



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Security Analysis of Code Bloat in Machine Learning Systems

Master's thesis in Computer science and engineering

Fahmi Abdulqadir Ahmed
Dyako Fatih

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

MASTER'S THESIS 2022

Security Analysis of Code Bloat in Machine Learning Systems

Fahmi Abdulqadir Ahmed
Dyako Fatih



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Security Analysis of Code Bloat in Machine Learning Systems
Fahmi Abdulqadir Ahmed
Dyako Fatih

© Fahmi Abdulqadir Ahmed, 2022.
© Dyako Fatih, 2022.

Supervisor: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering
Examiner: Philipp Leitner, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2022

Security Analysis of Code Bloat in Machine Learning Systems

Fahmi Abdulqadir Ahmed

Dyako Fatih

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

Code bloat is a significant issue in modern software systems as they continue to increase in size and complexity. Furthermore, with the widespread adoption of containerized applications, there is an abundance of unneeded packages that suffer from a wide range of vulnerabilities. In this thesis, we analyze the prevalence of security vulnerabilities in containers used for Machine Learning (ML) systems. We consider two popular ML frameworks, namely, PyTorch and TensorFlow. Making use of container scanning tools, we observed over 100 Common Vulnerabilities and Exposures (CVE) in the tested containers. Our experiments show that debloating using Cimplifier leads to a reduction in the image sizes of up to 49% and a reduction of vulnerabilities of at least 87%. The majority of the removed CVEs can be attributed to the removal of bloat specific to redundant parts of the containers' installed OS packages. A smaller portion of the CVEs detected in the Python packages were removed by Cimplifier.

Keywords: Security, Debloating, Vulnerability Scanning, Machine Learning Systems, Containers, Docker.

Acknowledgements

We would first like to thank our supervisor Asst. Prof. Ahmed Ali-Eldin Hassan for his continuous support, guidance and assistance throughout the whole project. We would also like to thank the our examiner Asst. Prof. Philipp Leitner for the time he took to both examine our thesis as well as give extensive feedback along the way. Their contributions are sincerely appreciated. Additional thanks are extended to Mohannad Alhanahnah and the team that developed Cimplifier for sharing it with us. Finally, we would like to thank our families and friends for their support and understanding throughout the long and arduous process.

Fahmi Abdulqadir Ahmed & Dyako Fatih, Gothenburg, June 2022

Contents

List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Problem Statement	2
1.2 Scope	3
1.3 Thesis Outline	3
2 Theory	5
2.1 Container Virtualization	5
2.2 Machine Learning	5
2.3 Debloating	7
2.4 Security Vulnerabilities	8
2.5 Vulnerability Scanning	9
2.5.1 Container Scanning Tools	9
2.6 Related work	11
3 Methods	13
3.1 Approach	13
3.1.1 Initial Component Analysis	13
3.1.2 Data Collection and Analysis	14
3.2 Problems and Mitigation	15
3.2.1 System Wide Inspection	15
3.2.2 ML Framework Inspection	16
3.3 Environment	18
3.3.1 Target Systems	18
3.3.2 ML Workloads	18
3.3.3 Hardware Configuration	19
4 Results	21
4.1 Prevalence of Vulnerabilities in ML Systems	21
4.2 Vulnerability Reduction by Debloating	24
4.2.1 Container Debloating	24
4.2.2 Scanning Debloated Containers	24
4.2.3 Verification of Removed CVEs	25

5	Discussion	27
5.1	Detected Vulnerabilities	27
5.2	Container Scanning	28
5.3	Effects of Debloating	28
5.4	Possibility of Inflated Results	29
5.5	Threats to Validity	31
6	Conclusion	33
6.1	Future Work	33
	Bibliography	35
A	Appendix	I

List of Figures

2.1	Debloating Docker images with Cimplifier as it pertains to the thesis project	7
2.2	Generalized chart of how container scanning tools function	10
3.1	Work-flow for running the experiment testing each ML system + model combination	14
4.1	Number of vulnerabilities found in each container (a) before debloating and (b) after debloating	22
4.2	Comparison of the size difference between containers before and after debloating	24

List of Tables

4.1	Number of CVEs for each severity level in the PyTorch v1.10.2 image (a) before debloating and (b) after debloating. Numbers in parenthesis represent the number of CVEs found in Python packages	21
4.2	Number of CVEs for each severity level found in the TensorFlow v2.7.0 image (a) before debloating and (b) after debloating. Numbers in parenthesis represent the number of CVEs found in Python packages	23
4.3	Number of CVEs for each severity level found in the TensorFlow v2.7.1 image (a) before debloating and (b) after debloating. Numbers in parenthesis represent the number of CVEs found in Python packages	23
A.1	No. of vulnerabilities detected by Gripe in <code>tensorflow/tensorflow:2.7.0-gpu</code> for each package	I
A.2	No. of vulnerabilities detected by Gripe in <code>tensorflow/tensorflow:2.7.1-gpu</code> for each package	V
A.3	No. of vulnerabilities detected by Gripe in <code>anibali/pytorch:1.10.2-cuda11.3</code> for each package	VII

1

Introduction

Large software projects tend to grow in size and complexity with time. As the amount of code increases with every release, so does the unnecessary features and unused code, also known as code bloat. Consequently, the attack surface of these complex systems expands. Code bloat is becoming a major issue in such systems and as a result has seen a significant interest in recent years [1]–[3]. With more code bloat comes higher energy consumption [4], increased risk of failure and worse security [5].

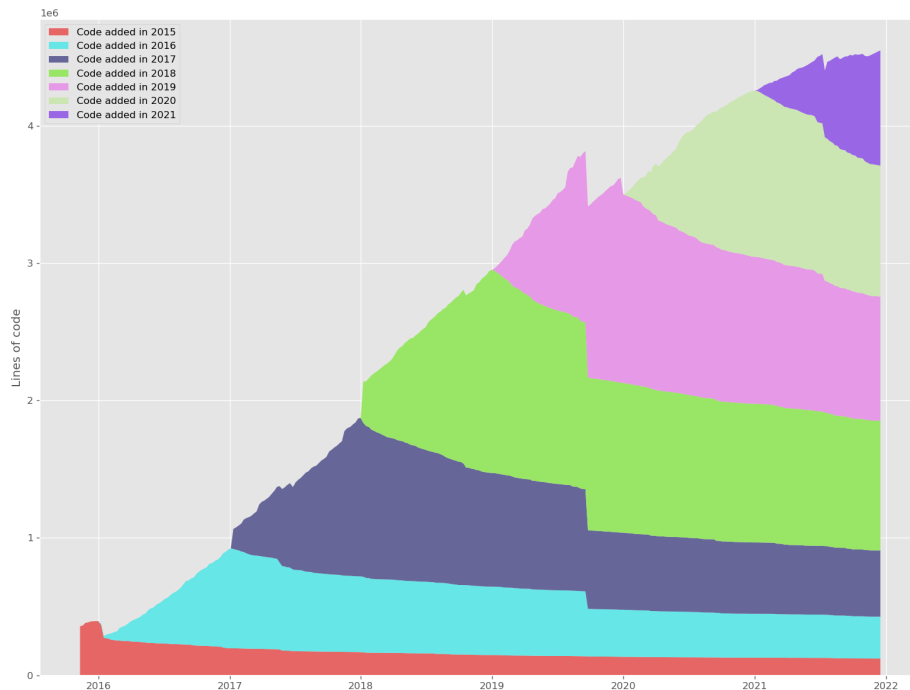


Figure 1.1: The growth of the TensorFlow code base in million lines of code ¹

¹Figure is generated with the tool `git-of-theseus`: <https://github.com/erikbern/git-of-theseus>

Given the many problems code bloat can have on large systems, most of the research focus has been given to big data systems [2]. On the other hand, bloat in Machine Learning (ML) systems have, to our understanding, not received much research attention. This is despite the large expansion of the code base in popular ML frameworks. Sildnik and Wang [6] have shown the prevalence of code bloat in ML systems and that the existing research in debloating can be adapted and applied to Docker containers running ML systems. Azad et al. [5] demonstrated the security risks caused by bloated software and presented a first analysis of the security benefits of debloating web applications.

Xiao et al. [7] found many vulnerabilities in different ML frameworks and some patches were made based on their research from 2017. Since then, frameworks such as TensorFlow have grown substantially introducing even more bloat. Figure 1.1 shows a five-fold increase in the TensorFlow code base containing over 4 million Lines of Code (LoC) since 2017. Therefore, there might be a non-trivial increase in the likelihood that new vulnerabilities have made their way into these frameworks and their long list of dependencies.

Docker containers are commonly used for their flexibility and are frequently distributed online with little oversight [8]. The study notes that containers containing vulnerabilities can spread and be used extensively by many users. Sildnik and Wang [6] showed that roughly half of the code in a Docker container used for ML workloads could be removed while retaining full functionality for each separate use case.

1.1 Problem Statement

It has been shown that debloating can improve the security of a system [5]. However, this was limited to web applications, and little (if any) has been done to quantify security vulnerabilities in ML systems introduced by code bloat. Sculley et al [9] discuss the unique challenges that are specific to ML systems in addition to the maintenance problems of traditional code. They summarize these challenges into several categories, including ML-system anti-patterns, Data dependencies bloat, Configuration bloat and Feedback loops.

The primary objective of this thesis is to quantify the security improvements caused by debloating Docker containers used for ML systems. In doing so, we attempt to answer the following Research Questions (RQs):

RQ1: How common are vulnerabilities within containers running ML systems?

RQ2: How well do existing vulnerability scanning tools work on debloated containers?

RQ3: How effective is debloating in reducing the amount of security vulnerabilities?

Answering these research questions requires a way to quantify vulnerabilities in Docker containers. One approach is to use existing vulnerability scanning tools

such as Trivy², Anchore³ and Clair⁴. These tools scan Docker images for publicly disclosed vulnerabilities found in the Common Vulnerabilities and Exposures (CVE) database [10].

1.2 Scope

The scope of this project will be limited to analyzing the security of containerized ML systems and any potential effects debloating may have in the context of improving security. We will also look into the efficacy of using vulnerability scanning tools for the purpose of analyzing the security implications of debloating.

A list of the constraints of the project is given below:

- The focus of the project will be limited to Docker containers given their popularity and consistency for repeated use.
- This project will not develop a debloating tool but will instead make use of an already existing tool.
- We will not build our own vulnerability scanning tools, but instead use and test existing ones.
- We will only make use of the container scanning tools and not other tools or software produced by their respective creators.
- The vulnerabilities that are investigated are limited to already disclosed ones and discovering new vulnerabilities is beyond the scope of the project.

1.3 Thesis Outline

In the next chapter we will give an overview of different topics, tools and technologies used, as well as some previous work done debloating and security. This includes virtualization with containers, Machine Learning, debloating with Cimplifier, security vulnerabilities, and vulnerability scanning. In Chapter 3 we go over our testing methodology, ways to validate removal vulnerabilities from debloating, and the working environment used in this thesis. Chapter 4 contains the results from our experiments. Chapter 5 discusses the achieved results, their validity, the answers to the research questions, and point out some ideas for future research. Finally, in Chapter 6 the conclusion is presented.

