CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Debloating Web Services

Master's thesis in Computer Systems and Networks

Badiuzzaman A B Iskhandar

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

Master's thesis 2022

Debloating Web Services

Badiuzzaman A B Iskhandar

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Debloating Web Services
Badiuzzaman A B Iskhandar

Supervisor: Ahmed Hassan, Chalmers Computer Science and Engineering
Examiner: Philipp Leitner, Chalmers 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 LaTeX
Gothenburg, Sweden 2022

Debloating Web Services
Badiuzzaman A B Iskhandar
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The era of digitalization has led to more reliance towards software especially in the form of web services. Throughout the years, the services get increasingly bloated to support new features and requirements. As a consequence, this results in large and bloated services making it slower, less secure, and consume more compute resources.

To address these bloats, we begin by analyzing the existence of bloats in some components of web services. This includes the front-end web interface, web servers, and software packaging using containers. Next, we selected the NginX web server, a JavaScript library, and three Docker container images (Python, NginX, and Node.js) to be analyzed and debloated. Then, we identified the debloating tools applicable for each component, and performs the debloating process. Our results show that we can reduce the size of the NginX web server by around 40%, improve NginX request-per-second performance by around 3%, reduce the bundle size of the JavaScript library by around 90%, reduce the image size of the container by around 80%, and reduce the debloated container image startup time by around 7%.

Keywords: Software Debloating, Web services

Acknowledgements

I would like to thank my supervisor, Ahmed Ali-Eldin Hassan, for his continuous support and guidance throughout the journey, especially during hard times. I would also like to thank the examiner, Philipp Leitner, for his many feedbacks and suggestions to improve the work. And also special thanks to friends who keep the motivation running.

Finally, thanks to my family, especially my parents, for their continuous moral support.

Badiuzzaman A B Iskhandar, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1

Introduction

1.1 Problem Statement

As society relies more on technology, with applications ranging from government services, health care, e-banking, and online education, so does the complexity of the underlying software that provides these services. These systems continue to grow to support more features and use cases, resulting in heavy and bloated web services. This is illustrated in Figure 1.1 where the webpage size has grown by more than four times (from 510 KB to 2100 KB) for the past ten years. A worthy example is CNN.com, a major news website in the US, with a whopping 8 MB[1].

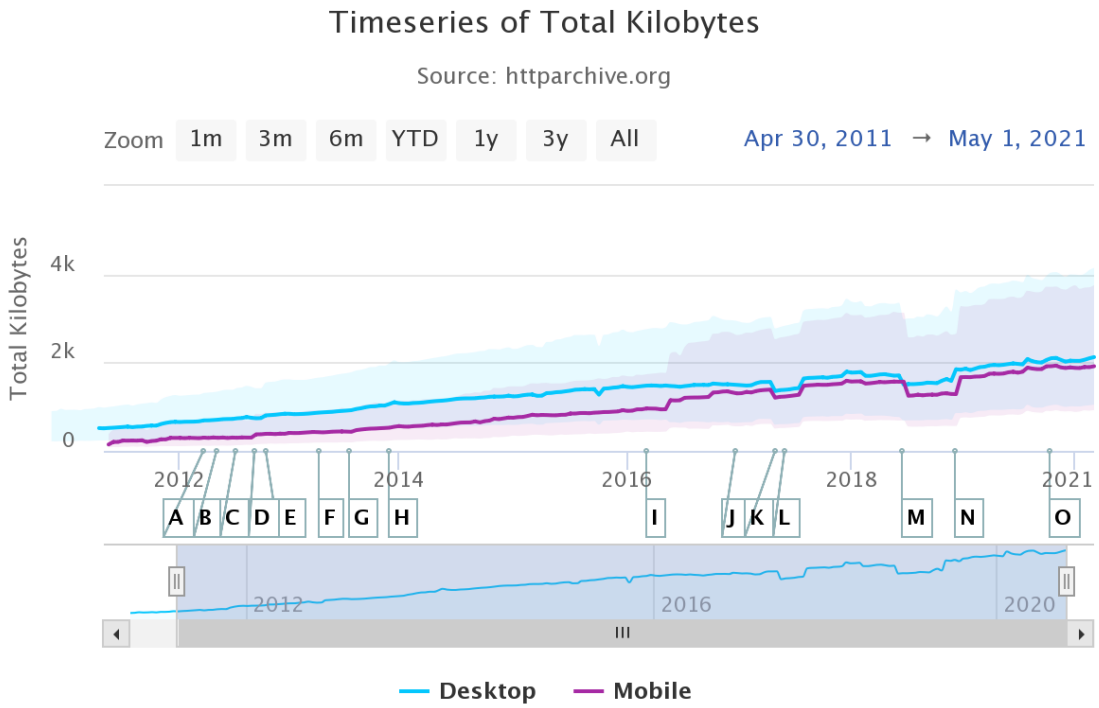

Figure 1.1: Webpage size (kilobytes) growth for the last decade [19].

