Shared Libraries Bloating in 3D Graphics Software

Giacomo Benedetti[1], Luca Caviglione[1,*] and Michela Spagnuolo[1]

[1]*Institute of Applied Mathematics and Information Technologies, Genova, Italy*

Abstract

Many modern frameworks take advantage of three-dimensional (3D) graphics to create and visualize digital representations of complex objects. Owing to the importance of 3D software, being able to assess and reduce its attack surface are urgent needs, for instance to prevent data leakages. To this aim, a major approach takes advantage of debloating, i.e., removing unnecessary features, code, and functionalities that may increase the resource footprint or provide entry points to attackers. Despite their effectiveness, debloating techniques have not been systematically investigated for the case of 3D-capable applications. Therefore, this paper is a prime attempt to fill this gap and it analyzes the level of bloating that may influence the security of 3D software. Results demonstrate that there is room for improving the security posture of graphic applications and for deploying an attack surface reduction strategy.

Keywords

Bloating, Attack Surface Reduction, Software Security, 3D graphics

1. Introduction

Three-dimensional (3D) graphics is a fundamental building block of many modern software ecosystems since it enables the creation, manipulation, and visualization of the digital representation of physical objects. For instance, high-quality 3D software is essential in urban intelligence applications to prepare accurate renditions of infrastructures as well as to simulate large-scale scenarios [1, 2]. Owing to its importance, ensuring the quality of software at the basis of 3D applications is a major need. Specifically, software *reproducibility* is critical to support collaborative research and development, especially to compare and validate 3D algorithms against other approaches [3]. Moreover, deploying lightweight 3D software is pivotal in resource-constrained environments, such as mobile or edge nodes, where storage and processing capabilities are limited [4]. An aspect often neglected concerns the *security* of software components taking advantage of some form of 3D modeling or visualization. In fact, vulnerabilities in tools, libraries and applications using 3D graphics can lead to information leakages, unauthorized access, or malicious manipulation of models, with potentially severe consequences in critical sectors, such as the healthcare and aerospace. To make a recent example, CVE-2024-8374[1] targeting the UltiMaker Cura 3D printing software allowed to inject arbitrary code by exploiting a Python function, which receives data without proper sanitization mechanisms.

A promising strategy to deliver optimized, secure, and high-quality 3D applications is to pursue some form of software *debloating*. Software debloating is the process of removing unnecessary code or functionalities to improve performance, limit resource consumption, and ultimately enhance security [5]. As software systems grow more complex, they often accumulate redundant or unused components, which can increase the attack surface and create vulnerabilities. Debloating then tries to streamline applications, resulting in leaner, more efficient software that is easier to maintain and less susceptible to performance degradations or *exploitable weaknesses*, which if not properly handled may lead to CVEs or enable attack campaigns. Even if a relevant corpus of works on software debloating has started to emerge, there is no prior research specifically devoted to streamline 3D libraries or frameworks (see,

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT

*Corresponding author.

giacomo.benedetti@ge.imati.cnr.it (G. Benedetti); luca.caviglione@ge.imati.cnr.it (L. Caviglione); michela.spagnuolo.imati.cnr.it (M. Spagnuolo)

0000-0003-2609-6787 (G. Benedetti); 0000-0001-6466-3354 (L. Caviglione); 000-0002-5682-6990 (M. Spagnuolo)

[1]https://www.cve.org/CVERecord?id=CVE-2024-8374

e.g., [6] for a recent review on the topic). Additionally, security of 3D-capable ecosystems has been mainly addressed in terms of traceability, copyright management, and anti-tampering of digital artifacts, rather than in terms of understanding security and privacy implications of graphic components [7]. As an example, [8] investigates how to secure 3D models by embedding "fingerprints" into a tamper-resistant blockchain structure, whereas [9] showcases self-checking and sealing mechanisms for the run-time verification of the authenticity of 3D files and digital assets.

Therefore, this work tries to fill this gap by offering an initial analysis of the relation between bloating and security properties of software at the basis of modern 3D-capable frameworks. Specifically, it proposes a methodological approach for investigating the presence of unnecessary code that can be safely eliminated without affecting the behavior of the program across all the intended use cases [10]. To evaluate the effectiveness of the presented debloating strategies, this prime investigation considers two different 3D settings, i.e., libraries and tools developed for research purposes as well as popular applications. Our analysis reveals how direct dependencies dominate the bloating profile, with graphics-related libraries contributing disproportionately to unnecessary code. Building on these findings, we outline some practical strategies that developers can adopt to minimize dependencies and reduce the exposure to security hazards.

Summing up, the contribution of this work are: *i*) the definition of a methodology to pursue debloating in software components devoted to 3D applications, *ii*) a prime analysis of the major dependencies that can inflate a software artifact, and *iii*) the presentation of some best-practices that can improve the overall security posture of 3D frameworks.

The rest of the paper is structured as follows. Section 2 reviews past works dealing with performance and security of 3D software, while Section 3 provides background details on the major causes of bloating and how they can be mitigated. Section 4 introduces the methodology used to investigate 3D software, Section 5 presents results on the level of bloating, and Section 6 discusses the resulting security posture with emphasis on the attack surface reduction practice. Lastly, Section 7 concludes the paper and hints at possible future research directions.

2. Related Works

As hinted, pursuing some form of debloating and investigating security properties are now common tasks for a vast array of tools and software artifacts [10]. However, only few works consider graphic-intensive applications and 3D software, often in a byproduct manner. For instance, [11] introduces a benchmarking framework for comparing four popular software debloating techniques. To this aim, the authors consider several applications, including GraphicsMagik, i.e., a collection of tools and libraries for processing 2D raster images. However, specific behaviors of graphic software are not addressed, and GraphicMagik has been solely included to increase the heterogeneity of the considered dataset. The research in [12] focuses on pruning unneeded functionalities of the Nginx web server. The analysis revealed that image manipulation functionalities provided via `libgd.so` are always included, even if Nginx does not allow to manipulate graphic contents by default. Thus, the authors propose to drop support for the library and reintroduce it only when actually needed. Even if partially unrelated, the work in [13] exhibits several contact points with 3D graphics libraries. Specifically, it investigates the debloating of a solver used to implement computational fluid dynamics applications.