²<https://aquasecurity.github.io/trivy>

³<https://github.com/anchore/anchore-engine>

⁴<https://quay.github.io/clair/>

2

Theory

This chapter explains the concepts that serve as background to the rest of the thesis. These concepts include virtualization tools, ML systems, debloating and vulnerability scanning. We close the chapter by discussing the related work.

2.1 Container Virtualization

Modern applications rely on other services each with their own set of dependencies that may conflict with others. Containers are a form of operating system virtualization that allow developers to package a software/application along with all of its dependencies. These include executable binaries, libraries and configuration files.

Docker is an open source containerization platform that allows developing, shipping and running applications [11]. It makes applications portable and isolated by packaging them in containers that run in the Docker Engine. A text document called Dockerfile serves as a blueprint to build a Docker image which includes everything needed to run an application. Docker images are a standalone executable packages of software that contain the instructions necessary to create containers that run on the Docker Engine. The use of containers to package different components of an application can be advantageous by avoiding many problems such as platform differences, conflicting dependencies and missing dependencies.

2.2 Machine Learning

Machine Learning (ML) is a branch of Artificial Intelligence (AI) that allows systems to learn through experience and improve their performance with respect to a given task [12]. In this context, experience is the observations of the available data by the ML algorithm. A ML task can be described in terms of how the ML system can process the dataset in order to achieve the type of inference we want it to make based on the problem we want to solve. The performance metric is specific to the task being solved and is usually used to evaluate how the ML model performs on unseen data.

There are mainly three types of ML algorithms:

- **Supervised learning:** This category of algorithms uses labelled datasets to learn a function that maps the inputs in the training dataset to the desired label. The training data consists of (x_i, y_i) pairs, where x_i is the input to the algorithm and y_i is the desired output. The learning task involves finding a function f such that $f(x_i)$ is as close to y_i as possible. Classification and regression tasks are examples of supervised learning.
- **Unsupervised learning:** These algorithms are trained with unlabelled datasets and learn useful properties of the structure of the unlabelled data. Such algorithms build an internal representation of the input data to discover its inherent structure, data groupings or hidden patterns. An example of this category is clustering, which divides the data into groups based on their similarities.
- **Reinforcement learning:** An agent observes the environment and learns the best strategy known as a policy. The agent selects actions based on the policy to maximize cumulative rewards. Such algorithms do not use a fixed dataset but instead interact with an environment and learn by trial and error.

There are many libraries and frameworks that allow developers to easily build ML models without needing to worry about the underlying algorithms. Some of the most popular ones are:

- **TensorFlow¹:** TensorFlow [13] was originally built by the Google Brain team within Google's Machine Intelligence Research organization for internal use but was released as open-source project in 2015. It is a ML framework that supports both CPU and GPU computing devices. The core of TensorFlow is built with C++ and runs on several operating systems. TensorFlow uses dataflow graph representation for numerical computation and state. This allows the deployment of applications on distributed clusters, local workstations, mobile devices and custom-designed accelerators [13]. Nodes in the dataflow graph represent units of local computation while the edges represent the numerical data arrays, i.e. tensors, that are communicating between the nodes.
- **PyTorch²:** PyTorch [14] is an open source ML framework developed by Meta AI research group (formerly known as FAIR) that supports hardware accelerators such as GPUs. Although most of PyTorch is written in C++, it is based on Python and the Torch library. PyTorch supports tensor computation with automatic differentiation and strong GPU acceleration [14]. In contrast to TensorFlow which uses static dataflow graphs, PyTorch uses dynamic computation graphs.
- **Scikit-learn³:** This is an open source Python framework built on top of NumPy and SciPy that is designed for supervised and unsupervised ML algorithms

¹<https://www.tensorflow.org/>

²<https://pytorch.org/>

³<https://scikit-learn.org/>

[15]. NumPy is used for data and model parameters while SciPy provides algorithms for optimization, linear algebra, interpolation and other tasks.

In this thesis, a ML system refers to a Docker container and an integrated ML framework.

2.3 Debloating

Software debloating is the process of removing unused code and unneeded functionalities such as excess libraries from a system or program. A debloated program may have reduced functionality beyond the specific use-case of the end user compared to the original program. As a result, debloating can significantly reduce resource consumption and attack surface. This can potentially improve the performance [4] and security of software systems [5].

Debloating can be performed on compiled binaries, at the source-code level or at a file level removing unused files from the target system. Most of the proposed techniques for software debloating target either source code or binaries. However, there are a few who target file-level debloating, such as Cimplifier [16], which is the main focus of this thesis project.

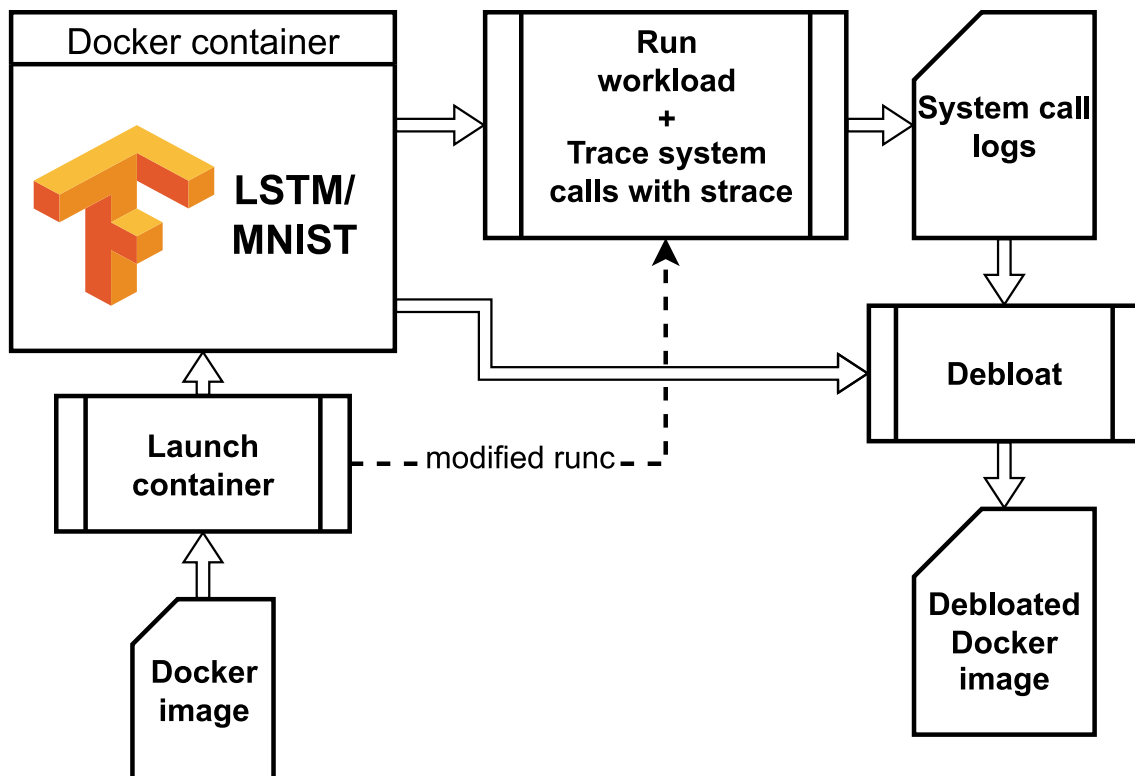


Figure 2.1: Debloating Docker images with Cimplifier as it pertains to the thesis project

Cimplifier is a file-level debloating tool that uses dynamic analysis to collect information on how the application executables use resources by collecting system call logs

with **strace**. It takes an input container and partitions it into a set of containers isolated from each other to provide privilege separation. The partitioned containers contain just the resources needed to run them to perform their functionality while still satisfying the user-defined constraints [16].

Cimplifier operates in three steps: resource identification, partitioning and glueing. The first step uses system call logs to collect information on how various processes access resources such as files, inter-process communication and network objects in a running container. This step identifies which files that are needed by the end user. The second step uses the results from the resource identification and any user-defined constraints to partition the original container into a number of smaller containers. However, a single container can also be produced by setting the partitioning policy to all-one-context. This policy does not perform any container partitioning and places all executables in one container. The final step, glueing, uses remote process execution (RPE) to allow the resulting containers to communicate with each other as necessary to maintain the functionality of the original container. RPE allows a process in one container to execute another process in a different container.

All relevant system call logs can be collected with **strace** automatically when the command **docker run** is used to launch the container, which can be achieved by using a modified version of **runc**⁴ written by Sildnik and Wang [6]. This version of **runc** starts the strace process right before the container starts to run the entry command, identifying the runc process ID (PID) and the current docker ID. Figure 2.1 shows an example of using Cimplifier to debloat a container running an LSTM model with the MNIST dataset using the all-one-context partitioning policy. The figure also demonstrates how we use Cimplifier in this thesis.

2.4 Security Vulnerabilities

Each of the machine learning algorithm types explained in Section 2.2 has its own set of vulnerabilities [17]. In general, security vulnerabilities in ML systems can be divided mainly into two groups. The first group consists of vulnerabilities in the ML system during training, e.g. data poisoning. The second group targets online ML systems after the underlying model has been trained. Evasion attacks and membership inference are two examples of such vulnerabilities.

There has been quite some effort put into tackling ML system vulnerabilities of the kinds mentioned above. These vulnerabilities are, however, not code-specific. Instead, they are “algorithm vulnerabilities” inherent to the model design as described by Spring et al. [18]. There are also what Spring et al. [18] refer to as “implementation vulnerabilities”, which are caused by the underlying source code or binaries of some software.

However, today’s systems not only contain the source code but also many dependencies in the form of libraries, frameworks and Operating System (OS) packages. Each

⁴<https://github.com/wy0917/runc/tree/v1.0.0-rc93-strace>

dependency might contain their own set of vulnerabilities. One way of identifying vulnerabilities is thorough scanning of the source code and/or the dependencies with a scanning tool. This allows the necessary countermeasures to be taken to eliminate or limit the identified security risks.

Publicly disclosed vulnerabilities are tracked and given a unique identification number, such vulnerabilities are commonly referred to as CVEs. CVE stands for Common Vulnerabilities and Exposures. The next section further discusses this topic.

2.5 Vulnerability Scanning

A growing number of security vulnerabilities in systems are discovered every year. Publicly disclosed vulnerabilities are listed in the CVE list [10]. The CVE program is maintained by the MITRE corporation, with funding from the US Cybersecurity and Infrastructure Security Agency (CISA) of the US Department of Homeland Security. Entries in this list are assigned unique identification numbers by a CVE Numbering Authority (CNA). The Common Vulnerability Scoring System (CVSS) is used to assess the severity level of a vulnerability with a score ranging from 0 to 10 [19]. A higher number indicates a higher degree of severity.

Vulnerability scanning is a powerful tool used to identify security weaknesses and vulnerabilities in a system with the aim of mitigating security risks and protecting the exposure of sensitive data. Scanning a container for vulnerabilities make it easier to detect and mitigate any openly known threats that the installed libraries and dependencies may have. Some of the common vulnerability scanning tools that scan Docker containers are Trivy, Anchore, Clair and Snyk⁵.

All four of the previously mentioned tools are able to be integrated into development pipelines making it easier to stay up to date on newly discovered vulnerabilities whenever they are disclosed or when an update is made to the container. We are, however, focusing on the core container scanning tool provided. Note that the core container scanning components Anchore provide are called Grype and Syft, and from now on, will be referred to as Grype.