[1]As listed in the Chrome DevTools

These bloated websites, or websites with many extra codes and unused functionalities, do not work well and load very slowly in areas with poor connection [22]. This is made worse for users using older devices with slower processors since to parse a webpage document, execute the big JavaScript codes, render the page, etc. are compute heavy operations in CPU [14]. Fortunately, this trend has not gone unnoticed; for example, Google provides better search engine scores for sites with faster speed.

Besides impacting end user experience, bloated software also consume more compute resources [3]. A lighter backend and web servers can help to alleviate such a problem. Another trend worth looking into is the advent of serverless and edge computing, where services are deployed and run on demand. In this case, we also need to optimize on how the services are packaged and deployed.

To that end, manually identifying all these bloats and fixing them one-by-one is a big undertaking, especially on legacy systems with millions of lines of code. Ideally, there should be tools that can automatically detect, diagnose, and provide remedies for such bloats. These tools can also be tied as part of the development lifecycle to detect problems early on, such as a linter or a memory leak checker.

In this work, we identify the bloats on different parts of a web service, find suitable debloating tools, and evaluate its benefits.

1.2 Aim

In recent years, there have been various works done to address bloats in various parts of the software stacks – from programming languages, runtime environments, and operating system kernel. In this project, we focus on debloating techniques relevant for web services.

Thus, the aim of this thesis is to explore debloating tools suitable for web services. We would like to understand the benefits and trade-offs of using such tools and present useful insights for others working in this area. We decompose a web service into three components: web server, JavaScript, and software packaging/deployment via containerization.

Following are the research questions for this project:

1. RQ1: Are there bloats in web services?

2. RQ2: Are there existing debloating researches and tools suitable for web services?

3. RQ3: What are the benefits for debloated components in terms of storage reduction and performance gain?

1.3 Contributions

The main contribution of this thesis is that we evaluate different debloating techniques suitable for web services. In particular, we select debloating techniques suitable for the web server, front-end (client-side JavaScript), and software packaging/deployment via containerization. Some of these tools are not actively developed, thus requiring technical work to enable it. After successfully performing the debloating process on the web service, we measure the improvements and describe possible trade-offs due to it.

1.4 Thesis Outline

The rest of the thesis is as follows. In Chapter 2, we discuss the relevant concepts and theories behind software debloating. Chapter 3 describes other related works for this thesis. Chapter 4 describes the methods we use to answer our RQs and how we evaluate them. Chapter 4 presents the result and the discussion after applying the debloating processes. Lastly, in Chapter 5, we present our conclusion and possible future work.

2

Technical Background

In this chapter, we provide an overview of the concepts behind software bloats and its debloating process. This chapter is split into two parts: code bloat and software packaging bloat via container.

2.1 Application-Level Bloat

Modern software development is rarely done from scratch. To save development time and cost, it is common for a software project to use libraries and frameworks that solve recurring problems such as user interface design, user authentication, cryptography, and networking. However, these libraries typically include more than what is actually needed by the depending application. Additionally, these libraries may depend on other libraries (transitive dependencies) which themselves can be bloated as well. In addition, the application itself may also deliver various different features that are not used by a common end-user.

To illustrate this, Figure 2.1 shows an example of a bloated application. The users only need to use some features of the application. Moreover, the application itself relies on other bloated libraries and frameworks, thus resulting in even more bloat. This bloated software may result in an increased attack surface [10] as well as lesser performance[5].

In the following subsections, we discuss the phenomenon of code bloat and its solution in software projects written in JavaScript and C. Both programming languages have different causes that lead to bloat since both of them fall in different criteria. For example, JavaScript is a dynamically typed language whereas C is a statically typed language.

2.1.1 Bloat in C Program

We first describe bloats within the main application code itself. Typically, the main application code is the portion of a code that calls other external library functions, manipulates data structures, presents output to the user, etc. This may also be called a driver code. Next, we describe bloats from external libraries, where the main

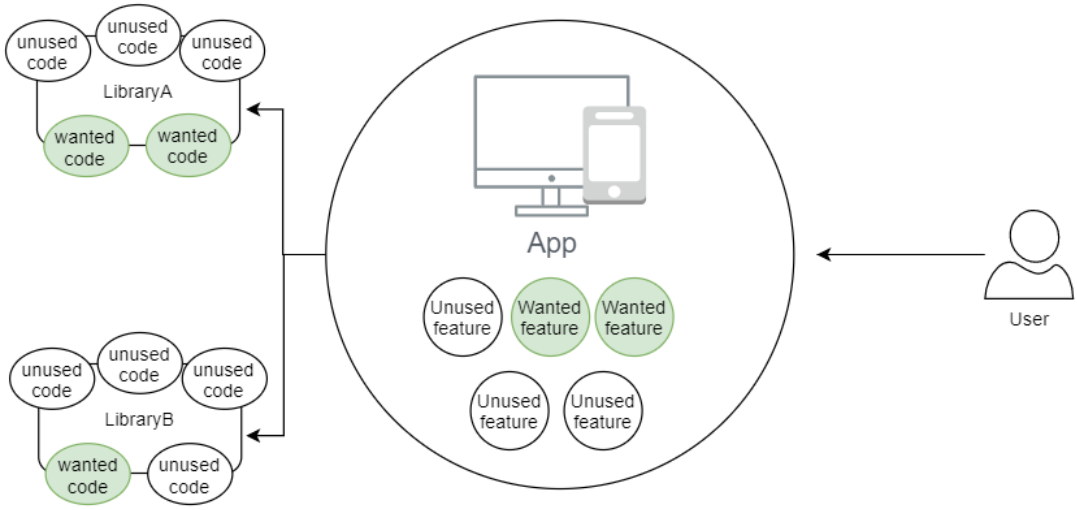

Figure 2.1: Bloated application and bloated dependencies.