Many 3D libraries and graphic-intensive applications need to interact with a vast amount of information, for instance LiDAR point clouds or geometric models of urban areas [14]. In this vein, an interesting approach concerns "data lineage", i.e., methods to identify unnecessary data instead of code. Even if the work in [15] does not focus on graphic-related software, it offers many insights on containerized applications, which are often used to distribute or encapsulate 3D frameworks. Specifically, authors discovered that the majority of applications tend to access all the provided data files, but they often use very small portions of the whole content.

As regards security of 3D software, some works are starting to emerge, even if primarily targeting specific OS/development ecosystems or considering low-level implementation aspects. A major example

is the work in [16], which investigates the security posture of debloated Android applications. The work also addresses the various 3D engines deployed for implementing advanced animations and transitions. Despite leaner applications have a smaller size and a faster performance, the resulting level of security is not always improved. In fact, debloat often requires to deploy APIs for collecting the behavior of the software for its assessment, logging libraries to support debug operations that may leak personal data, and lightweight versions of core routines with a less-defined attack surface. Concerning security aspects of code devoted to implement 3D graphic functionalities, [17] deals with threats that can abuse the device drivers of Windows. In more detail, graphics drivers offer several functionalities that can be abused with malformed input data or forced to cause buffer overflows. Another possible entry point for an attacker is how graphic data is handled. In this case, graphic buffers could be abused and requires a suitable isolation to prevent percolation of information among different rendering contexts [18]. Due to the computationally-intensive nature of several applications, security of 3D graphics libraries and frameworks is often tightly coupled with security of GPUs. In this perspective, a major issue is that GPUs do not erase memory (either shared or global) allowing an adversary to leak sensitive information [19]. When local protection of data is unfeasible, a viable approach is to offload the computation. In this case, rendering operations can be performed on a remote, secure system, which guarantees that 3D geometry information has not been accessed by an unauthorized party or extracted by a threat actor [20].

Thus, at the best of our knowledge, our work is the first research attempt focusing on debloating of 3D graphics libraries with the aim of understanding security improvements that can be achieved via an attack surface reduction approach.

3. Background

This section provides background concepts of the approach proposed for investigating security implications of bloated 3D software. Specifically, Section 3.1 introduces concepts related to dependencies and shared libraries, while Section 3.2 presents debloating techniques.

3.1. Dependencies and Shared Libraries

While the majority of modern applications contains a non-negligible amount of code, many core functionalities are not directly implemented by the final developer. In fact, to accelerate the development cycle and support complex requirements, applications rely on *dependencies*, i.e., external components such as libraries or frameworks. Usually, dependencies may provide general-purpose utilities (e.g., mathematical functions, cryptographic primitives, and input/output handlers) or domain-specific features (e.g., geometric processing, rendering operations, and GPU interaction in 3D applications). While dependencies improve productivity, they also increase the complexity of software products and reduce transparency over the final code base. Moreover, *direct depedendencies* of the software can also rely on dependencies, taking the name of *transitive dependencies*. The structure created by this linkage among dependencies is usually denoted as the *software supply chain* [21].

In compiled languages such as C and C++, dependencies are typically implemented through *shared libraries*. A shared library is a standalone binary module that exposes a set of functions or symbols, which can be dynamically linked by one or more executables at runtime. This architectural blueprint offers several advantages. For instance, relying upon an "external" shared block of compiled code reduces duplication across applications, simplifies updates, and enables modular software design. However, to be of general utility as well as to support a wide-range of applications and use cases, shared libraries often implement a relevant amount of functionalities. In general, the average library contain a greater amount of functions compared to those required by the single, served program [22]. As a result of this behavior, executables may load into memory large portions of unused code, a phenomenon known as *dependency bloat*.

From a security perspective, unused code in shared libraries accounts for an increased attack surface. For instance, despite being never invoked, a linked function may be plagued with a weakness potentially leading to an exploitable vulnerability. An application not using but linking a vulnerable function

may be then exploited by an attacker through techniques such as Return-Oriented Programming (ROP) or symbol interposition. Another important security hazard is rooted within how dependencies are managed by C and C++ ecosystems. In fact, both languages rely upon an "ad-hoc" blueprint for the overall development/building process. Unlike languages with centralized registries (e.g., npm for JavaScript or Cargo for Rust), in C/C++ dependencies are usually manually specified within the build configuration. This makes it harder to track versions, enforce updates, and verify trustworthiness.

Like many other complex applications, also 3D-capable software requires a suitable dependency management and the ability of precisely tracking how shared library are used [23]. Therefore, identifying relevant information on used functionalities is critical, especially since many libraries rely on poorly-automatable building procedures. In fact, the most advanced 3D applications often take advantage of large, multi-purpose libraries, e.g., rendering engines, GUI toolkits, or numerical computation packages. As a consequence, the developer exploiting such a multifaceted ecosystem of functionalities introduces thousands of symbols, many of which remain potentially unused. In general, each symbol corresponds to a function and hence a pattern of code that may or may not be used at runtime. Unfortunately, the resulting bloat does not only impact performance and portability of the overall software. Indeed, the burden of unneeded code increases the exposure of the whole software artifact to security vulnerabilities hidden in code paths that were never intended to be part of the application. The mitigation of such a poor practice is especially important in resource-constrained environments like mobile and edge devices, which may be used as entry points to more complex environments or contain sensitive data, e.g., measurements from IoT sensors or medical devices.

