficulty in providing an accurate analysis of the indirect control flow of data access, making it challenging to reduce data sections [18],

[22], [23], [24]. Additionally, conventional methods, such as deploying probes, that set breakpoint instructions at the function entry point are insufficient to detect access to data blocks. These limitations make it challenging to resolve data references using symbol tables and disassembly code with existing techniques. Hence, debloating data sections with such tools has been a significant challenge.

Why D-Linker can Debloat Data Sections? D-Linker employs an innovative method to debloat shared libraries through relinking from object files. This allows direct debloating of data sections and circumvents the complexities of analyzing control flows of data access. In object files, functions access data either via symbolic tables from other object files or within the object itself. Upon linking a shared library, these data blocks consolidate, leading to runtime calculations and updated pointers, complicating reference resolution via binary analysis.

D-Linker addresses this by utilizing symbol reference analysis between object files, eliminating the need to examine consolidated data blocks within shared libraries. Additionally, by debloating at the object level, D-Linker mitigates concerns about data access within object file. Consequently, by selecting necessary data blocks and discarding redundant object files before linking, D-Linker effectively reduces data section sizes without delving into indirect control flow analysis.

V. IMPLEMENTATION

In this section, we provide details on the implementation of D-Linker.

Binary Analysis: We use static and dynamic binary analysis to generate the complete FCG: the static analysis part uses capstone [27] and objdump for binary disassembly; the dynamic analysis part is done by using uprobes to get the triggered functions. We use python and nm [29] to read the symbol table of ELF files and resolve dependencies between object files. For the shared library files opened by `dlopen()`, D-Linker needs to know in advance the shared libraries and their object file locations that need to be opened. Static analysis cannot guarantee the correctness of its FCG, and dynamic analysis is required to ensure its correctness.

Obtain the Object Files: Due to the fact that the majority of shared libraries have corresponding static libraries, D-Linker can extract object files from the static libraries, which enables D-Linker to debloat shared libraries without source code in most cases. In our evaluation, aside from libc, which required additional object files for dynamic linking (e.g., dlstart.o), all other shared libraries were debloated by relinking the object files extracted from the static libraries without the source code.

VI. EVALUATION

We conducted an evaluation of D-Linker on Ubuntu 20.04, which is a Linux operating system running on x86-64 architecture. To test the effectiveness, we executed various applications, including Nginx (v1.9.2), Coreutils (v9.1), Sqlite (v3.40.0), Openssh (v7.3), and Vsftpd (v3.03), to evaluate their effectiveness in reducing bloat. All of the aforementioned applications were implemented using musllibc (v1.2.3) [30] as the C library.

For in-depth debloating, we selected Vsftpd, which is a lightweight and secure FTP server for Unix-like systems, used in debloating evaluation of prior work [13] and a complete Alpine [8] system which is often used in docker and IoT as our evaluation targets. We developed a testing script for Vsftpd (both normal and in-depth debloating), encompassing various tasks, including user login, logout, file upload, and download, with SSL transfers facilitated by Vsftpd. During the dynamic analysis phase, this script was deployed on both a standalone Vsftpd application and a whole Alpine system equipped with Vsftpd. Following the debloating procedure, we persistently employed this script as a test case to authenticate the accuracy of our in-depth debloating methodology.

Following is the structure of this section. In Section VI-A, we analyze the file size of debloated shared libraries. In Section VI-B, we discusses the effectiveness of debloating object files and functions, evaluating the dependencies of the object files. In Section VI-C, we analyze the debloating of data sections. In Section VI-D, we carried out an investigation into the dependencies among object files within multiple shared libraries to conduct further analysis of our debloating results. In Sections VI-E and VI-F, we evaluate the security benefits of D-Linker by analyzing reduction of gadgets and compare D-Linker with related works. In Section VI-F, we quantitatively compared the code debloating capabilities of D-Linker and ELFtailor [13], as well as the security benefits observed with D-Linker and piecewise. Finally, we evaluated D-Linker's effectiveness against other debloating techniques in terms of file size reduction, runtime availability, and security analysis, demonstrating D-Linker's superior advantages in embedded scenarios.