2.5.1 Container Scanning Tools

Trivy, Grype, and Clair are all open source and free to use. Snyk on the other hand is a paid service with limited trial use and is not fully open source, hence we cannot make any concrete claims as to how it actually functions. Based on testing the different container scanning tools, Snyk appears to function in a similar fashion to the other three container scanning tools (will be referred to as scanners from now on).

Following Figure 2.2 from left to right, the scanner pulls the Docker image from either a local repository or a remote one, which can be composed of several layers,

⁵<https://snyk.io/>

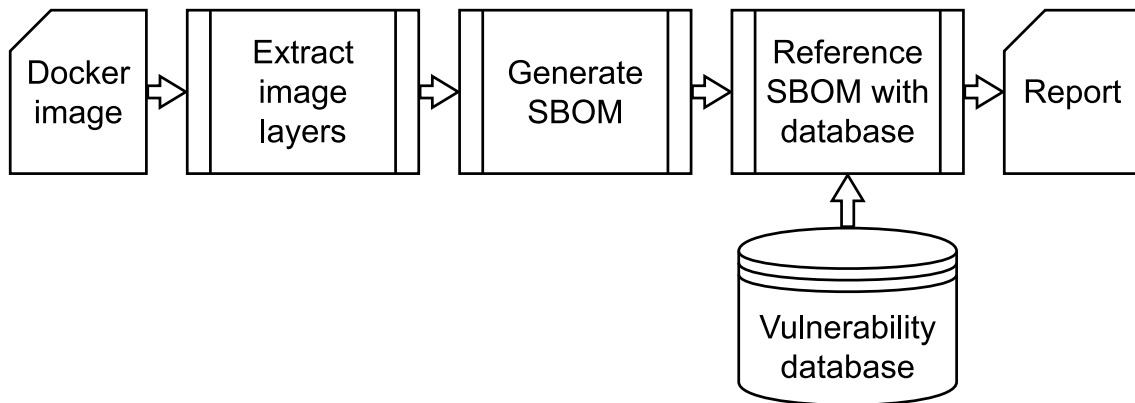


Figure 2.2: Generalized chart of how container scanning tools function

allowing it to scan the file system of the docker container as defined by the Docker image.

After extracting the image layers, the next step is to generate a Software Bill Of Materials (SBOM). The SBOM is a list of all the different packages, libraries and/or software along with their respective versions installed on the container. Generating the SBOM is achieved by inspecting log files generated by different package managers. For Ubuntu based images, `/var/lib/dpkg/status` is the file which documents the list of all the preinstalled packages found in the Operating System (OS). It also includes packages installed with Debian PacKaGe (DPKG), or the more commonly used tool Advanced Packing Tool (APT). Trivy and Grype have separate software, which is included with their installs, called Fanal and Syft respectively. These are also able to detect packages installed with other package managers, such as Pip⁶, by inspecting log files similar to the `status` file for the OS packages. A list of the files Trivy (Fanal) inspects can be found in the documentation for Trivy⁷. Grype (Syft) inspects similar files.

The final step is to take the SBOM and match the entries with a database that is regularly updated from several sources. This database is normally stored locally, but it can be hosted remotely in a Docker container. Sources include National Vulnerability Database (NVD), GitHub Security Advisory (GHSA), and many of the popular Linux distributions' own security advisories. The full list can be found in the scanners' documentation. Each entry in the generated SBOM will be referenced with the database and reported if a vulnerability relating to the specific entry exists. The format of the scanning report can be customized or use some of the predefined formats, such as, JSON. Each detected vulnerability is reported with a vulnerability ID, which in most cases is a CVE ID, the vulnerable package, severity level, and other descriptive information if available.

⁶<https://pypi.org/project/pip/>

⁷<https://aquasecurity.github.io/trivy/v0.25.3/docs/vulnerability/detection/language/>

2.6 Related work

Code bloat is claimed to increase energy consumption and have a negative impact on security. There exists some research in the field of debloating from recent years [20], [21] with a focus on either debloating the source code or the binary files directly. We will go over a couple of different debloating techniques, discuss how they function and the security implications of the different techniques.

CARVE [20] is a debloating technique used to debloat the source code of programs or packages. This technique allow developers to define how the debloating should occur in a fine grained manner. The developer adds feature mappings to the source code, and then instruct the debloater which pieces of code to remove or replace it with. The feature mappings can be done in two different ways:

- Implicit Feature Mapping (`///FEATURE_X`): Maps the code segment underneath the tag to **FEATURE_X**. A code segment may be a function, if block, for loop, or any other syntactically enclosed segment.
- Explicit Feature Mapping (`///FEATURE_X ...code... ///`): Maps anything withing between the tags to **FEATURE_X**. Additionally, a developer may specify code to replace the mapped code with enclosing it within a pair of `///^` tags placed withing the explicit feature mapping.

These feature mappings along with the source code is then given to the debloater. Depending on the specifications given to the debloater as to which features should be removed, it removes the source code segments with the tags matching the names specified.

A part of the debloating process is to rebuild the software from the debloated source code. This may however introduce potential vulnerabilities in the form of new gadgets [22]. Gadgets are a set of instructions that ends with a return, indirect jump, or function call instruction. These kinds of gadgets may be used by an attacker by chaining instructions to build a payload with the existing code, also called code reuse attack. In [22], they demonstrate that although the total number of gadgets may be reduced by debloating using the kind of technique used in [20], [23], the new gadgets introduced may worsen the security of the program.

LibFilter [21] is another debloating system developed to debloat dynamically linked libraries. The system uses a tool called Egalito⁸ to extract the Function Call Graph (FCG). The FCG starts from the entry point of an executable binary and traces all the functions used from the libraries the binary calls. Anything that is not used is then replaced with a halt command (`halt`).

This method of debloating does not involve altering the source code of any binaries or libraries. In fact, it does not need access to the source code at all. Another benefit of this kind of debloating is the fact that instructions are only removed from the libraries, hence there is no way for new gadgets to be introduced.

⁸<https://egalito.org/>

These kinds of techniques are effective at debloating and are shown to generally improve security. However, debloating whole ML systems with such techniques is most likely an unfeasible task due to the size of ML systems. It is also not clear if the use of these debloating techniques would work with the mix of languages used in the ML systems (mainly Python and C/C++). To the best of our knowledge, the only successful debloating endeavor of ML systems was demonstrated in [6]. Thus, the security benefits of debloating ML systems have not been researched as far as we are aware.

3

Methods

This chapter describes the approach taken to tackle the RQs described in Section 1.1, problems encountered early on and how they were mitigated, as well as the testing environment, the tools used, and the target systems analyzed in the thesis.

3.1 Approach

The approach is split into two primary parts: (1) Initial Component Analysis and (2) Data Collection and Analysis. The first part of this section aims to test and analyze different scanners, select ML systems to test the scanners on, and debloat the ML systems. This is done in order to enable us to choose the appropriate scanners for this thesis, as well as setting up an environment where the systems and scanners work properly. Information gathered during the first part will answer RQ1. It will also give us the prerequisite knowledge needed to proceed with running the experiments required to help answer RQ2 and RQ3. The RQs are stated again for ease of reading:

RQ1: How common are vulnerabilities within containers running ML systems?

RQ2: How well do existing vulnerability scanning tools work on debloated containers?

RQ3: How effective is debloating in reducing the amount of security vulnerabilities?

3.1.1 Initial Component Analysis

Firstly, we select two popular ML frameworks that the ML systems will be based on. We choose to focus on two large and popular ML frameworks, specifically PyTorch and TensorFlow. The choice of Docker images is dictated by ML frameworks and their corresponding versions used in this thesis. The versions of the ML frameworks should be recent to keep the thesis relevant.

Secondly, different container scanning tools are analyzed and tested. The scanners considered for this thesis are: (1) Trivy, (2) Grype, (3) Clair, and (4) Snyk. They are each tested on a ML system image to assess their applicability for the type of

systems we are analyzing. The scanners need to preferably be capable of scanning the Python based frameworks and packages installed on them.

Many of the existing debloating tools target source code or executable binaries and are meant for code which is compiled, such as C/C++. However, based on the research conducted by Sildnik and Wang [6], Cimplifier [16] is the only tool that produces working Docker containers running ML systems. This project will therefore use Cimplifier.

Lastly, we will set up the final environment used for the experiments in the second part of the approach. This will primarily be determined by the requirements of the software tools that will be used. Additionally, it will be based on what hardware is required to properly make use of hardware acceleration for efficiently running ML systems. Due to limited hardware resources available on premise, we opted to perform the experiments on the cloud platform Amazon Web Services (AWS). Delays in receiving access to the cloud platform halted us from proceeding with the experiments for a substantial amount of time.

3.1.2 Data Collection and Analysis

After analyzing and selecting suitable scanners, receiving access to the AWS cloud platform, choosing the Docker containers, and configuring the environment, we continue with the experiments and gather data for analysis. Several combination of Docker containers running different ML systems and models will be tested as shown in Figure 3.1. The data gathered here will help answer RQ2 and RQ3.

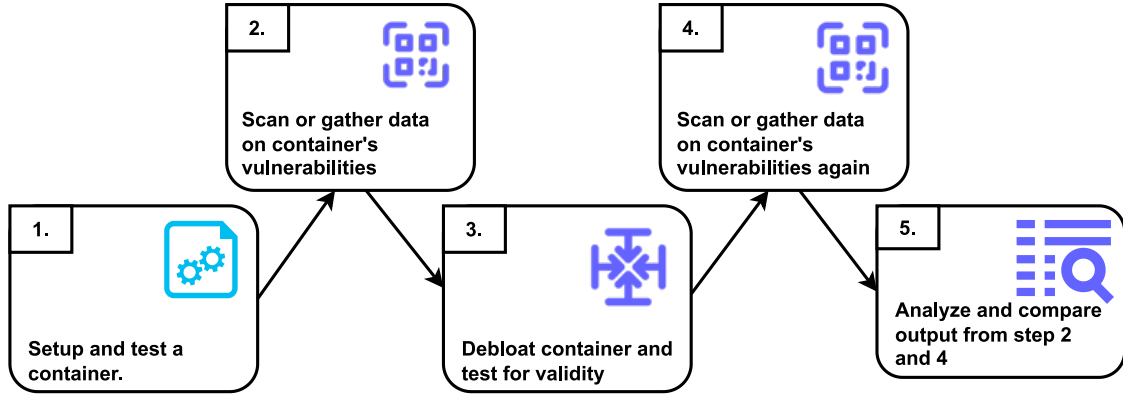


Figure 3.1: Work-flow for running the experiment testing each ML system + model combination

The *first* and *second* steps, as seen in Figure 3.1, are meant to gather data on the the baseline performance of the ML system and on how vulnerable the system is respectively. Baseline performance of each configuration of ML framework plus reference workload (ML model) will be compared to their debloated variant in the *third* step. The extent to how vulnerable the ML system is will be assessed based on the CVE count reported after scanning the Docker image the system is based on. The reports from the scanners are used in later steps.

In step *three*, the ML system is debloated with the same reference workload used in the *first* step. The debloated ML system is run again, its performance is measured, and then compared with the results from the *first* step to ensure that the system is still performing properly.