3.2. Debloating

As said, software debloating refers to the systematic removal of unnecessary code, features, or dependencies from an application with the objective of reducing its footprint and improving the software quality. While originally proposed as a mean of addressing performance and maintainability requirements in large-scale systems, debloating has more recently emerged as a promising strategy for enhancing security. By eliminating unused or overlapped functionalities, debloating effectively decreases the number of potential attack vectors, thereby reducing the overall attack surface [10].

A wide range of debloating techniques have been introduced in the literature. *Static analysis-based* approaches rely on methods such as symbol enumeration and dead-code elimination to identify code regions that can be safely removed [10]. In contrast, *dynamic analysis-based* approaches monitor program execution in order to determine which portions of dependency code are exercised under representative workloads [24]. When source code is available, *compiler-assisted* techniques can also be employed to exclude unnecessary elements during compilation and linking [25]. In the specific context of *shared libraries*, two strategies have been shown to be particularly effective. *Symbol trimming* entails the identification and removal of unused exported symbols (e.g., functions) from library dependencies. *Partial linking* involves the extraction of only those object files that correspond to library components actually utilized by the application, followed by relinking against this reduced library. Both techniques allow for the specialization of shared libraries to the needs of a given application, thereby achieving reductions in code size and attack surface without sacrificing required functionalities [10, 26].

To pursue some form of optimization, a fundamental aspect concerns with the quantification of the bloating caused by the various dependencies. The enumeration phase of debloating is an enabler of both an efficient remediation as well as a constructive empirical evaluation of the current status of the software. To this aim, the aforementioned techniques are the main building blocks for the debloating process, since they allow to identify which parts of code and dependencies are not necessary. According to the specific use case, they can be used standalone or combined (e.g., static approaches supported by information provided at compile-time). This guarantees the retrieval of a comprehensive list of functionalities imported through a library that are actually used by the application that needs optimization.

Table 1
List of the C/C++ applications considered in this work.

Software / Library	GitHub Repository	Description
CDT - Constrained Delaunay Tetrahedrization	marcoattene/CDT	3D constrained Delaunay tetrahedralization
Fast And Robust Mesh Arrangements	gcherchi/-FastAndRobustMeshArrangements	Fast mesh arrangement and intersection operations
HexBox	cg3hci/HexBox	Hexahedral mesh generation and manipulation
PEMesh	DanielaCabiddu/PEMesh.2	Polygonal element mesh generation
Interactive And Robust Mesh Booleans	gcherchi/-InteractiveAndRobustMeshBooleans	Robust boolean operations on triangle meshes
OOCTriTile	DanielaCabiddu/OOCTriTile	Out-of-core tiled triangle mesh processing
ImatiSTL	qnzhou/ImatiSTL	STL mesh repair and processing
SemantisedTriangleMesh	andreasscalas/SemantisedTriangleMesh	Semantic annotation of triangle meshes
VolumeMesher	MarcoAttene/VolumeMesher	Volumetric mesh generation from surface meshes
MeshFix	MarcoAttene/MeshFix-V2.1	Typical raw digitized mesh models flaws correction
MeshLab	cnr-isti-vclab/meshlab	Processing and editing of unstructured large 3D triangular meshes
PCL	PointCloudLibrary/pcl	2D/3D image and point cloud processing
FreeFEM++	FreeFem/FreeFem-sources	Finite element method solver and scripting
PyMeshFix	pyvista/pymeshfix	Python wrapper for mesh repair and fixing

4. Investigation Methodology

This section presents the approach used to assess the bloating of 3D graphics software. Specifically, Section 4.1 describes the shortlisting process of applications and Section 4.2 elaborates on the design of the methodology for quantifying the amount of bloating.

4.1. Selection Process

Understanding the characteristics of the software under investigation is fundamental to design an effective debloating methodology, especially to enforce attack surface reduction practices. Our analysis mainly focuses on the 3D libraries providing a foundation to the research ecosystem of the Institute for Applied Mathematics and Information Technologies (IMATI) of the National Research Council of Italy. In fact, during the years, IMATI has developed numerous tools for the manipulation of 3D meshes and for geometric processing tasks. Such tools are commonly used to implement a wide-range of functionalities and have been incorporated in the research pipelines of major research groups advancing in 3D graphics. The resulting corpus of code encompasses both command-line utilities and applications endowed with GUIs, providing an ideal case study that captures the diversity of 3D software architectures commonly found in research and industrial contexts. The applications collection process involved cross-referencing information from IMATI research scientists and applying systematic filtering criteria to ensure the reliability and relevance of our dataset.

In more detail, software components were selected according to three key criteria:

- **License**: we focused exclusively on open-source libraries hosted on public repositories such as GitHub and GitLab. This ensures reproducibility and enables the broader community to validate our findings and extend the proposed methodology to additional software packages.
- **Maintenance**: we shortlisted only libraries with a recent development activity, specifically those receiving commits within the last two years. This ensures that our analysis reflects current

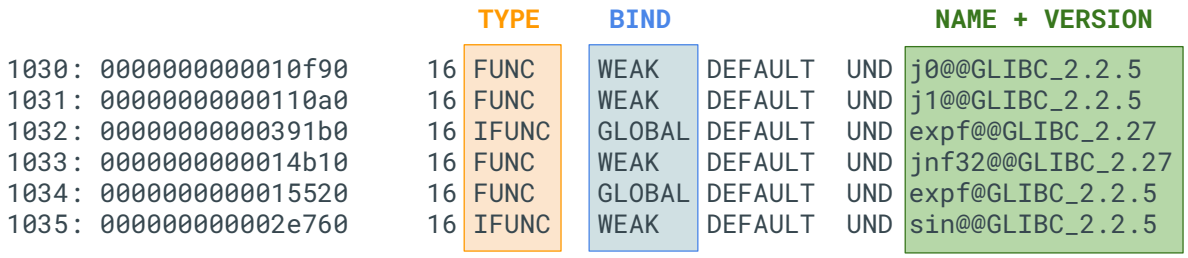

Figure 1: Excerpt of the `readelf` command output. It reports information necessary to the identification of libraries required by a binary.

development practices and dependency management strategies, rather than obsolete software patterns that may not represent anymore modern 3D development workflows.