A. Reduction of File Size

Reduction of Normal Debloating: Table I presents the file sizes of shared libraries for both debloated and baseline (stripped) versions across various target applications(the unannotated applications are normal debloating, and the same applies to the following tables). The Program column denotes the shared libraries associated with each target application case, noting that Vsftpd does not debloat libgcc_s. D-Linker achieved the highest reduction in Nginx, minimizing 77.4% of its shared library size, whereas Openssh had the lowest reduction at 21.2%. The variation in debloating efficacy among different applications primarily stems from the disparate effectiveness in debloating their dependent libraries, which we will elaborate on. Specifically, D-Linker consistently performed best with libc across all cases, achieving a reduction of 72.3% for Nginx and 49.6% for Coreutils. This efficacy is attributed to musllibc's extensive set of functions, including multithreading and mathematical libraries, which are largely unused and thus removable. Other notable results include an 86.4% reduction in libcrypto for Nginx. In contrast, debloating is less effective for libraries closely aligned with application requirements; for example, libreadline used in Sqlite was only reduced by 3%. This limited reduction is due to the significant

TABLE II
REDUCED OBJECT FILES OF THE LIBRARY

Program	Number of object files		
	Baseline	Tailored	Reduction
Nginx			
libc	1341	366	-72.7%
libcrypto	556	25	-95.5%
libpcre	22	9	-59.1%
libz	16	5	-68.8%
Coreutils			
libc	1341	464	-65.4%
Sqlite			
libc	1341	377	-71.9%
libz	16	10	-37.5%
libreadline	35	33	-5.7%
libncurses	149	31	-79.2%
Openssh			
libc	1341	456	-66.0%
libcrypto	556	404	-27.3%
libz	16	8	-50.0%
Vsftpd			
libc	1341	349	-74.0%
libcrypto	556	390	-29.9%
libssl	46	38	-17.4%
Vsftpd(In-depth)			
libc	1341	349	-74.0%
libcrypto	556	233	-58.1%
libssl	46	24	-47.8%
Alpine(In-depth)			
libc	1341	508	-62.1%
libcrypto	556	233	-58.1%
libssl	46	24	-47.8%

functional overlap between the library and application, coupled with the dense distribution of functions and data blocks within the library's object files, leading to a retention of many superfluous functions.

Reduction of Indepth Debloating: The results of the in-depth debloating experiment demonstrate a significant improvement in the overall reduction effect of vsftpd, by approximately 23.8%, compared to normal debloating. Regarding the reduction of libraries, the libcrypto library was identified as the most significant, resulting in an additional reduction of approximately 38%. The libssl library also achieved a reduction of above 29.1%, which achieved an implementation above 20% compared to normal debloating.

Overall, D-Linker's normal debloating and in-depth debloating achieved satisfactory results in most cases, demonstrating D-Linker's effectiveness in file size reduction. However, certain shared libraries (e.g., libssl) do not exhibit similar effectiveness in debloating. This issue will be discussed in detail in Section VI-D.

B. Reduction of Object Files and Functions

Reduction of Normal Debloating: Tables II and III summarize the normal debloating results for object files and function symbols across five applications. Our method showed significant reductions, particularly with libc, which was reduced by 66%-74% in object files and approximately 70% in functions. Similarly, libcrypto usage in Nginx was reduced by 95.5%, requiring only 25 out of 556 object files. In contrast, libssl and libreadline demonstrated minimal reduction, with libssl in Vsftpd and libreadline in Sqlite having only 8 and 9 object files debloated, respectively, and this will be discussed in Section VI-D.

Reduction of Indepth Debloating: In the case of in-depth debloating, the reduction of function symbols achieved a 42.2% improvement and the reduction of object files achieved a 20.2% improvement compared to normal debloating. Furthermore, it was evident that the reduction of libcrypto surpasses that of libssl. This will be discussed in Section VI-D.

Overall, we can observe that D-Linker removed plenty of redundant object files, thereby eliminating redundant functions. This indicates that, in most scenarios, some redundant object files are linked into the shared libraries. Therefore, for resource-constrained environments, object-level debloating proved to be an efficient approach.