The *fourth* step was supposed to be the same as the *second* step, but due to how Cimplifier debloats Docker containers, the ML systems cannot be properly scanned with any of the scanning tools and produce usable reports. Thus to be able to detect vulnerabilities in the debloated ML systems, we resorted to manually inspect the debloated systems with assistance of two different scripts made to mitigate the previously mentioned problem. The specific problem and how the manual inspection was performed is described in more detail in Section 3.2. The detected vulnerabilities are documented and then we proceed with the final step.

In the *fifth* and final step, the reports produced from the container scanners in step *two* and the manually detected vulnerabilities are compared between the original ML system and its debloated variants. CVE count is our main metric for determining if debloating may improve the security of the ML system. We will also look into the vulnerabilities to see what parts of the ML system are affected the most/least.

3.2 Problems and Mitigation

Attempting to scan a debloated ML system was not as fruitful as described in Section 3.1.2. We therefore had to resort to manually inspect the debloated systems for vulnerabilities, or CVEs. In this section, we are describing the reason why the scanners were not effective, and the approach we took to inspect the systems manually.

3.2.1 System Wide Inspection

A container scanner attempts to document each and every package installed in a container. It does so not by searching through the entire file system, but rather by reading key files which document all the installed packages and their versions. The documented packages are then referenced with a database to determine if any package is vulnerable or not. In Section 2.5.1, this is described in more detail. When Cimplifier is used to debloat a container, files that are not necessary to run the reference workload are removed. This includes the exact files the scanner reads which document the installed packages and important system information such as OS distribution and version, leading to no packages being detected and therefore no vulnerabilities being reported. Accessing the files as a part of the reference workload will leave the files intact after debloating. This however inverts the problem, causing the scanner to “detect” every package just as before the container was debloated. Thus leading to the scanner producing a report identical to the one before the system is debloated.

The approach we took to find the remaining CVEs mixes manual work and a script to speed up the process making it feasible to perform. Note that this was done on a Ubuntu system. The step by step approach is described below:

- Extract the entire file system from the debloated container, making sure the actual container is created then extracted to get the complete file system and not extracting the layered structure of a Docker image
- Using the report generated when scanning the original system, parse each specific package name correlating to every vulnerability reported as entries in the report. Additional information, such as, severity level may also be extracted to better compare the result with the original scanning results
- Using the `find` command, we are able to search through the file system and find anything that contains a string of choice. Each vulnerable package is searched for as such:

```
find <path> -iname "*<name>*"
```

where `<path>` is the root directory of the extracted container file system, `<name>` is the parsed name of a vulnerable package, and flag `-iname` tells the tool to be case insensitive. Anything in the file system containing the string `<name>` in any way is returned along with the path to where it was found

- Whatever is found by the script is then manually checked to make sure what is found is actually the binary file that is vulnerable, and not something else that happens to contain the searched string in its name

3.2.2 ML Framework Inspection

Another problem we faced was trying to pinpoint the vulnerabilities found in the ML framework TensorFlow. Unlike the OS packages, which have one binary file for each piece of software or feature, the ML framework is compiled into relatively few Shared Object (SO) files. The CVEs detected by the scanner did not point to specific SO files in the file system, but rather the installation of the entire framework. The scanner also referenced the source code files that are vulnerable. However, the source code files are not a part of the installed framework.

TensorFlow v2.7.0 consists of 4755 C and C++ source code files which are compiled down to 61 SO files when installed on the Docker container. Reading the code repository to match the vulnerable source code files with the SO files is a daunting and time consuming task. Thus we resorted to inspect the SO files directly. The compiled SO files comply with the Executable and Linking Format (ELF)¹ allowing us to potentially extract useful information from them. The step by step approach is described below:

¹<https://man7.org/linux/man-pages/man5/elf.5.html>

- We created a list of all the vulnerable source code files referenced by the scanning report. We double checked that the reported files match the ones documented in the TensorFlow GitHub repository
- The path to the folder that contains all the installed components of TensorFlow is specified
- The locations of all SO files are found with the command:

```
find <path> -iname "*.so"
```

where `<path>` is the initial directory to search through (path to the TensorFlow folder). The path to every SO in the TensorFlow framework is gathered

- Each SO file is inspected with the `readelf` command to find every source code file used to compile them:

```
readelf -sW <so-file> | grep 'FILE' | awk '{print $NF}'
```

The `-sW` flags tell the `readelf` command to return all the entries found in the symbols table. Then we use `grep` to return the entries of type `FILE`. Finally, `awk '{print $NF}'` returns the last column of every remaining entry, which are the names of the files of interest. This extracts the names of every source code file used to compile the SO file

- The extracted file names are then compared to the list of vulnerable files created earlier. Any match would indicate that the CVE(s) are present in the file. This is done for every SO file
- In the case when a Python file contains a vulnerable code or function, we use the following command to locate it:

```
grep --include=*.py -rnw . -e "<function-name>"
```

The meaning of different options is as follows: `-r` is recursive, `-n` option includes the line number in each output, `-w` matches the whole word and `-e` is used to specify the pattern used for the search. The `<function-name>` is the name of the vulnerable function

It is possible to find the file names in the SO because the TensorFlow framework is open source. Thus, all debug information is left in there and not stripped out to hide any information from the public.

3.3 Environment

This section will describe ML systems that are analyzed as well as the underlying hardware used during the experiments.

3.3.1 Target Systems

The target system refers to the Docker containers which will be analyzed for security vulnerabilities. These containers are generated from Docker images that contain all the required software and ML frameworks preinstalled to be able to train and use ML models.

The Docker images used to generate the containers are all based on an Ubuntu 20.04 base image with CUDA installed to allow the use of GPU acceleration which is necessary to effectively train and use ML models. This thesis focused on two popular frameworks, namely PyTorch and TensorFlow. Different versions of the ML frameworks are preinstalled in the different container images.

PyTorch Image

For testing a PyTorch based system, we are using the `anibali/pytorch:1.10.2-cuda11.32` image to generate the Docker container. PyTorch v1.10 was the latest version available when we started the project.

TensorFlow Image

On the other hand for TensorFlow, we are using two images with different versions of TensorFlow v2.7. The two images (`tensorflow/tensorflow:2.7.0-gpu3` and `tensorflow/tensorflow:2.7.1-gpu`) are both identical feature wise. Version v2.7.1 was a security update that patched over 50 vulnerabilities and was released in February 2022. Using both versions of the TensorFlow framework allow us to see if the scanning tools are able to detect vulnerabilities between versions of the same framework.

3.3.2 ML Workloads

To enable the debloating of the ML systems, a reference workload is required by Cimplyfier. The workload gets logged with the tool `strace`, and the resulting logs will tell Cimplyfier what components need to remain in the debloated container to still function. The reference workload will be ML models being trained and tested.

LSTM/MNIST: The LSTM model is trained with the MNIST dataset and will be used with both target systems. The code for both PyTorch and TensorFlow frameworks were acquired from GitHub⁴. Minor modifications were made to fix a "divide-by-zero" bug.

²<https://hub.docker.com/r/anibali/pytorch>

³<https://hub.docker.com/r/tensorflow/tensorflow>

⁴<https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML>

DLA/CIFAR-10: The Deep Layer Aggregation (DLA) model is trained with the CIFAR-10 dataset. This workload will be used for the system with the PyTorch framework. The code was acquired from GitHub⁵.

3.3.3 Hardware Configuration

The hardware available to a ML system affects the results of the debloating process. It is common practice to use GPU(s) to accelerate a ML workload, be that training, testing or inference. This is commonly achieved with the use of the CUDA API stack allowing the workload to be offloaded to Nvidia GPU(s). To make sure the CUDA API is used, we used an AWS EC2 cloud instance.

The specific instance used during the project is the **g4dn.xlarge**⁶. This instance has 4 virtual threads of Intel Xeon Scalable (Cascade Lake) processors, 16 GB of memory, and a Nvidia T4 Tensor Core GPU⁷. The operating system used with the instance is the **UBUNTU 18.04 based AWS Deep Learning AMI**⁸.

⁵<https://github.com/kuangliu/pytorch-cifar>

⁶<https://aws.amazon.com/ec2/instance-types/g4/>

⁷<https://www.nvidia.com/en-us/data-center/tesla-t4/>

⁸https://aws.amazon.com/marketplace/pp/prodview-x5nivojpquy6y?sr=0-1&ref_=beagle&applicationId=AWSMPContessa

4

Results

This chapter presents the results of the conducted experiments. First, we look into the vulnerabilities of different severity levels found in the containers before debloating. Second, we present the results obtained by debloating the selected containers.

4.1 Prevalence of Vulnerabilities in ML Systems

The first step was to analyze the prevalence of publicly disclosed vulnerabilities in containers running ML systems. This was achieved by using the scanning tools mentioned in Chapter 3. There are three images we used for the two chosen frameworks, TensorFlow v2.7.0, TensorFlow v2.7.1 and PyTorch v1.10.2. Trivy and Gype divided the installed packages in these images into two categories, OS packages and language-specific packages, whereas Clair and Snyk only detected the OS packages. The total number of CVEs detected by all four scanners can be seen in Figure 4.1a.

Table 4.1a shows the results of scanning the PyTorch container. We can see that the results of Gype and Trivy are very similar, reporting a total of 119 and 118 CVEs respectively. However, Trivy does not have **Negligible** severity level and instead reports these as **Low**. Both scanners identified 113 CVEs in OS packages and only 5 and 6 CVEs are found in Python packages by Trivy and Gype respectively.

Table 4.1: Number of CVEs for each severity level in the PyTorch v1.10.2 image (a) before debloating and (b) after debloating. Numbers in parenthesis represent the number of CVEs found in Python packages

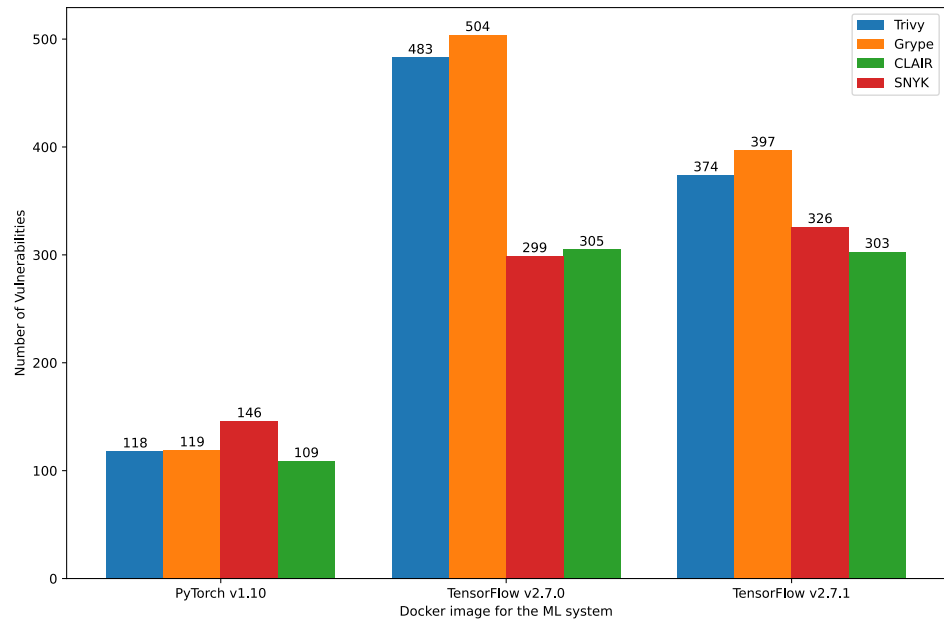
(a) Before debloating

Scanner	Total	Critical	High	Medium	Low	Negligible
Gype	119 (6)	3 (3)	6	52 (2)	46 (1)	12
Trivy	118 (5)	3 (3)	6	51 (1)	58 (1)	-