- **Documentation**: we identified libraries endowed with a comprehensive documentation, particularly build instructions for C-like languages. Unlike higher-level languages with standardized package managers, C/C++ projects often require specific configuration procedures. In general, missing documentation creates barriers for reproducibility and prevents the systematic analysis of the build process as well as of the resulting dependencies.

The initial set of candidate libraries underwent an additional validation step based on the successful compilation. In essence, each library was cloned from its main branch and built by following the documentation provided in the respective repository. Libraries that failed to build were excluded from the analysis, as build failures prevent accurate dependency extraction and bloating assessment.

To make our findings more general and enhance the validity of the proposed approach, we also extended our investigation beyond IMATI-developed tools. Specifically, we investigated four software packages widely-used in the 3D research community: `MeshLab`, `PCL`, `FreeFem++` and `PyMeshfix`. In more detail, `MeshLab` is a popular CLI-based software for processing and editing unstructured large 3D triangular meshes (it has more than $5,000$ stars and 70 contributors on GitHub), while `PCL` is one of the most used tool for processing point clouds and 2D/3D images (it has an active development community of more than 500 contributors). Besides, `FreeFem++` represents a state-of-the-art software for finite element methods and operates as a command-line tool, providing insight into the dependency patterns of computational mathematics software. Lastly, `PyMeshfix` is a Python port of the `Meshfix` tool developed by IMATI, thus offering a valuable comparison point for quantifying how programming language choices influence software bloating patterns.

Table 1 provides the final set of libraries considered in our analysis, specifying the software name, the repository handler where the application is available, the primary programming language, and a brief description extracted from the various `README.md` files. This collection captures various aspects of 3D software development, from low-level geometric processing to high-level visualization frameworks. As it can be seen, the resulting dataset predominantly consist of C/C++ software, requiring the design of a bloating analysis approach that targets this specific ecosystems. The only exception is `PyMeshfix` containing Python code that "wraps" the core functionalities implemented in C/C++. As a consequence, about the majority of tools and major functionalities do not rely on package managers making the analysis more laborious and error-prone.

4.2. Bloating Investigation

To address the challenge of performing a prime bloat analysis of 3D-software, we developed a three-step methodology. This allows to systematically identify unused functionalities, while accounting for the complexities of dynamic linking in Unix-like systems.

Shared Library Functions Extraction The first step establishes a comprehensive inventory of all functions available through the dependency chain of the software. To this aim, we extract two

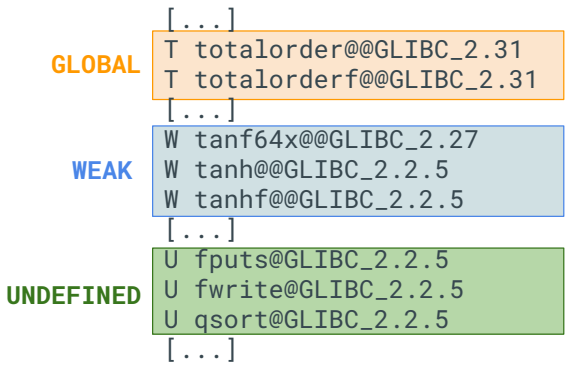

Figure 2: Output example for the `nm` program.

critical pieces of information: the complete list of shared libraries required by each program and the collection of functions exported by those libraries. Therefore, we used the `ldd` utility to identify all shared libraries linked to the target 3D software as well as their respective file system locations. For each identified shared library, we then employed the `readelf` tool to parse the ELF sections and extract symbolic information.[2] Figure 1 shows an example output of the `readelf` tool launched against the CDT software package. Upon the symbol list has been obtained, we followed the methodology presented in [27] to identify the *three* most important information for characterizing an exported function. As a result of this well-established approach, for all the functions contained in the considered software in Table 1, we obtained the following information:

- **type**: FUNC or IFUNC denoting the runtime linking procedure;
- **binding**: GLOBAL or WEAK indicating the precedence order during the symbol resolution phase;
- **function name** and **version**: with @@ denoting default versions and @ indicating optional versions.

This systematic extraction creates a complete map of the functional interface provided by each 3D software, establishing the baseline of our experimental evaluation.

Undefined Functions Identification The second step is aimed at determining which functions a given program actually requires from its shared libraries during execution. To collect such information, we take advantage of the distinction between static and dynamic strategies implemented by C/C++ compilers. We point out that for the case of `PyMeshFix`, this approach is still valid. In fact, despite the presence of a Python wrapper, the "core" responsible for delivering all the 3D-based functionalities is still completely written in C/C++. To identify undefined functions, we analyzed the dynamic section of ELF binaries by using the `nm` program, which returns a categorized list of symbols. Figure 2 showcases an example output of the symbols required by the CDT software tool. In this case, each function is marked according to its binding status: T for Global, W for Weak, and U for Undefined. For the purpose of quantifying the bloating, only undefined functions (i.e., U) play a role. In fact, they are provided through additional code that the main program expects to be resolved at runtime. As a result, functions U constitute the actual set of requirements of the software.

Unused Functions Computation The final step quantifies the precise extent of bloating. This requires to evaluate the difference between the functions exported by a shared library at compile time (i.e., T and W) and those potentially used at runtime (i.e., U). Before performing this step, we implemented a sanitization process to ensure an accurate matching between the function lists obtained via `readelf` and `nm`. Such an additional step prevents from considering duplicated functions, i.e., different versions

[2] In Unix-like architectures, e.g., Linux, a shared library is essentially a shared object in ELF format.