C. Reduction of Data Sections

A clear advantage of D-Linker over previous works is that D-Linker can debloat data sections (e.g., `.data` and `.rodata` sections) in a shared library, bypassing the accurate analysis of the control flow of data access.

Tables III and IV summarize the number of data symbols and the size of the data section debloated by D-Linker. The data symbols were counted from the symbol table of the shared library; the size of the data section was read from the `.data` and `.rodata` sections (in addition to the code and data sections, other sections, such as `.rela.dyn`, were also debloated, and here, we are only concerned with the debloating of the data part). In normal debloating, nginx achieved the most efficient reduction above 70% in data size and 78% in data symbol.

In the case of in-depth debloating, the in-depth debloating technique achieved an improvement of 20.6% in data symbol reduction and 11.3% in size reduction compared to normal debloating. In summary, the effectiveness in the libc library was excellent both in terms of number (about 70% on average) and size (about 80% on average), while the effectiveness in the libssl (6% on number and 10% on size) and libreadline (almost no debloating) were not noticeable. The reason of this will also be discussed in Section VI-D.

Overall, we can observe that, through object-level debloating, D-Linker effectively debloated the data sections in shared libraries.

D. Analysis of Object File Dependency

As the analysis from above sections, the effectiveness of D-Linker on the different shared libraries was different. The main reason is the different organization of object file dependence in shared libraries, and this section analyzes the object file dependencies in different libraries to evaluate the unequal effectiveness of debloating is related to the object file dependence in shared libraries.

Tables I–III show that shared libraries, such as libc, had a more excellent debloating effectiveness. In contrast, libraries such as libssl and libreadline were poor, mainly related to the dependencies of object files in shared libraries. The reasons are as follows.

TABLE III
REDUCED SYMBOLS OF THE LIBRARY

Program	Export func			Local func			Data		
	Baseline	Tailored	Reduction	Baseline	Tailored	Reduction	Baseline	Tailored	Reduction
Nginx									
libc	1670	403	-75.9%	435	245	-43.7%	498	239	-52.0%
libcrypto	3563	320	-91.0%	823	62	-92.5%	1318	97	-92.6%
libpcre	27	9	-66.7%	50	42	-16.0%	51	47	-7.8%
libz	88	32	-63.6%	38	23	-39.5%	66	29	-56.1%
Coreutils									
libc	1670	617	-63.1%	435	232	-46.7%	498	239	-52.0%
Sqlite									
libc	1670	424	-74.6%	435	207	-52.4%	498	198	-60.2%
libz	88	55	-37.5%	38	26	-31.6%	66	38	-42.4%
libreadline	490	482	-1.6%	227	222	-2.2%	700	700	0.0%
libncurses	464	99	-78.7%	192	37	-80.7%	194	166	-14.4%
Openssh									
libc	1670	548	-67.2%	435	261	-40.0%	498	289	-42.0%
libcrypto	3563	2873	-19.4%	823	690	-16.2%	1318	1173	-11.0%
libz	88	50	-43.2%	38	26	-31.6%	66	38	-42.4%
Vsftpd									
libc	1670	388	-76.8%	435	252	-42.1%	498	258	-48.2%
libcrypto	3563	2878	-19.2%	823	673	-18.2%	1318	1133	-14.0%
libssl	483	453	-6.2%	63	51	-19.0%	74	70	-5.4%
Vsftpd(In-depth)									
libc	1670	388	-76.8%	435	252	-42.1%	498	258	-48.2%
libcrypto	3563	1837	-48.4%	823	541	-34.3%	1318	846	-35.8%
libssl	483	395	-18.2%	63	34	-46.0%	74	56	-24.3%
Alpine(In-depth)									
libc	1670	666	-60.1%	435	266	-38.9%	498	296	-40.6%
libcrypto	3563	1837	-48.4%	823	541	-34.3%	1318	846	-35.8%
libssl	483	395	-18.2%	63	34	-46.0%	74	56	-24.3%