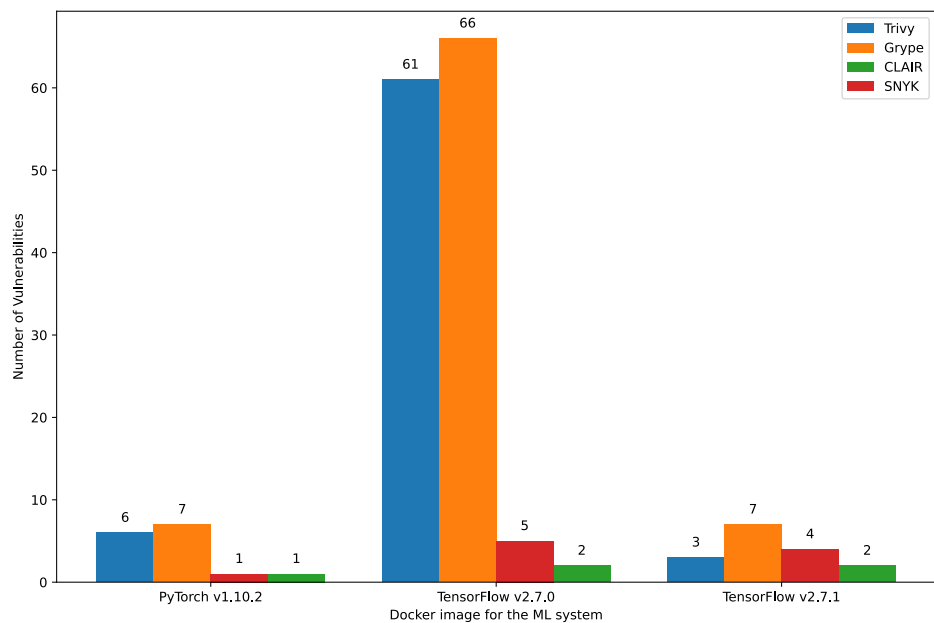
(b) After debloating

Scanner	Total	Critical	High	Medium	Low	Negligible
Gype	7 (6)	3 (3)	0	2 (2)	2 (1)	0
Trivy	6 (5)	3 (3)	0	1 (1)	2 (1)	-

4. Results



(a) Number of vulnerabilities detected by each scanner before debloating



(b) Number of vulnerabilities remaining after debloating

Figure 4.1: Number of vulnerabilities found in each container (a) before debloating and (b) after debloating

The scanning results of the two TensorFlow images are shown in Tables 4.2a and 4.3a. Similar as before, the results of both scanners are similar. However, in Table 4.2a, 440 of the 504 CVEs identified by Gripe are categorized as OS-package vulnerabilities while 64 are Python-package CVEs. For Trivy, 425 are OS packages and the Python-package vulnerabilities are 58. Most of the language-specific CVEs are related to TensorFlow v2.7.0.

Table 4.2: Number of CVEs for each severity level found in the TensorFlow v2.7.0 image (a) before debloating and (b) after debloating. Numbers in parenthesis represent the number of CVEs found in Python packages

(a) Before debloating

Scanner	Total	Critical	High	Medium	Low	Negligible
Gripe	504 (64)	1 (1)	42 (22)	296 (41)	127	38
Trivy	483 (58)	1 (1)	40 (20)	279 (37)	163	-

(b) After debloating

Scanner	Total	Critical	High	Medium	Low	Negligible
Gripe	66 (62)	1	22 (22)	41 (39)	2	0
Trivy	61 (57)	1	20 (20)	38 (36)	2	-

On the other hand, the TensorFlow framework v2.7.1 does not have any publicly disclosed CVEs at the time of writing. As a result, Python-package CVEs are only 6 for Gripe and 1 for Trivy.

Table 4.3: Number of CVEs for each severity level found in the TensorFlow v2.7.1 image (a) before debloating and (b) after debloating. Numbers in parenthesis represent the number of CVEs found in Python packages

(a) Before debloating

Scanner	Total	High	Medium	Low	Negligible
Gripe	397 (6)	24	224 (2)	111 (4)	38
Trivy	374 (1)	22	206	146 (1)	-

(b) After debloating

Scanner	Total	High	Medium	Low	Negligible
Gripe	7 (4)	3 (2)	3 (2)	1	0
Trivy	3	1	1	1	-

4.2 Vulnerability Reduction by Debloating

This section presents the results after debloating and the observed reduction of CVEs that are still present in the containers. The number of vulnerabilities found in each container after debloating is summarized in Figure 4.1b.

4.2.1 Container Debloating

Four containers were debloated with the Cimplifier tool. The two models used for the experiments are LSTM using the MNIST dataset and DLA with CIFAR-10.

Results from the debloating of all the containers show that the performance of all the tested models remained unchanged and the container size was reduced substantially. As seen in Figure 4.2, the size reduction was 37% for the PyTorch container running both models, which is a smaller reduction to what Sildnik and Wang [6] observed, and 49% for both TensorFlow containers running the LSTM/MNIST model. The accuracy and training time for each model was effectively the same before and after debloating, which tells us the debloating process has not affected the integrity of the models.

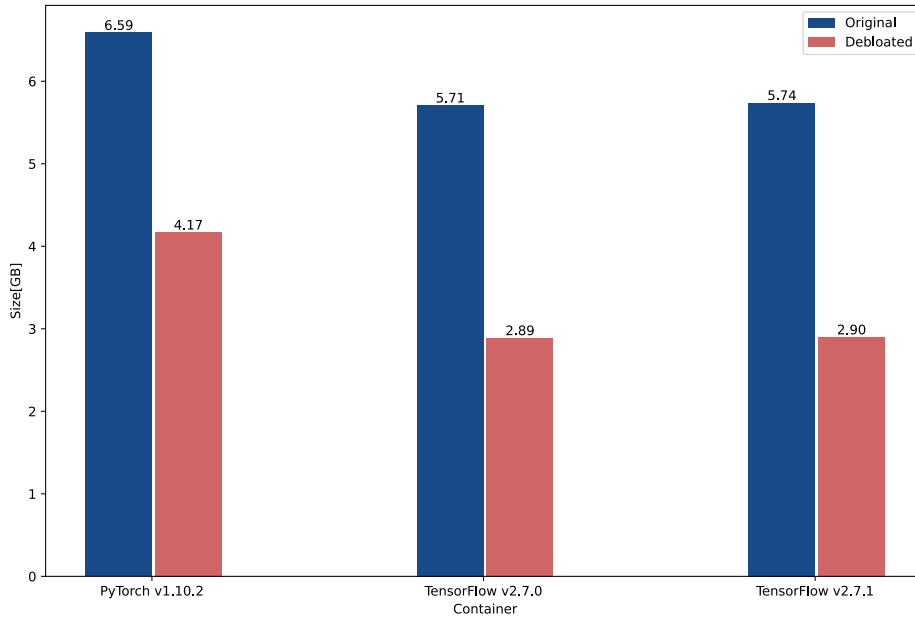


Figure 4.2: Comparison of the size difference between containers before and after debloating

4.2.2 Scanning Debloated Containers

In Section 4.1, we saw the scanning results of the original images. The number vulnerabilities detected in each container before debloating can be seen in Tables 4.1a,

4.2a, 4.3a and in Figure 4.1a. However, all scanners failed to detect any vulnerabilities after the images were debloated.

This is due to the fact that the relevant metadata files and package managers are removed by the debloating process. Thus they fail to detect the installed packages and their versions.

4.2.3 Verification of Removed CVEs

To mitigate the problem of not being able to scan the debloated images, we verified the packages that remain installed in the resulting container from debloating as explained in Section 3.2.1. The result is the list of CVEs associated with each remaining package from the vulnerability report that was obtained from scanning the original container.

As shown in Table 4.1b, the number of CVEs that remain after debloating the container running the LSTM/MNIST model is just 7 for Grype and 6 for Trivy. All vulnerabilities reported are found in three packages: `bash`, `Pillow` and `NumPy`. Both scanners detect `bash` and `Pillow` whereas Trivy does not detect `NumPy`. The debloated `anibali/pytorch` container that was running DLA/CIFAR-10 gave identical results as in Table 4.1b.

In Table 4.3b, we can see that there are only 7 CVEs found by Grype and 3 by Trivy that are left in the debloated LSTM/MNIST-TensorFlow container. All of those CVEs are found in three packages: `bash`, `python3.8` and `urllib3`. The `urllib3` python library is present in the debloated container and there are four CVEs in the library. However, only Grype detected `urllib3` when scanning the original container, hence the different results in the table.

The scanning result of `tensorflow/tensorflow:2.7.0-gpu` has many more vulnerabilities compared to `tensorflow/tensorflow:2.7.1-gpu`. This is because TensorFlow v2.7.1 is an updated version of v2.7.0 with all the known vulnerabilities patched. Both Trivy and Grype reported 57 CVEs in the TensorFlow framework in `tensorflow/tensorflow:2.7.0-gpu`. Most of these CVEs are found in the C/C++ files in the source code of TensorFlow which are compiled to a number of SO files. This makes it hard to verify exactly which files from the source code are removed from the package by debloating because the scanning reports only reference the source code of the framework and not the SO files.

Section 3.2.2 describes how we verified the presence of CVEs in the SO files of the TensorFlow v2.7.0 framework. There are 61 SO files in the framework before debloating and 13 are removed in the debloated version. All 57 CVEs found in the TensorFlow framework are still present after debloating in the 48 remaining SO files. The remaining CVEs in this container are shown in Table 4.2b.

5

Discussion

The chapter discusses the results presented in Chapter 4. The first section focuses on the detected vulnerabilities to answer RQ1. Sections 5.2 and 5.3 focus on RQ2 and RQ3 followed by discussing the validity of the results. The chapter ends with suggestions for future work.

5.1 Detected Vulnerabilities

In Section 4.1, we presented the results of scanning the ML systems. As shown in Figure 4.1a, we can see that there are many vulnerabilities detected by all four scanners. The scanning reports by Trivy and Grype divided the installed packages into two categories, OS-packages and language-specific packages, i.e., Python packages for ML Systems. Clair and Snyk, however, only reported vulnerabilities found in the OS packages. Snyk does support detecting Python packages in container images, but for container registry integration and not the Snyk Container command line interface (CLI), which scans local images.

As shown in Tables 4.1a, 4.2a and 4.3a, the overwhelming majority of the CVEs are found in the OS packages. For example, the Python-package CVEs in the PyTorch container only account for 5% and 4% of the total CVEs reported by Grype and Trivy respectively. All of those were found in only two packages: `Pillow` and `NumPy`. Although TensorFlow v2.7.0 image had the most Python-package CVEs of all the images scanned, they are still around 12% of the total CVEs detected by both Trivy and Grype in the TensorFlow v2.7.0 image. This time, all of those CVEs are found in four packages: `pip`, `NumPy`, `urllib3` and `TensorFlow`.