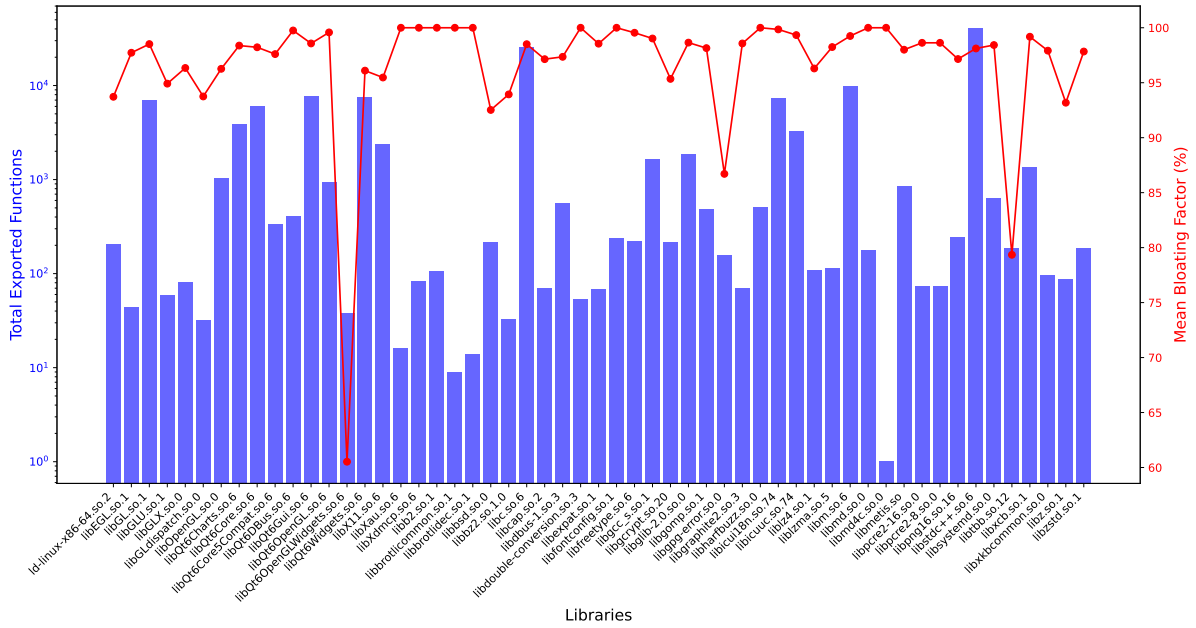

Figure 3: Bloating caused by shared libraries across the considered dataset.

of a given function. When both default and optional versions of the same function are available, we then prioritize the optional version over the default one, as this reflects the actual resolution behavior of the dynamic linker. As a result, for all the software presented in Table 1, we obtained the data needed to quantify the so-called "Type I bloating". In other words, the final outcome is the list of functions that are loaded into the address space of the program but never invoked during the execution. We point out that this work focuses on direct dependencies and extending the proposed methodology to target transitive dependencies is part of our ongoing research.

5. Quantitative Results

This section presents the results obtained by applying our methodology on the shortlisted 3D software. In more detail, Section 5.1 discusses the general bloating behavior, whereas Section 5.2 showcases the impact of dependencies. All the relevant data, containerized test environments, and supporting Python scripts are publicly available in the repository[3] linked with this paper.

5.1. General Bloating Analysis

To have a precise estimate of the general bloating of considered 3D software, we introduced and quantified a *bloating factor* indicator. This metric is in the $[0, 1]$ range and has been designed to capture the real usage of functions. Specifically, it provides a rough indicator of efficiency as the metric considers the unused functions with respect to the number of functions exported by a library. Hence, higher values indicate a reduced bloat, since all the imported functions are expected to be used. Figure 3 showcases the results. As it can been seen, the bloating factor is generally stable across all the shared libraries used by the considered 3D software, i.e., it does not vary more than the 5.6% respect to the average value. However, there are few exceptions. The most notable ones are `libQt6OpenGLWidgets.so.6` and `libtbb.so.12`. This is an expected behavior, since they are designed to provide many functionalities, which are seldom completely used/needed by a single application. Another exception is `libgpg-error.so.0` since it represents a common dependency for

[3] https://github.com/giacomobenedetti/libraries-debloating-proj

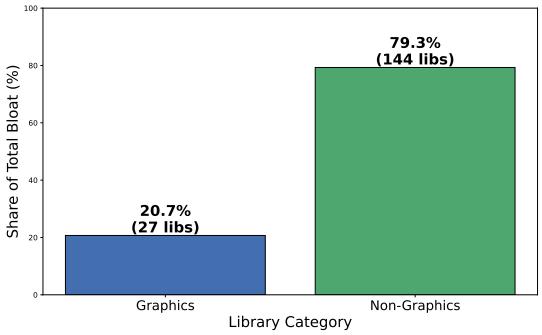

Figure 4: Classification per type of shared library.

all the GnuPG components. Hence, it has been designed to serve a broad set of use cases leading to an increased amount of bloating.

The "stability" of the bloating factor could suggest that bloating should be tackled without distinction by the purpose of the dependency. That is, it does not matter whether the shared library deals with standard functionalities provided by the underlying OS or for implementing complex and feature-rich GUIs. However, when clustering libraries according to their purpose/goal, certain patterns become more evident. In particular, graphics-related shared libraries account for a disproportionately large contribution to the overall bloating factor. Figure 4 reports how a given class of library contributes to the overall bloating factor. As shown, 144 general-purpose libraries contribute to the 79% of the total bloating. Instead, just 27 libraries offering graphic functionalities are responsible for 21% of the total bloating. Although no single shared library can be identified as the sole "responsible", this imbalance suggests a correlation between graphics-related libraries and higher bloating levels.