TABLE IV
REDUCED DATA OF THE LIBRARY

Program	Data size		
	Baseline	Tailored	Reduction
Nginx			
libc	218432	34607	-84.2%
libcrypto	188424	41232	-78.1%
libpcre	13872	11480	-17.2%
libz	18426	12248	-33.5%
Coreutils			
libc	218432	166710	-23.7%
Sqlite			
libc	218432	39501	-81.9%
libz	18426	15432	-16.2%
libreadline	28254	28254	0.0%
libncurses	46915	14049	-70.1%
Openssh			
libc	218432	30189	-86.2%
libcrypto	188424	171604	-8.9%
libz	18426	15427	-16.3%
Vsftpd			
libc	218432	23868	-89.1%
libcrypto	188424	158400	-15.9%
libssl	53045	37732	-28.9%
Vsftpd(In-depth)			
libc	218432	23868	-89.1%
libcrypto	188424	133610	-29.1%
libssl	53045	37732	-28.9%
Alpine(In-depth)			
libc	218432	182788	-16.3%
libcrypto	188424	133610	-29.1%
libssl	53045	37732	-28.9%

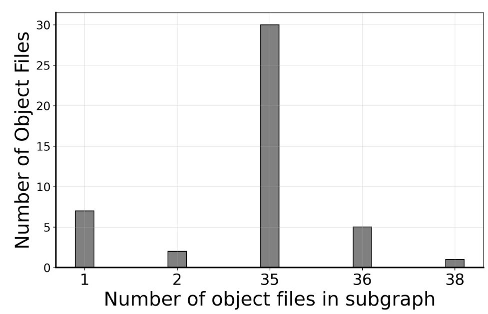

Fig. 3. Distribution of symbols in Libssl and Libc.

Number of Symbols in Object Files: The number of object files and functions in shared libraries is not proportional; libraries with fewer symbols per object file exhibit better-debloating effectiveness. For instance, libc, the shared library with the most object files (1341), has 1670 exported functions, whereas libcrypto has 556 object files but 3563 functions, and libssl has 46 object files with 489 functions. Consequently, libc's sparser distribution of functions results in superior debloating effectiveness. Detailed analysis (Fig. 3) shows that libc's object files uniformly contain fewer symbols (within 5), while libssl has object files with many symbols, such as `ssl_lib.o` with 148 symbols. This dependency on heavily populated object files renders libssl essential to the shared library, leading to poor debloating effectiveness.

Reference Graph of the Library: We also analyzed the reference graph of libssl, as shown in Fig. 4, the reference graph between the object files in libssl; we can see that most of the nodes have a large degree of entry and exit, which causes the entire graph to be distributed more aggregated and thus the number of complete subgraphs (smallest subgraph without undefined symbols) is small. As shown in Fig. 5, in libssl, 80% of the object files are in the subgraph with more than

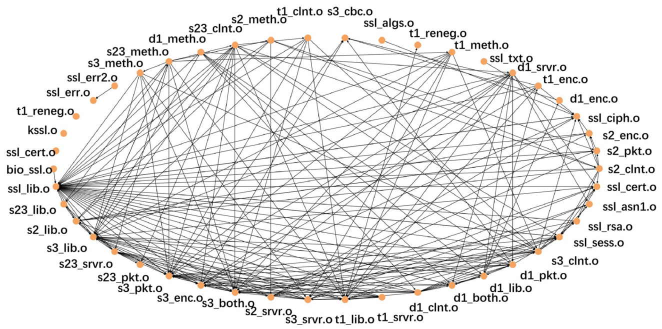

Fig. 4. Reference graph of Libssl.

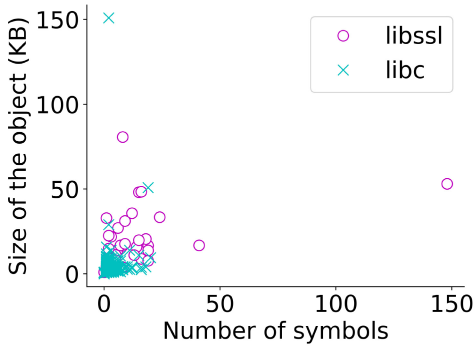

Fig. 5. Object files in the subgraph of Libssl.