In Figure 4.1a, we can see that Grype detected 504 CVEs in the TensorFlow v2.7.0 container. All 504 vulnerabilities were found in the 115 packages listed in Table A.1. Of the 504 total CVEs, the package `linux-libc-dev` and the `TensorFlow` library accounted for 205 and 190 vulnerabilities reported by Grype and Trivy respectively. Neither of those two packages were detected by Snyk or Clair, which explains the large difference in the total number of CVEs reported by them compared to Trivy and Grype.

Based on the results of the scanners shown in these tables and Figure 4.1a, we can answer **RQ1** and conclude that containers running ML systems are vulnerable and contain numerous CVEs of varying severity levels in both OS and Python packages.

5.2 Container Scanning

Using container scanning tools such as Trivy and Gype, we were able to successfully quantify the number of vulnerabilities in all three container images. However, the process of debloating the containers with Cimplifier caused the scanners to not yield any results. All the tested scanners fail to scan the debloated container images because the meta-data files that document the installed packages are removed. This problem and our mitigation strategy is described with more details in Section 3.2.1.

Even though the container scanning tools failed to scan the debloated container images, the reports they produced from scanning the original container images were still useful in detecting the remaining CVEs. We know that Cimplifier does not alter files but only removes them. Thus we can assume with certainty that, in the worst case, the maximum number of CVEs remaining is equal before and after debloating. Therefore, we can use the original scanning reports to narrow down our search for the remaining CVEs.

In the case of the CVEs detected in TensorFlow v2.7.0 seen in Table 4.2a, we needed to perform some deeper analysis. The reports produced by Trivy and Gype did neither point to nor name any specific installed files when reporting the CVEs. Rather, they referenced the original source code found on the TensorFlow’s GitHub page. By compiling a list of all the vulnerable files and inspecting each SO file, which is explained in greater detail in Section 3.2.2, we were able to find which SO files were vulnerable. This allows us to determine if any CVEs are removed by debloating the container image.

The short answer to **RQ2** is: The existing vulnerability scanning tools we tested do not work on debloated containers. However, the reports produced when scanning the non-debloated container images are very useful in helping us asses the reduction of CVEs due to debloating with Cimplifier.

5.3 Effects of Debloating

The debloating process successfully shrunk the containers and reduced the number of CVEs by up to 98% while retaining the desired functionality for each tested use case. This is because most of the removed CVEs were detected in the OS packages, which account for the majority of the detected CVEs by Trivy and Gype. For the purpose of a ML system, only a few of these packages are needed. Out of the reported vulnerable OS packages, only `bash` and `python3.8` are retained after debloating the TensorFlow containers. The reason `python3.8` is reported as an OS package is because it is installed directly with the OS package manager.

However, debloating did not help in removing any of 57 CVEs detected in the source code of the TensorFlow v2.7.0 library. The majority of these 57 CVEs were found in the C/C++ files. As explained previously in Section 3.2, the C/C++ source files are compiled into a few SO files, and most of the vulnerable C/C++ files are found in the `libtensorflow_framework.so.2` and `_pywrap_tensorflow_internal.so` files. As the names imply, these files contain the bulk of the source code of the TensorFlow framework. Only one CVE, CVE-2022-23563, was found in Python source code files where TensorFlow uses the function `mktemp` to create temporary files. Given that Cimplifier is a file-level debloating tool, all used files are left intact, which makes it difficult for file-level debloating tools to effectively reduce such vulnerabilities.

Considering the substantial reduction in the number of CVEs found in all the tested containers, we can answer **RQ3**. Debloating is very effective in removing all of the unneeded packages along with their vulnerabilities. The few packages that remain installed in the debloated container are mostly the Python packages that were used by the ML system.

5.4 Possibility of Inflated Results

The results from the experiments show a remarkable reduction in the total number of CVEs after debloating the tested containers. These results may raise some suspicions regarding the methodology for finding the figures. The large majority of the vulnerable packages are part of the OS in the container. Most of these packages are redundant and not used because of the fact that the host system’s OS kernel is shared with the container and we were only running the ML models and nothing else.

The only OS package with a known CVE remaining in the debloated PyTorch container is `bash`. This makes sense as we used a Bash terminal within the container during the debloating process. The higher values reported by Trivy and Gripe seen in Figure 4.1b, are due to them also detecting language based packages that remain after debloating.

Both TensorFlow containers have hundreds of CVEs detected by the vulnerability scanners and almost all of the CVEs were removed after debloating. There are a few reasons why the number of detected CVEs are so high:

- In Tables A.1 and A.2, we can observe the presence of “dev” variants of many of the vulnerable packages are reported for the TensorFlow containers. An examples of this are the `libexpat1` package and its “dev” counterpart `libexpat1-dev`, both accounting for 15 CVEs each. These 15 CVEs from the `libexpat1` package and its “dev” variants are technically duplicates, because they share the same dependencies that are actually vulnerable. Even though they are duplicate, they still impose a valid concern. Having two packages using the same dependencies increases the entry points to the vulnerable dependencies. This phenomenon is not present in the PyTorch container. Our

assumption is that the “dev” packages are not supposed to be there, as there are in fact developer versions of the TensorFlow containers available.

- Similar to the previous point, some of the reported packages share CVEs. `binutils`, `libbinutils`, `binutils-common`, `libctf-nobfd0`, `libctf0` and `binutils-x86-64-linux-gnu` are a set of reported packages all sharing the same dependencies. This phenomenon can be observed in all three containers, as seen in Tables A.1, A.2, and A.3.
- We observed an occurrence of actual duplicate reporting of vulnerabilities. The `urllib3` package, which was detected in both TensorFlow containers with Gype, has two vulnerabilities, but four are reported. The two vulnerabilities are documented in both the CVE list and in the GHSA, hence having both CVE and GHSA IDs. Gype has for some reason reported both sets of vulnerabilities, somewhat inflating the CVE count. This is however the only occurrence of such duplication we managed to detect.

We consider that the repetition of vulnerabilities reported described in the the first two points as valid. If we decide to only count vulnerabilities with unique IDs, the `tensorflow/tensorflow:2.7.0-gpu` container would still have detected 301 vulnerabilities based on the the report produced by Gype.

Furthermore, the difference in the number of vulnerabilities detected by different scanners as seen in Figures 4.1a and 4.1b can be due to several reasons:

- Differences in which metadata files they read and how they interpret them.
- What sources they use to update their vulnerability databases.
- The scanners tend to update their vulnerability databases at different rates. For example, Trivy does it every 12 hours, Clair every 6 hours and Anchore checks for updates every time a scan is performed. Moreover, the time we performed scanning with different scanners is not necessarily the same for all images. This is because the goal was not to compare the results obtained from different scanners but rather a comparison between the number of CVEs detected before and after debloating.

Whichever way one might decide to look at the reported CVE count of the bloated containers, the number of CVEs are still substantial even if one makes a more conservative interpretation of the reports. Based on the analysis of the reported vulnerabilities, we can say with confidence that the total number of CVEs is reduced substantially and that it can be attributed to the debloating of the containers.

5.5 Threats to Validity

This section discusses some potential threats to the validity of this thesis.

Construct Validity

Using the scanners to detect CVEs in the non-debloat ML systems can be assumed to be accurate to a certain degree. We did see variations in the results from different vulnerability scanners as well as a very large number of CVEs for some of the ML systems. There can be many explanations as to why these variations may happen, which we discuss in more detail in Section 5.4. This is the main reason we used several scanners to better understand the overall security state of the systems. The use of CVE counts as a measure for the vulnerability of a system is not all-encompassing. However, it is a useful way of analyzing larger pieces of software or systems where analyzing the code directly is unfeasible.

In the case of validating the remaining CVEs after debloating, we believe our method is accurate. The method was tested on the non-debloat containers and we always found all the CVEs reported by each scanner. Therefore, it can be assumed that we would not miss any remaining CVEs when searching for them in the debloat containers.

Internal Validity

The substantial reduction of CVEs found in the ML systems can only be attributed to the debloating done with Cimplifier.

External Validity

Although the number of analyzed systems was small, we believe that the results should generalize quite well to other models. This can already be seen in the PyTorch container when training both the LSTM and DLA models, where the results were identical. This can be attributed to the fact that most of the removed CVEs are found in OS packages that are unlikely to be used by other models. On the other hand, most of the CVEs detected in the Python packages were still found in the debloat containers, which are the packages commonly used in ML systems, such as NumPy. However, there might be variations in what Python packages that are used by different models, which would still account for a very small proportion of the overall CVE count.

Our method for validating the overall number of remaining CVEs in a debloat container should be applicable to any container with a Linux-based file system. However, the methodology used to verify the remaining CVEs in the TensorFlow shared object library files installed in the container is more specific to TensorFlow. However, the core idea of the method should work, with small modifications, for any open-source code base written in C/C++ with the plain-text symbols still available in the compiled files.

6

Conclusion

This thesis demonstrated the prevalence of security vulnerabilities in containers used for ML systems. Furthermore, we analyzed the effectiveness of debloating in reducing the number of publicly exposed vulnerabilities in such containers. Our experiments show that debloating bloated containers running ML systems can reduce up to 98% of the vulnerabilities detected by the Gripe vulnerability scanner while retaining the desired functionality of the original containers. The majority of the removed CVEs are detected in the OS packages installed in the containers. However, the extent of the reduction in the reported vulnerabilities depends on the scanner used and whether it can detect the programming language-specific packages.

While file-level debloating proved to be effective in reducing the total number of CVEs in the Docker container as a whole, it is limited in removing vulnerabilities in the ML frameworks as Cimplifier failed to remove any CVEs from the TensorFlow framework. A more fine-grained debloating technique would more likely be capable of removing CVEs from such frameworks.

6.1 Future Work

The file-level debloating done with Cimplifier is effective in reducing the total number of vulnerabilities found in containerized ML systems. According to our observations on the structure of ML frameworks such as TensorFlow, there is further room for improvement. The majority of the library files comprising the framework are still found in the debloated ML system, hence most vulnerabilities associated with the framework remains. Performing a more fine-grained method of debloating could yield even better results in removing vulnerabilities.

Making use of a tool such as LibFilter is an avenue to potentially further improve upon the results achieved in this thesis. Even though the ML frameworks analyzed are Python based, a large portion of the core functionality is written in C/C++, especially TensorFlow. One idea is to test a two-stage debloating process. First stage would be the file level debloating, and the second stage would be the lower level library debloating.

6. Conclusion

Another path for the future work is to try debloating on distributed ML systems using a framework such as Horovod¹. With more complex ML models being trained, the use of distributed setups with more total computing power is beneficial. It would be an interesting challenge coordinating the debloating of multiple containers working on the same task.