5.2. Dependency Analysis

Understanding whether dependencies causing the bloating are direct or transitive is necessary to design and enforce proper debloating strategies. To this aim, we recursively applied the techniques presented in Section 4.2 also to transitive dependencies, i.e., dependencies of direct dependencies. As a result, we obtained data on the impact of transitive dependencies on the bloating caused to the overall dependency network. Figure 5 depicts the outcome of the investigation. In general, it can be observed a predominance of bloating caused by direct dependencies. This trend is largely influenced by the choice of using C/C++ as the main programming language. Most of the bloating is caused by utility libraries, such as those for standard input/output tasks. The latter contains several functionalities necessary to support a wide-range of operations across multiple domains. The only application where transitive dependencies constitute the primary source of bloating is `FreeFem++`. This 3D software demonstrates a bloating profile similar to `PEMesh`, which stems from the use of the `Qt` library. Another relevant result is that even a single library may cause a significant impact in terms of overall bloating. To make a notable example, `MeshLab` exhibits bloating percentages similar to those characterizing less complex software, e.g., `CDT`. This behavior can be ascribed to the fact that it only exploits few "core" shared libraries, such as `libpthread.so.0` for multithreaded programming and `libz.so.1` for compressing/decompressing data. Besides, the bloating of `PCL` stands out due to both direct and transitive dependencies. However, it must be considered that `PCL` is actually a suite of software, composed of 18 different executable artifacts. It is also worth mentioning that `PCL` handles many functionalities related to image processing and internal operations through custom shared libraries, e.g., `libpcl_io_ply.so.1.15`. This design choice drastically reduces the level of bloating that would be otherwise much higher. Concerning `PyMeshFix`, we recall the it is the only application written in Python[4]. Yet, it essentially exhibits the

[4]We do not account for the bloating introduced by Python packages, since our focus is on shared libraries. For the sake of completeness, we note that `PyMeshFix` solely depends on the `numpy` package, which should therefore be considered part of the bloating.

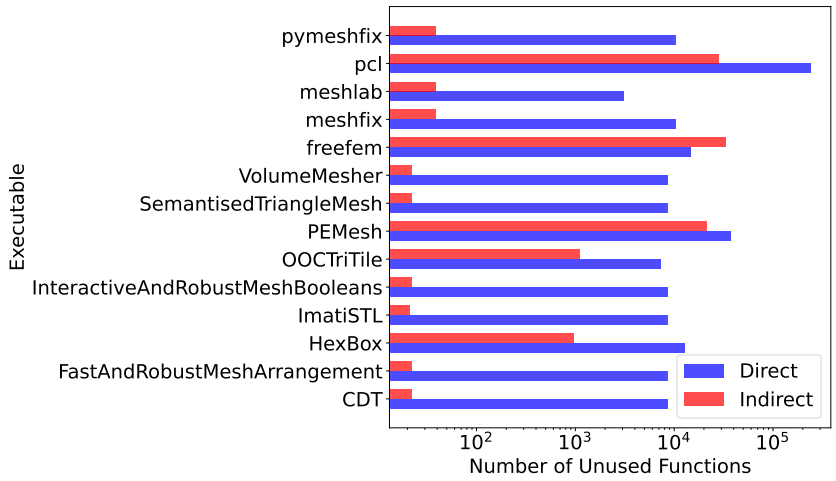

Figure 5: Classification per level of dependency.

same bloating profile as its C counterpart, i.e., `MeshFix`. This behavior can be ascribed to the fact that core functionalities of the software are implemented in C and the Python wrapper only plays a marginal role.

6. Security Analysis

As discussed, the software bloat increases complexity and the resource usage. Most importantly, a bloated software leads to a greater attack surface. A notable example is the offensive campaign carried out by leveraging the `XZ` compression utility. Specifically, on March 2024, CVE-2024-3094[5] disclosed the presence of a backdoor in the `liblzma` shared library. The CVE has been made possible by targeting the build process, i.e., some functions were modified by injecting malicious code. A modified version of `liblzma` can then be linked against a software to conduct a wide-range of attacks. In our analysis performed on the 3D software of Table 1, we identified the presence of the `liblzma` library in the transitive dependencies of `PEMesh`. We found that just three functions of the library were actually used. Thus, debloating the unused functions of `liblzma` would have been mitigated the attack surface characterizing the `XZ` utility ad would prevent the possibility of using `PEMesh` as a vector for installing a backdoor.

Another major source of bloating, which should be assessed in the vein of operating in an attack-surface-reduction flavor is the ubiquitous `Qt` library. In more detail, it has more than 200 active CVEs[6] characterized by a CVSS score ≥ 3. This underlines the importance of reducing the number of unused components in mission-critical 3D software to not endanger the overall security posture, see, e.g., the case of healthcare applications.

In this perspective, for the goal of improving the security posture of modern 3D ecosystems, two "defensive" strategies are relevant. The first concerns *pre-build* debloating, which is particularly suited for developers who wish to minimize external dependencies and avoid the uncontrolled growth of code. Section 6.1 presents some guidelines that should be used to prevent that bloating will impact on the security level of a 3D application. The second is *post-build* debloating, which is essential when integrating third-party or legacy software where the source code or dependency structure cannot be directly managed. Section 6.2 introduces tools that should be incorporated within the development process to reduce possible attack entry points against 3D software.

[5] https://www.cve.org/CVERecord?id=CVE-2024-3094
[6] https://www.cvedetails.com/vulnerability-list/vendor_id-6363/QT.html?page=1&cvssscoremin=3

6.1. Pre-build Debloating

Reducing the number of dependencies Dependency management is central to enforce pre-build debloating practices. Third-party libraries accelerate the development process but introduce risks ranging from accidental vulnerabilities to the intentional injection of malicious code. In general, many libraries implement far more functionalities than actually required, leaving in place large portions of unused, yet still exploitable code. Therefore, limiting dependencies directly reduces the attack surface. In languages such as C and C++, dependency tracking is often done in an ad-hoc manner. While build systems like `CMake` allow developers to specify dependencies (e.g., through `CMakeLists.txt`), C/C++ ecosystems lack a central, trusted registry. For this reason, a more secure generation of 3D software should consider other ecosystems, for instance, Cargo (Rust) or npm (JavaScript). Yet, if performance is a main constraint, Rust should be preferred, while JavaScript may be considered for portability across web-based frameworks. An important aspect to consider when pursuing pre-build debloating is the absence of a reference ecosystem. In fact, its absence complicates auditing, version pinning, and vulnerability tracking. By contrast, registry-based ecosystems provide explicit dependency manifests, which enhance transparency and reproducibility. They also align with regulatory requirements such as the Network and Information System - NIS2 directive of the European Commission, which emphasizes the need of having a suitable visibility across supply chains. Similarly, U.S. cybersecurity directives now mandate Software Bills of Materials for certain categories of software, underscoring that transparency in dependency chains is becoming a baseline requirement [28, 29]. Therefore, modern 3D graphics software and applications should adhere to such practices and regulations in order to be incorporated to the emerging global software supply chain without endangering its posture.