35 object files, which makes access to the symbols in these objects necessarily dependent on most of the object files in libssl and also causes poor debloating effectiveness because the subgraphs contain numerous nodes (object files).

E. Security Evaluation

We used D-Linker to measure four common gadget types [11]: 1) syscall; 2) stack pointer update (SPU); 3) jump-oriented programming (JOP); and 4) call-oriented programming (COP). As shown in Table V, D-Linker performed exceptionally well in removing the syscall gadget class, with its reduction ratio far exceeding that of the corresponding code reduction. We found that most of the erased syscall instructions come from unused functions in musllibc. The reduction for SPU and JOP types was similar to our achieved code reduction. Overall, this experiment indicated that D-Linker can significantly increase the costs for adversaries to launch code-reuse attacks. Furthermore, the security benefits of D-Linker will be compared with piecewise [6, Sec. VI-F].

F. Comparative Analysis With Prior Work

We divide our comparative experiments into three parts. In the first part, we aim to validate the effectiveness of our embedded pruning approach by comparing it with Elftailor [13], which also targets embedded scenarios. We will evaluate the shared library size, memory usage, code section size, and the amount of dead code executable code. In the second part, we assess the security benefits of our approach by comparing it with piecewise, a prior debloating work that also targets musllibc, in order to demonstrate the security improvements achieved by D-Linker. In the third part, we conduct a functional comparison of our pruning methodology with all known binary pruning techniques to elucidate the advantages and limitations of D-Linker.

TABLE V
GADGETS REDUCTION OF D-LINKER

Program	Reduction of Gadgets				
	Total	SPU	COP	JOP	Syscall
Nginx					
libc	-66.77%	-73.26%	-53.55%	-67.30%	-52.94%
libcrypto	-88.99%	-83.87%	-89.03%	-90.64%	N/A
libpcre	-26.04%	-5.80%	-10.81%	-26.34%	N/A
libz	-60.55%	-68.00%	-70.14%	-64.02%	N/A
Coreutils					
libc	-59.64%	-67.69%	-41.00%	-60.63%	-35.69%
Sqlite					
libc	-66.72%	-66.50%	-58.10%	-67.51%	-57.36%
libz	-26.33%	-45.14%	-38.85%	-27.39%	N/A
libreadline	-1.61%	-5.19%	-7.16%	-1.84%	N/A
libcernuses	-78.91%	-88.89%	-69.31%	-79.81%	N/A
Openssh					
libc	-58.93%	-65.59%	-48.70%	-58.71%	-46.18%
libcrypto	-16.34%	-14.22%	-9.09%	-16.51%	N/A
libz	-27.93%	-44.57%	-40.65%	-29.24%	N/A
Vsftpd					
libc	-63.97%	-72.12%	-54.96%	-63.52%	-55.46%
libcrypto	-19.34%	-15.86%	-7.59%	-19.36%	N/A
libssl	-15.22%	-7.75%	-7.47%	-14.99%	N/A
Vsftpd(in-depth)					
libc	-63.97%	-72.12%	-54.96%	-63.52%	-55.46%
libcrypto	-44.41%	-38.14%	-34.08%	-45.06%	N/A
libssl	-37.94%	-24.32%	-35.90%	-39.61%	N/A
Apline(In-depth)					
libc	-44.09%	-54.57%	-36.50%	-42.91%	-32.41%
libcrypto	-44.41%	-38.14%	-34.08%	-45.06%	N/A
libssl	-37.94%	-24.32%	-35.90%	-39.61%	N/A

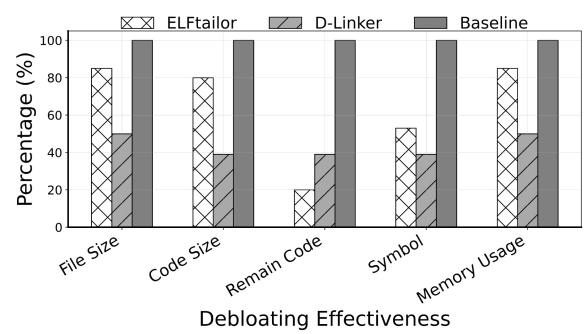

Fig. 6. Comparison with ELFtailor, remain code means code not debloated and overwritten in shared libraries, baseline is the original musllibc.