¹<https://horovod.ai/>

Bibliography

- [1] S. Bhattacharya, K. Gopinath, and M. G. Nanda, “Combining concern input with program analysis for bloat detection”, in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds., ACM, 2013, pp. 745–764. DOI: 10.1145/2509136.2509522. [Online]. Available: <https://doi.org/10.1145/2509136.2509522>.
- [2] Y. Bu, V. R. Borkar, G. Xu, and M. J. Carey, “A bloat-aware design for big data applications”, in *International Symposium on Memory Management, ISMM 2013, Seattle, WA, USA, June 20, 2013*, P. Cheng and E. Petrank, Eds., ACM, 2013, pp. 119–130. DOI: 10.1145/2491894.2466485. [Online]. Available: <https://doi.org/10.1145/2491894.2466485>.
- [3] A. Quach, A. Prakash, and L. Yan, “Debloating software through piece-wise compilation and loading”, in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds., USENIX Association, 2018, pp. 869–886. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/quach>.
- [4] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta, “The interplay of software bloat, hardware energy proportionality and system bottlenecks”, in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems, HotPower ’11, Cascais, Portugal, October 23, 2011*, R. Bianchini and P. Dutta, Eds., ACM, 2011, 1:1–1:5. DOI: 10.1145/2039252.2039253. [Online]. Available: <https://doi.org/10.1145/2039252.2039253>.
- [5] B. A. Azad, P. Laperdrix, and N. Nikiforakis, “Less is more: Quantifying the security benefits of debloating web applications”, in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., USENIX Association, 2019, pp. 1697–1714. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/azad>.
- [6] M. Sildnik and Y. Wang, “Debloating machine learning systems”, 2021. [Online]. Available: <https://hdl.handle.net/20.500.12380/302760>.

- [7] Q. Xiao, K. Li, D. Zhang, and W. Xu, “Security risks in deep learning implementations”, in *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, IEEE Computer Society, 2018, pp. 123–128. DOI: 10.1109/SPW.2018.00027. [Online]. Available: <https://doi.org/10.1109/SPW.2018.00027>.
- [8] D. Huang, H. Cui, S. Wen, and C. Huang, “Security analysis and threats detection techniques on docker container”, in *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, 2019, pp. 1214–1220. DOI: 10.1109/ICCC47050.2019.9064441.
- [9] D. Sculley, G. Holt, D. Golovin, *et al.*, “Hidden technical debt in machine learning systems”, in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2503–2511. [Online]. Available: <https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcdf2674f757a2463eba-Abstract.html>.
- [10] *About the cve program*. [Online]. Available: <https://www.cve.org/About/Overview>.
- [11] B. B. Rad, H. J. Bhatti, and M. Ahmadi, “An introduction to docker and analysis of its performance”, *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.
- [12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [13] M. Abadi, P. Barham, J. Chen, *et al.*, “Tensorflow: A system for large-scale machine learning”, in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds., USENIX Association, 2016, pp. 265–283. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- [14] A. Paszke, S. Gross, F. Massa, *et al.*, “Pytorch: An imperative style, high-performance deep learning library”, in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 8024–8035. [Online]. Available: <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html>.
- [15] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, “Scikit-learn: Machine learning in python”, *CoRR*, vol. abs/1201.0490, 2012. arXiv: 1201.0490. [Online]. Available: <http://arxiv.org/abs/1201.0490>.

-
- [16] V. Rastogi, D. Davidson, L. D. Carli, S. Jha, and P. D. McDaniel, “Cimplifier: Automatically debloating containers”, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds., ACM, 2017, pp. 476–486. DOI: 10.1145/3106237.3106271. [Online]. Available: <https://doi.org/10.1145/3106237.3106271>.
- [17] P. Xiong, S. Buffett, S. Iqbal, P. Lamontagne, M. S. I. Mamun, and H. Molyneaux, “Towards a robust and trustworthy machine learning system development”, *CoRR*, vol. abs/2101.03042, 2021. arXiv: 2101.03042. [Online]. Available: <https://arxiv.org/abs/2101.03042>.
- [18] J. M. Spring, A. Galyardt, A. D. Householder, and N. M. VanHoudnos, “On managing vulnerabilities in AI/ML systems”, *CoRR*, vol. abs/2101.10865, 2021. arXiv: 2101.10865. [Online]. Available: <https://arxiv.org/abs/2101.10865>.
- [19] N. I. of Standards and T. (NIST), *Vulnerability metrics*. [Online]. Available: <https://nvd.nist.gov/vuln-metrics/cvss>.
- [20] M. D. Brown and S. Pande, “Carve: Practical security-focused software debloating using simple feature set mappings”, in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST’19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 1–7, ISBN: 9781450368346. DOI: 10.1145/3338502.3359764. [Online]. Available: <https://doi.org/10.1145/3338502.3359764>.
- [21] B. Shteinfeld, “Libfilter: Debloating dynamically-linked libraries through binary recompilation”, *Undergraduate Honors Thesis. Brown University*, 2019.
- [22] M. D. Brown and S. Pande, “Is less really more? towards better metrics for measuring security improvements realized through software debloating”, in *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: <https://www.usenix.org/conference/cset19/presentation/brown>.
- [23] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning”, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, 2018, pp. 380–394. DOI: 10.1145/3243734.3243838. [Online]. Available: <https://doi.org/10.1145/3243734.3243838>.

A

Appendix

The Appendix contains a table listing every vulnerable package detected with Grype and how many CVEs are related to every package for all three Pre-debloated container.

Table A.1: No. of vulnerabilities detected by Grype in tensorflow/tensorflow:2.7.0-gpu for each package

Package	Version	No. of Vulns
linux-libc-dev	5.4.0-89.100	148
tensorflow	2.7.0	57
libexpat1	2.2.9-1build1	15
libexpat1-dev	2.2.9-1build1	15
libhdf5-103	1.10.4+repack-11ubuntu1	11
libc-dev-bin	2.31-0ubuntu9.2	11
libhdf5-dev	1.10.4+repack-11ubuntu1	11
hdf5-helpers	1.10.4+repack-11ubuntu1	11
libc6-dev	2.31-0ubuntu9.2	11
libhdf5-cpp-103	1.10.4+repack-11ubuntu1	11
libc-bin	2.31-0ubuntu9.2	11
libc6	2.31-0ubuntu9.2	11
libctf-nobfd0	2.34-6ubuntu1.3	4
binutils-x86-64-linux-gnu	2.34-6ubuntu1.3	4
libsqlite3-0	3.31.1-4ubuntu0.2	4
libctf0	2.34-6ubuntu1.3	4
libbinutils	2.34-6ubuntu1.3	4
urllib3	1.25.8	4
libsepol1	3.0-1	4
binutils-common	2.34-6ubuntu1.3	4
binutils	2.34-6ubuntu1.3	4
libpolkit-gobject-1-0	0.105-26ubuntu1.1	3
libpython3.8-minimal	3.8.10-0ubuntu1 20.04.1	3
libpython3.8	3.8.10-0ubuntu1 20.04.1	3
python3.8-minimal	3.8.10-0ubuntu1 20.04.1	3
libzmq3-dev	4.3.2-2ubuntu1	3

libpcre3	2:8.39-12build1	3
libpolkit-agent-1-0	0.105-26ubuntu1.1	3
policykit-1	0.105-26ubuntu1.1	3
libpython3.8-dev	3.8.10-0ubuntu1 20.04.1	3
python3.8-dev	3.8.10-0ubuntu1 20.04.1	3
unzip	6.0-25ubuntu1	3
libzmq5	4.3.2-2ubuntu1	3
libpython3.8-stdlib	3.8.10-0ubuntu1 20.04.1	3
python3.8	3.8.10-0ubuntu1 20.04.1	3
bsdutils	1:2.34-0.1ubuntu9.1	2
libgcrypt20	1.8.5-5ubuntu1	2
libkrb5-3	1.17-6ubuntu4.1	2
libfdisk1	2.34-0.1ubuntu9.1	2
libgssapi-krb5-2	1.17-6ubuntu4.1	2
python-pip-whl	20.0.2-5ubuntu1.6	2
patch	2.7.6-6	2
libmount1	2.34-0.1ubuntu9.1	2
libkrb5support0	1.17-6ubuntu4.1	2
libkadm5srv-mit11	1.17-6ubuntu4.1	2
libgssrpc4	1.17-6ubuntu4.1	2
util-linux	2.34-0.1ubuntu9.1	2
libuuid1	2.34-0.1ubuntu9.1	2
fdisk	2.34-0.1ubuntu9.1	2
python3-pip	20.0.2-5ubuntu1.6	2
libk5crypto3	1.17-6ubuntu4.1	2
libkrb5-dev	1.17-6ubuntu4.1	2
libblkid1	2.34-0.1ubuntu9.1	2
libsmartcols1	2.34-0.1ubuntu9.1	2
mount	2.34-0.1ubuntu9.1	2
libkadm5clnt-mit11	1.17-6ubuntu4.1	2
krb5-multidev	1.17-6ubuntu4.1	2
libkdb5-9	1.17-6ubuntu4.1	2
gcc-9	9.3.0-17ubuntu1 20.04	1
libhcrypto4-heimdal	7.7.0+dfsg-1ubuntu1	1
libudev1	245.4-4ubuntu3.11	1
libasan5	9.3.0-17ubuntu1 20.04	1
gcc-9-base	9.3.0-17ubuntu1 20.04	1
perl-modules-5.30	5.30.0-9ubuntu0.2	1
systemd-sysv	245.4-4ubuntu3.13	1
libjpeg-turbo8-dev	2.0.3-0ubuntu1.20.04.1	1
libssl1.1	1.1.1f-1ubuntu2.8	1
cpp	4:9.3.0-1ubuntu2	1
libgcc-9-dev	9.3.0-17ubuntu1 20.04	1
perl	5.30.0-9ubuntu0.2	1

libperl5.30	5.30.0-9ubuntu0.2	1
libxml2	2.9.10+dfsg-5ubuntu0.20.04.1	1
numpy	1.21.3	1
systemd-timesyncd	245.4-4ubuntu3.13	1
systemd	245.4-4ubuntu3.13	1
g++	4:9.3.0-1ubuntu2	1
zlib1g-dev	1:1.2.11.dfsg-2ubuntu1.2	1
zlib1g	1:1.2.11.dfsg-2ubuntu1.2	1
g++-9	9.3.0-17ubuntu1 20.04	1
pip	20.2.4	1
libcryptsetup12	2:2.2.2-3ubuntu2.3	1
libgssapi3-heimdal	7.7.0+dfsg-1ubuntu1	1
passwd	1:4.8.1-1ubuntu5.20.04.1	1
libhx509-5-heimdal	7.7.0+dfsg-1ubuntu1	1
liblzma5	5.2.4-1ubuntu1	1
libkrb5-26-heimdal	7.7.0+dfsg-1ubuntu1	1
libasn1-8-heimdal	7.7.0+dfsg-1ubuntu1	1
libheimntlm0-heimdal	7.7.0+dfsg-1ubuntu1	1
libroken18-heimdal	7.7.0+dfsg-1ubuntu1	1
libgmp10	2:6.2.0+dfsg-4	1
libc6	2.34-1ubuntu2	1
dbus-user-session	1.12.16-2ubuntu2.1	1
xz-utils	5.2.4-1ubuntu1	1
login	1:4.8.1-1ubuntu5.20.04.1	1
libssl2-modules-db	2.1.27+dfsg-2	1
pip	20.0.2	1
coreutils	8.30-3ubuntu2	1
dbus	1.12.16-2ubuntu2.1	1
gcc	4:9.3.0-1ubuntu2	1
libheimbase1-heimdal	7.7.0+dfsg-1ubuntu1	1
perl-base	5.30.0-9ubuntu0.2	1
libstdc++-9-dev	9.3.0-17ubuntu1 20.04	1
libpam-systemd	245.4-4ubuntu3.13	1
libdbus-1-3	1.12.16-2ubuntu2.1	1
bash	5.0-6ubuntu1.1	1
gzip	1.10-0ubuntu4	1
libapparmor1	2.13.3-7ubuntu5.1	1
tar	1.30+dfsg-7ubuntu0.20.04.1	1
libjpeg-turbo8	2.0.3-0ubuntu1.20.04.1	1
libssl2-2	2.1.27+dfsg-2	1
python3-urllib3	1.25.8-2ubuntu0.1	1
openssl	1.1.1f-1ubuntu2.8	1
libsystemd0	245.4-4ubuntu3.13	1
cpp-9	9.3.0-17ubuntu1 20.04	1