Avoid over-importing of dependencies Large dependencies are often introduced to implement trivial functionalities. A typical example is relying on a full cryptographic framework merely to compute checksums. This leads to importing thousands of lines of potentially vulnerable code and expands the attack surface of the application. Where feasible, developers should replace heavyweight dependencies with minimal, self-developed alternatives. Beyond security, this approach improves maintainability by reducing exposure to breaking updates in third-party code.

Using a common shared library for utilities When multiple 3D applications are developed within the same organization, maintaining an internal shared library for common utilities can reduce reliance on third-party packages. Many simple functionalities, e.g., string parsing, checksum computation, or logging, are often re-imported from large external libraries, even though they are already present across other in-house projects. By consolidating these recurring functions into a vetted internal library, organizations not only reduce redundant dependencies but also minimize the attack surface introduced by external code. Moreover, this approach preserves the efficiency of the development process while strengthening the control that can be enforced over widely-used components. For the case of the 3D software developed by IMATI researchers, all the software rely on a subset of 160 functions implemented in the `libc` and on a subset of 34 functions implemented in the `libm`. Such a behavior suggests that the needed subsets should be transferred to a self-maintained shared library, hence reducing the bloating caused by larger codebases.

6.2. Post-build Debloating

Post-build debloating is required when dependencies cannot be fully controlled during the development phase, or when external binaries are "merged" during the build process. As an example, this can happen when the 3D software has to be incorporated within complex services, such as those at the basis of urban intelligence applications [30]. In such cases, binary-level trimming can restrict an application to only the symbols and functions it actively uses. One approach involves re-linking executables against stripped versions of libraries, ensuring that unnecessary code is excluded. Tools such as `libfilter` [31] and Razor [13] exemplify this strategy. In essence, they allow to analyze how an application utilizes

imported libraries with the aim of removing unused components in a best-effort manner. From a security perspective, post-build debloating mitigates risks by reducing the availability of unused functionalities that attackers could otherwise exploit for techniques such as ROP. In more detail, a 3D library could be plagued by memory-safety vulnerabilities, enabling a threat actor to overwrite control data, e.g., return addresses or function pointers. As consequence, an attacker can chain existing instruction snippets (i.e., denoted as gadgets) into a ROP attack, effectively bypassing protections like non-executable memory and achieving arbitrary code execution.

We point out that, despite being effective, trimming approaches should be handled with care. In fact, trimming requires to pursue various trade-offs, such as potential compatibility issues vs enhanced security. Hence, this strategy provides a practical safeguard when dealing with opaque third-party or legacy software but at the price of impairing some functionalities. Another major aspect to consider is that graphic-related applications primarily produce visual outputs or data for building real-world objects, as it happens in CAD. Even if debloated 3D software should preserve functionalities, subtle alterations could lead to unpredictable performance issues. For instance, timing-accurate behaviors may be influenced by a reduced number of instructions to be executed, thus disrupting temporization or timers. The debloating process should then also consider Quality of Experience parameters, which could be hard to evaluate and may discourage the diffusion of practices for reducing the attack surface of 3D applications or graphic-intensive tools.

7. Conclusions

In this paper, we provided a prime analysis on how bloating influences 3D software, especially in terms of properties of its attack surface. To this aim, we investigated various state-of-the-art libraries and tools used for research applications. We found that supporting graphic libraries account for a high level of bloating in dependencies. Surprisingly, transitive dependencies are only responsible for a smaller number of unused functions than direct dependencies, as instead happens in popular software ecosystems like PyPI for Pyhton and npm for JavaScript. The investigation on bloating of 3D software also allowed to outline possible attack surface reduction practices. Promising approaches take advantage of pre-build and post-build debloating strategies, e.g., migrating towards ecosystems with a suitable dependency manager or trim a legacy executable.

Future work aims at refining the "pipeline" created for the analysis of software endowed with 3D functionalities. In more detail, we are working towards approaches able to increase the reproducibility of builds containing 3D libraries for supporting the diffusion of software within the scientific community. Part of our ongoing research is also devoted to investigate more complex 3D applications, e.g., software taking advantage of libraries or components that are loaded at runtime. In parallel, we are interested in preventing attacks caused by the injection of malicious code within the software supply chain composing modern, graphic-intensive applications.

Acknowledgments

This work was partially supported by Project SERICS (PE00000014) and Project RAISE (ECS00000035) under the NRRP MUR program funded by the EU - NGE. This work was also partially supported by the Project "Analisi delle dipendenze e dell'efficacia del debloating per il software sviluppato dal gruppo di ricerca afferente al progetto autofinanziato DIT.AD004.181".

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] C. Davis, J. Collins, J. Fraser, H. Zhang, S. Yao, E. Lattanzio, B. Balakrishnan, Y. Duan, P. Calyam, K. Palaniappan, 3D Modeling of Cities for Virtual Environments, in: 2021 IEEE International Conference on Big Data, 2021, pp. 5587–5596.

[2] C. Romanengo, D. Cabiddu, S. Pittaluga, M. Mortara, Semantic Segmentation of High-resolution Point Clouds Representing Urban Contexts, in: Smart Tools and Applications in Graphics-Eurographics Italian Chapter Conference, The Eurographics Association, 2023.