Comparison With ELFtailor: To validate the advantages of D-Linker in terms of disk space and memory savings, we compare it with ELFtailor [13], a prior approach that also targets binary scenarios and utilizes dynamic and static analysis. As illustrated in Fig. 6, we utilized D-Linker and ELFtailor to debloat shared library (not stripped because of symbol number analysis) of coreutils, with the test cases derived from the coreutils test suite. The comparison encompasses four aspects: 1) total file size; 2) code section size; 3) unused code size; and 4) memory usage.

From the experimental results, it can be observed that due to ELFtailor's reliance on binary rewriting, it encounters limitations related to 4KB block size constraints and an inability to eliminate data sections. This granted D-Linker a significant advantage above 30% over ELFtailor in terms of file size

TABLE VI
COMPARING WITH PIROR WORKS

	Piece-wise [6]	ELFtailor [13]	BlankIt [14]	μTrimmer [18]	D-Linker (this study)
No Source Code Needed	No	Yes	No	Yes	Yes(need object files)
No Runtime Support	No	Yes	No	Yes	Yes
Debloating Granularity	Function	Function	Function	Code Block	Object file
Load Time Slowdown	20X	0%	10X	0%	0%
Runtime Slowdown	0%	0%	18X	0%	0%
Gadgets Reduce	70-80%	50-60%	90%	50-60%	50-60%
Data Debloating	No	No	No	No	Yes

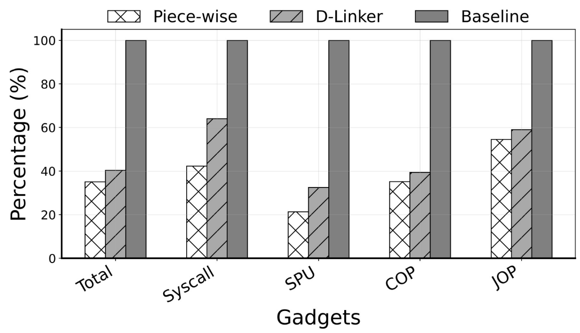

Fig. 7. Reduced gadgets comparison with piecewise, baseline is the original musllibc.

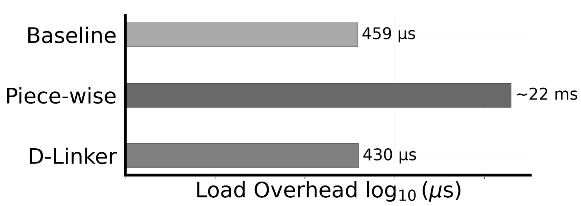

Fig. 8. Load time comparison with piecewise, baseline is the original musllibc.

reduction. Consequently, ELFtailor's advantage in code size reduction was not as prominent when compared to D-Linker. However, ELFtailor's finer-grained binary analysis enabled it to replace unused code with NOP operations, offering advantages above 10% over D-Linker in terms of unused code elimination. In terms of memory usage, D-Linker's file size reduction can be effectively reflected in memory utilization, leveraging the existing Linux memory management mechanisms. Additionally, the debloating of the data section translates to significant memory savings in multiprocess scenarios, surpassing the capabilities of ELFtailor and other related works.

Comparison With Piecewise: To evaluate the security performance of D-Linker, we compared its effectiveness in reducing gadgets with that of piecewise [6]. Piecewise optimizes the compilation process at the source code level during linking, thereby minimizing the amount of unused code loaded into memory and enhancing security benefits. As illustrated in Fig. 7, our analysis of various gadgets showed that although piecewise, due to its source-level optimizations, had certain advantages above 5% over D-Linker, and the requirement for source code and the additional costs incurred during load time (as shown in Fig. 8) make D-Linker highly attractive for security optimizations in embedded scenarios.