A. Appendix

libwind0-heimdal	7.7.0+dfsg-1ubuntu1	1
------------------	---------------------	---

Table A.2: No. of vulnerabilities detected by Grype in tensorflow/tensorflow:2.7.1-gpu for each package

Package	Version	No. of Vulns
linux-libc-dev	5.4.0-97.110	117
libexpat1-dev	2.2.9-1build1	15
libexpat1	2.2.9-1build1	15
libc-bin	2.31-0ubuntu9.2	11
libhdf5-cpp-103	1.10.4+repack-11ubuntu1	11
libc6-dev	2.31-0ubuntu9.2	11
hdf5-helpers	1.10.4+repack-11ubuntu1	11
libc-dev-bin	2.31-0ubuntu9.2	11
libc6	2.31-0ubuntu9.2	11
libhdf5-dev	1.10.4+repack-11ubuntu1	11
libhdf5-103	1.10.4+repack-11ubuntu1	11
binutils	2.34-6ubuntu1.3	4
urllib3	1.25.8	4
binutils-common	2.34-6ubuntu1.3	4
binutils-x86-64-linux-gnu	2.34-6ubuntu1.3	4
libbinutils	2.34-6ubuntu1.3	4
libsqlite3-0	3.31.1-4ubuntu0.2	4
libctf-nobfd0	2.34-6ubuntu1.3	4
libsepol1	3.0-1	4
libctf0	2.34-6ubuntu1.3	4
libpcre3	2:8.39-12build1	3
libzmq3-dev	4.3.2-2ubuntu1	3
libzmq5	4.3.2-2ubuntu1	3
unzip	6.0-25ubuntu1	3
python3.8-dev	3.8.10-0ubuntu1 20.04.2	2
libmount1	2.34-0.1ubuntu9.1	2
python3.8	3.8.10-0ubuntu1 20.04.2	2
libpolkit-gobject-1-0	0.105-26ubuntu1.2	2
util-linux	2.34-0.1ubuntu9.1	2
policykit-1	0.105-26ubuntu1.2	2
libsmartcols1	2.34-0.1ubuntu9.1	2
python-pip-whl	20.0.2-5ubuntu1.6	2
libpolkit-agent-1-0	0.105-26ubuntu1.2	2
libk5crypto3	1.17-6ubuntu4.1	2
python3-pip	20.0.2-5ubuntu1.6	2
libkadm5srv-mit11	1.17-6ubuntu4.1	2
libkrb5-dev	1.17-6ubuntu4.1	2
libblkid1	2.34-0.1ubuntu9.1	2
libkrb5-3	1.17-6ubuntu4.1	2
libkrb5support0	1.17-6ubuntu4.1	2
libgssrpc4	1.17-6ubuntu4.1	2

patch	2.7.6-6	2
libgssapi-krb5-2	1.17-6ubuntu4.1	2
libpython3.8-stdlib	3.8.10-0ubuntu1 20.04.2	2
libkdb5-9	1.17-6ubuntu4.1	2
libuuid1	2.34-0.1ubuntu9.1	2
krb5-multidev	1.17-6ubuntu4.1	2
libkadm5clnt-mit11	1.17-6ubuntu4.1	2
libpython3.8	3.8.10-0ubuntu1 20.04.2	2
libpython3.8-minimal	3.8.10-0ubuntu1 20.04.2	2
python3.8-minimal	3.8.10-0ubuntu1 20.04.2	2
bsdutils	1:2.34-0.1ubuntu9.1	2
mount	2.34-0.1ubuntu9.1	2
fdisk	2.34-0.1ubuntu9.1	2
libpython3.8-dev	3.8.10-0ubuntu1 20.04.2	2
libfdisk1	2.34-0.1ubuntu9.1	2
libwind0-heimdal	7.7.0+dfsg-1ubuntu1	1
passwd	1:4.8.1-1ubuntu5.20.04.1	1
libhx509-5-heimdal	7.7.0+dfsg-1ubuntu1	1
zlib1g	1:1.2.11.dfsg-2ubuntu1.2	1
libgssapi3-heimdal	7.7.0+dfsg-1ubuntu1	1
libasan5	9.3.0-17ubuntu1 20.04	1
libstdc++-9-dev	9.3.0-17ubuntu1 20.04	1
libdbus-1-3	1.12.16-2ubuntu2.1	1
libjpeg-turbo8	2.0.3-0ubuntu1.20.04.1	1
libperl5.30	5.30.0-9ubuntu0.2	1
dbus-user-session	1.12.16-2ubuntu2.1	1
libkrb5-26-heimdal	7.7.0+dfsg-1ubuntu1	1
gzip	1.10-0ubuntu4	1
libheimntlm0-heimdal	7.7.0+dfsg-1ubuntu1	1
perl-base	5.30.0-9ubuntu0.2	1
libssl1.1	1.1.1f-1ubuntu2.10	1
gcc	4:9.3.0-1ubuntu2	1
libcryptsetup12	2:2.2.2-3ubuntu2.3	1
gcc-9-base	9.3.0-17ubuntu1 20.04	1
pip	20.2.4	1
g++-9	9.3.0-17ubuntu1 20.04	1
liblzma5	5.2.4-1ubuntu1	1
pip	20.0.2	1
libapparmor1	2.13.3-7ubuntu5.1	1
tar	1.30+dfsg-7ubuntu0.20.04.1	1
libsasl2-modules-db	2.1.27+dfsg-2	1
bash	5.0-6ubuntu1.1	1
dbus	1.12.16-2ubuntu2.1	1
perl-modules-5.30	5.30.0-9ubuntu0.2	1

libgcc-9-dev	9.3.0-17ubuntu1 20.04	1
gcc-9	9.3.0-17ubuntu1 20.04	1
libasn1-8-heimdal	7.7.0+dfsg-1ubuntu1	1
cpp	4:9.3.0-1ubuntu2	1
cpp-9	9.3.0-17ubuntu1 20.04	1
login	1:4.8.1-1ubuntu5.20.04.1	1
python3-urllib3	1.25.8-2ubuntu0.1	1
libheimbase1-heimdal	7.7.0+dfsg-1ubuntu1	1
libroken18-heimdal	7.7.0+dfsg-1ubuntu1	1
libhcrypto4-heimdal	7.7.0+dfsg-1ubuntu1	1
libxml2	2.9.10+dfsg-5ubuntu0.20.04.1	1
zlib1g-dev	1:1.2.11.dfsg-2ubuntu1.2	1
g++	4:9.3.0-1ubuntu2	1
openssl	1.1.1f-1ubuntu2.10	1
libudev1	245.4-4ubuntu3.13	1
libsasl2-2	2.1.27+dfsg-2	1
xz-utils	5.2.4-1ubuntu1	1
perl	5.30.0-9ubuntu0.2	1
libgmp10	2:6.2.0+dfsg-4	1
libjpeg-turbo8-dev	2.0.3-0ubuntu1.20.04.1	1
coreutils	8.30-3ubuntu2	1

Table A.3: No. of vulnerabilities detected by Grype in `anibali/pytorch:1.10.2-cuda11.3` for each package

Package	Version	No. of Vulns
libexpat1	2.2.9-1build1	15
libc-bin	2.31-0ubuntu9.2	11
libc6	2.31-0ubuntu9.2	11
Pillow	8.4.0	5
libsepol1	3.0-1	4
libsqlite3-0	3.31.1-4ubuntu0.2	4
openssh-client	1:8.2p1-4ubuntu0.4	3
libpcre3	2:8.39-12build1	3
libkrb5support0	1.17-6ubuntu4.1	2
git-man	1:2.25.1-1ubuntu3.2	2
git	1:2.25.1-1ubuntu3.2	2
util-linux	2.34-0.1ubuntu9.1	2
bsdutils	1:2.34-0.1ubuntu9.1	2
libkrb5-3	1.17-6ubuntu4.1	2
fdisk	2.34-0.1ubuntu9.1	2
libk5crypto3	1.17-6ubuntu4.1	2
libblkid1	2.34-0.1ubuntu9.1	2
krb5-locales	1.17-6ubuntu4.1	2

mount	2.34-0.1ubuntu9.1	2
libgssapi-krb5-2	1.17-6ubuntu4.1	2
libmount1	2.34-0.1ubuntu9.1	2
patch	2.7.6-6	2
libuuid1	2.34-0.1ubuntu9.1	2
libfdisk1	2.34-0.1ubuntu9.1	2
libsmartcols1	2.34-0.1ubuntu9.1	2
login	1:4.8.1-1ubuntu5.20.04.1	1
numpy	1.21.2	1
libasn1-8-heimdal	7.7.0+dfsg-1ubuntu1	1
libsasl2-2	2.1.27+dfsg-2	1
gzip	1.10-0ubuntu4	1
libwind0-heimdal	7.7.0+dfsg-1ubuntu1	1
perl	5.30.0-9ubuntu0.2	1
liblzma5	5.2.4-1ubuntu1	1
coreutils	8.30-3ubuntu2	1
libhx509-5-heimdal	7.7.0+dfsg-1ubuntu1	1
zlib1g	1:1.2.11.dfsg-2ubuntu1.2	1
perl-modules-5.30	5.30.0-9ubuntu0.2	1
libhcrypto4-heimdal	7.7.0+dfsg-1ubuntu1	1
openssl	1.1.1f-1ubuntu2.10	1
libsasl2-modules-db	2.1.27+dfsg-2	1
bash	5.0-6ubuntu1.1	1
passwd	1:4.8.1-1ubuntu5.20.04.1	1
tar	1.30+dfsg-7ubuntu0.20.04.1	1
libssl1.1	1.1.1f-1ubuntu2.10	1
libkrb5-26-heimdal	7.7.0+dfsg-1ubuntu1	1
libperl5.30	5.30.0-9ubuntu0.2	1
libheimntlm0-heimdal	7.7.0+dfsg-1ubuntu1	1
libroken18-heimdal	7.7.0+dfsg-1ubuntu1	1
libsystemd0	245.4-4ubuntu3.13	1
perl-base	5.30.0-9ubuntu0.2	1
libudev1	245.4-4ubuntu3.13	1
libheimbase1-heimdal	7.7.0+dfsg-1ubuntu1	1
libgmp10	2:6.2.0+dfsg-4	1
libgssapi3-heimdal	7.7.0+dfsg-1ubuntu1	1