[3] P. Ivie, D. Thain, Reproducibility in Scientific Computing, ACM Computing Surveys 51 (2018) 1–36.

[4] J. Lee, C. Jung, J. Kim, H. Cha, Panopticus: Omnidirectional 3D Object Detection on Resource-constrained Edge Devices, in: Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, 2024, pp. 1207–1221.

[5] C. Soto-Valero, N. Harrand, M. Monperrus, B. Baudry, A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem, Empirical Software Engineering 26 (2021) 1–44.

[6] M. Alhanahnah, Y. Boshmaf, A. Gehani, SoK: Software Debloating Landscape and Future Directions, in: Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation, 2024, pp. 11–18.

[7] K. Wang, G. Lavoué, F. Denis, A. Baskurt, A Comprehensive Survey on Three-dimensional Mesh Watermarking, IEEE Transactions on Multimedia 10 (2008) 1513–1527.

[8] Y. Wang, Y. Yang, S. Suo, M. Wang, W. Rao, Using Blockchain to Protect 3D Printing From Unauthorized Model Tampering, Applied Sciences 12 (2022) 1–11.

[9] F. Toffalini, M. Ochoa, J. Sun, J. Zhou, Careful-packing: A Practical and Scalable Anti-tampering Software Protection Enforced by Trusted Computing, in: Proceedings of the 9th ACM Conference on Data and Application Security and Privacy, 2019, pp. 231–242.

[10] A. Quach, A. Prakash, L. Yan, Debloating Software Through Piece-Wise Compilation and Loading, in: 27th USENIX Security Symposium, 2018, pp. 869–886.

[11] M. Ali, M. Muzammil, F. Karim, A. Naeem, R. Haroon, M. Haris, H. Nadeem, W. Sabir, F. Shaon, F. Zaffar, et al., SoK: A Tale of Reduction, Security, and Correctness - Evaluating Program Debloating Paradigms and Their Compositions, in: European Symposium on Research in Computer Security, 2023, pp. 229–249.

[12] H. Koo, S. Ghavamnia, M. Polychronakis, Configuration-driven Software Debloating, in: Proceedings of the 12th European Workshop on Systems Security, 2019, pp. 1–6.

[13] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, W. Lee, {RAZOR}: A Framework for Post-deployment Software Debloating, in: 28th USENIX Security Symposium, 2019, pp. 1733–1750.

[14] A. Boyko, T. Funkhouser, Extracting Roads From Dense Point Clouds in Large Scale Urban Environment, ISPRS Journal of Photogrammetry and Remote Sensing 66 (2011) S2–S12.

[15] A. Modi, R. Tikmany, T. Malik, R. Komondoor, A. Gehani, D. D'Souza, Kondo: Efficient Provenance-driven Data Debloating, in: 2024 IEEE 40th International Conference on Data Engineering, 2024, pp. 4965–4978.

[16] Y. Tang, X. Du, A Comparative Study of Full Apps and Lite Apps for Android, arXiv preprint arXiv:2501.06401 (2025).

[17] T. Svoboda, J. Horalek, Analysis of Security Possibilities of Platforms for 3D Graphics, Journal of Telecommunication, Electronic and Computer Engineering 8 (2016) 43–47.

[18] X. Zhang, A. Ahlawat, W. Du, AFrame: Isolating Advertisements From Mobile Applications in Android, in: Proceedings of the 29th Annual Computer Security Applications Conference, 2013, pp. 9–18.

[19] S. Mittal, S. Abhinaya, M. Reddy, I. Ali, A Survey of Techniques for Improving Security of GPUs, Journal of Hardware and Systems Security 2 (2018) 266–285.

[20] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, R. Scopigno, Protected Interactive 3D Graphics via Remote Rendering, ACM Transactions on Graphics 23 (2004) 695–703.

[21] Google, SLSA: Supply-chain Levels for Software Artifacts, 2021.

[22] A. Quach, A. Prakash, Bloat Factors and Binary Specialization, in: Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation, New York, NY, USA, 2019, p. 31–38.
[23] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, Y. Liu, Towards Understanding Third-party Library Dependency in C/C++ Ecosystem, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 2023.
[24] M. Egele, M. Woo, P. Chapman, D. Brumley, Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components, in: 23rd USENIX Security Symposium, San Diego, CA, 2014, pp. 303–317.
[25] S. Mishra, M. Polychronakis, Shredder: Breaking Exploits through API Specialization, in: Proceedings of the 34th Annual Computer Security Applications Conference, New York, NY, USA, 2018, p. 1–16.
[26] M. Zhang, R. Sekar, Control Flow Integrity for COTS Binaries, in: 22nd USENIX Security Symposium, Washington, D.C., 2013, pp. 337–352.
[27] I. Agadakos, N. Demarinis, D. Jin, K. Williams-King, J. Alfajardo, B. Shteinfeld, D. Williams-King, V. P. Kemerlis, G. Portokalidis, Large-scale Debloating of Binary Shared Libraries, Digital Threats 1 (2020).
[28] Cybersecurity and Infrastructure Security Agency, 2025 Minimum Elements for a Software Bill of Materials (SBOM), 2025.
[29] European Commission, NIS2 Directive: Securing Network and Information Systems, 2024.
[30] G. Castelli, A. Cesta, M. Diez, M. Padula, P. Ravazzani, G. Rinaldi, S. Savazzi, M. Spagnuolo, L. Strambini, G. Tognola, E. F. Campana, Urban Intelligence: a Modular, Fully Integrated, and Evolving Model for Cities Digital Twinning, in: 2019 IEEE 16th International Conference on Smart Cities: Improving Quality of Life Using ICT & IoT and AI, 2019, pp. 033–037.
[31] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, G. Portokalidis, Nibbler: Debloating Binary Shared Libraries, in: Proceedings of the 35th Annual Computer Security Applications Conference, New York, NY, USA, 2019, p. 70–83.