Comparison With Other Works: Finally, we compared existing shared library pruning tools and evaluated their features against those of D-Linker. As shown in Table VI, we assessed each tool based on several criteria: requirement for source code, need for additional runtime mechanism support, debloating granularity, introduction of runtime overhead, security evaluation, and ability to debloat data sections. Approaches we compared included piecewise [6] and ELFtailor [13], as previously mentioned, along with BlankIt [14] and μTrimmer [18]. We observed that source-based optimizations by piecewise and BlankIt offer apparent advantages in code debloating, reflected in security assessments. However, these come at the cost of increased runtime overhead. Additionally, ELFtailor, while effective, is less efficient than D-Linker in reducing shared library size due to the overhead introduced by binary-level rewriting, as previously discussed. Both ELFtailor and D-Linker employ dynamic analysis during the debloating process, whereas μTrimmer avoids dynamic analysis but is limited to the MIPS instruction set. Among all the tools evaluated, only D-Linker effectively debloats data sections. Based on the above evaluations, we find that D-Linker's object-file-level debloating offers several advantages: reduced disk and memory usage, elimination of the need for a compilation process during linking, no introduction of runtime overhead, and significant security benefits. These advantages make D-Linker more suitable for embedded scenarios compared to prior works.

VII. DISCUSSION

In this section, we discuss D-Linker's effectiveness, limitations, and future works.

Analysis of D-Linker's Effectiveness: The advantage of D-Linker in terms of debloating, compared to other works, mainly stems from its object file-based debloating granularity. This approach ensures that many unnecessary sections beyond the code segments, such as relocation information, symbol information, and data section, are omitted from the binary. This advantage becomes particularly evident in in-depth debloating scenarios. However, the number of object files that can be debloated is heavily dependent on the interdependencies among object files within shared libraries. For instance, in musllibc, most functions correspond to individual object files, enabling D-Linker to achieve nearly function-level debloating granularity, resulting in better outcomes. This distinction accounts for the variation in debloating effectiveness across different shared libraries and underscores why D-Linker, with its object file-based debloating approach, achieves superior reduction in size compared to previous works.

D-Linker's Limitations: D-Linker debloats shared library by object-level granularity, which causes more reduction effectiveness than other tools [13], [17], [18] that only debloat the code section. However, D-Linker still has two limitations. First, due to the poor code design or other reasons, there are always software or shared libraries with high coupling between code modules, resulting in a tight dependency between the object files in the shared libraries, and more symbols are contained in the same object file, which can affect the effectiveness of D-Linker. Anyhow, for this case, it is still possible to use other binary-level debloating tools to debloat the shared library after being debloated by D-Linker, which is in principle better than any binary-level debloating tool. Second, as for the in-depth debloating, the tradeoff for its superior debloating effect over static linking is that it can only guarantee effectiveness within the scope of the test cases.

Future Work: D-Linker only relinks the object files without making further binary-level changes and does not use the reference information of applications and libraries completely. In future work, further analysis of dependencies and unused code between object files during the debloating process can reduce useless dependencies between objects by wiping out symbols, binary rewriting, etc., which will further improve the debloating effectiveness.

VIII. Conclusion

In this article, we propose a object-level debloating approach to enhance debloating effectiveness while ensuring flexibility of usage. The object-level debloating offers higher flexibility compared to the source-code-level debloating, and is more precise compared to the binary-level debloating, resulting in a better-debloating effectiveness. Based on this approach, we also propose D-Linker, a tool that debloats shared libraries by reducing both code and data sections at object-level without recompilation. D-Linker effectively extract object files from the corresponding static library of shared libraries, thereby eliminating the need of compilation and source code. We have applied D-Linker to debloat shared libraries of vsftpd and our approach achieves an average reduction in size of 27.6% for shared libraries, a maximum reduction of vsftpd in size of 44.9% when specific features are identified. The results indicate that D-Linker improves debloating effectiveness by approximately 30% compared to binary-level shared library debloating. Additionally, in terms of security, it incurs a 5% decrease in code gadgets reduction compared to source-code-level shared library debloating.

References

[1] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*. Cambridge, MA, USA: MIT Press, 2000.

[2] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Commun. ACM*, vol. 19, no. 5, pp. 266–272, 1976.

[3] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of software in distributed embedded automotive systems," in *Proc. 4th ACM Int. Conf. Embed. Softw.*, 2004, pp. 203–210.

[4] J. Larus, "Spending moore's dividend," *Commun. ACM*, vol. 52, no. 5, pp. 62–69, 2009.

[5] "Shared library loading." Accessed: Aug. 30, 2022. [Online]. Available: https://os.educg.net/2022CSCC?op=6

[6] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *Proc. 27th USENIX Security Symp. (USENIX Security)*, 2018, pp. 869–886. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[7] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *Proc. 35th Annu. Design Autom. Conf.*, 1998, pp. 516–521.

[8] "Alpine official image." DockerHub. 2022. Accessed: Aug. 30, 2022 from [Online]. Available: https://hub.docker.com/_/alpine

[9] J. Lee, J. Park, and S. Hong, "Memory footprint reduction with quasi-static shared libraries in MMU-less embedded systems," in *Proc. 12th IEEE Real-Time Embed. Technol. Appl. Symp. (RTAS)*, 2006, pp. 24–36.

[10] "readahead(2)—Linux manual page." 2022. Accessed: Aug. 30, 2022. [Online]. Available: https://www.man7.org/linux/man-pages/man2/readahead.2.html

[11] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[12] "BlueKeep vulnerability (CVE-2019-0708): National vulnerability database." NIST. 2019. [Online]. https://nvd.nist.gov/vuln/detail/CVE-2019-0708

[13] A. Ziegler, J. Geus, B. Heinloth, T. Hönig, and D. Lohmann, "Honey, I shrunk the ELFs: Lightweight binary tailoring of shared libraries," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, pp. 1–23, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358222

[14] C. Porter, G. Mururu, P. Barua, and S. Pande, "Blankit library debloating: Getting what you want instead of cutting what you don't," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2020, pp. 164–180.

[15] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "TRIMMER: Application specialization for code debloating," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 329–339.

[16] B. Shteinfeld, "LibFilter: Debloating dynamically-linked libraries through binary recompilation," Undergraduate Honors Thesis, Brown Univ., Providence, RI, USA, 2019.

[17] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proc. 35th Annu. Comput. Security Appl. Conf.*, 2019, pp. 70–83.

[18] H. Zhang, M. Ren, Y. Lei, and J. Ming, "One size does not fit all: Security hardening of MIPS embedded systems via static binary debloating for shared libraries," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2022, pp. 255–270.

[19] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proc. 12th Eur. Workshop Syst. Security*, 2019, pp. 1–6.

[20] I. Grammatech. "Binary reduce." 2022. [Online]. Available: https://grammatech.github.io/prj/binary-reduce/

[21] A. Turcotte, E. Arteca, A. Mishra, S. Alimadadi, and F. Tip, "Stubbifier: Debloating dynamic server-side JavaScript applications," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 161, 2022.

[22] A. Altinay et al., "BinRec: Dynamic binary lifting and recompilation," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.

[23] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2020, pp. 151–163.

[24] X. Meng and W. Liu, "Incremental CFG patching for binary rewriting," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, pp. 1020–1033.

[25] "Executable and linkable format." 2022. Accessed: Aug. 8, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[26] C. Mulliner and M. Neugschwandtner, "Breaking payloads with runtime code stripping and image freezing," presented at Proc. Black Hat USA, 2015.

[27] A. Q. Nguyen. "Capstone: The ultimate disassembler." 2019. Accessed : Jul. 30, 2019. [Online]. Available: https://www.capstone-engine.org/

[28] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad, "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps," in *Proc. Linux Symp.*, 2007, pp. 215–224.

[29] "nm." 2022. Accessed: Aug. 30, 2022. [Online]. Available: https://linux.die.net/man/1/nm

[30] "Musl libc." Accessed: Aug. 30, 2022.[Online]. Available: https://www.musl.libc.org/