application code imports as part of its dependencies.

By way of example, Listing 1 shows a sample C program where both the function `unused` and the global variable `unused_var` are not used and can be safely removed. After compilation, global variables are placed in the data segment, while functions are placed in the text section (Figure 2.2). Removing both of these unused codes helps to reduce the resulting binary size. It can be done during the compilation process; for example, with `gcc`, a compilation flag can be added to give warnings (or errors) when the compiler detects unused variables or functions. The programmer can then remove or refactor the code accordingly.

Next, we observe the external library calls: `printf` from the C standard IO library (`stdio.h`), and `some_func_from_somelib()` from `somelib.h`. Here, we only use a single function from `stdio.h` and `somelib.h`, but including both libraries gives us all other functions as well.

Finally, we observe a function call made using a function pointer; that is, we call `my_func_pointer` through the function pointer `my_func_ptr_var`. To statically detect function calls made through function pointers is a hard problem [2]. An aggressive debloating tool might remove functions that are actually called through a pointer, thus leading to errors.

2.1.2 Bloat in JavaScript Programs

Client-side JavaScript, running in a user's web browser, is used to provide richer behavior to a web page alongside HTML and CSS. On a big project, it is a common to separate the logics and functionalities into different modules or libraries. All these source files and its dependencies are then bundled together into a single file.

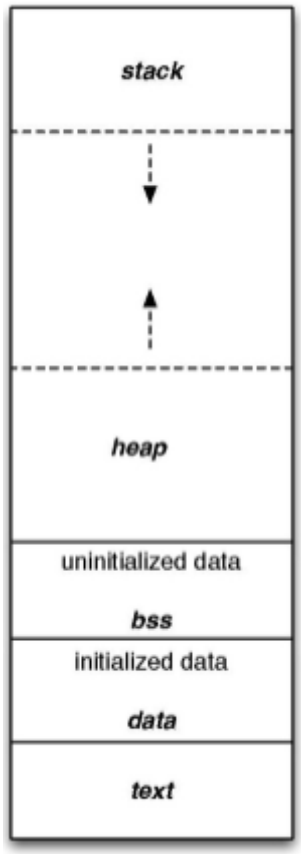

Figure 2.2: Program memory layout [4]

This file is then passed to the user upon server request. Since JavaScript needs to be downloaded by the user, having a large code can lead to poor user experience, especially for user with slower connection speed.

Similar to C, JavaScript also has the unused variables and functions symptoms described in the previous section. In particular, bloat due to external libraries is even more prominent in JavaScript; a phenomenon known in the JavaScript community as *dependency hell* [17]. One of the reason this happens is due to the lack of a standard library in JavaScript, resulting in developers needing to import lots of external libraries. There is an effort to introduce a standard library by TC39, the JavaScript programming language steering committee. At the time of this is writing, the proposal is still under discussion [20].

To illustrate this large dependency problem, we present Figure 2.3[1]. This figure shows the partial dependency graph for `mathjs`, a JavaScript library that supports various advanced mathematical operations.

To get a sense of how much storage does these dependencies consume, we can check the size of the `node_modules` folder:

[1]Using NPM Graph: https://npmgraph.js.org/?q=mathjs

```c
#include <stdio.h>
#include "somelib.h"

int unused_var = 1;

void used() {
    printf("used function\n");
}

void unused() {
    printf("unused function\n");
}

void my_func_pointer(char* str_in) {
    printf("Hello %s from my_func_pointer\n", str_in);
}

int main() {
    void (*my_func_ptr_var)(char *) = my_func_pointer;

    used();
    some_func_from_somelib(123); // defined in somelib.h
    my_func_ptr_var("there");

    return 0;
}
```

Listing 1: Unused code example in C

```
$ du -sh node_modules/
220M    node_modules/
```

The bad news is that the size is not that big at all, considering that the average size of the `node_modules` folder for the top 100 Node.js packages is around 700 MB[2].

2.2 Container-Level Bloat

Container or operating system (OS) virtualization is a lightweight virtualization where multiple isolated instances share the same operating system. This is in contrast to hardware virtualization where each instance has its own operating system (Figure 2.4). To achieve isolation while sharing the same OS, the OS kernel needs to provide both resource consumption limitations and resource isolation similar to

[2]https://github.com/vnglst/size-of-npm/

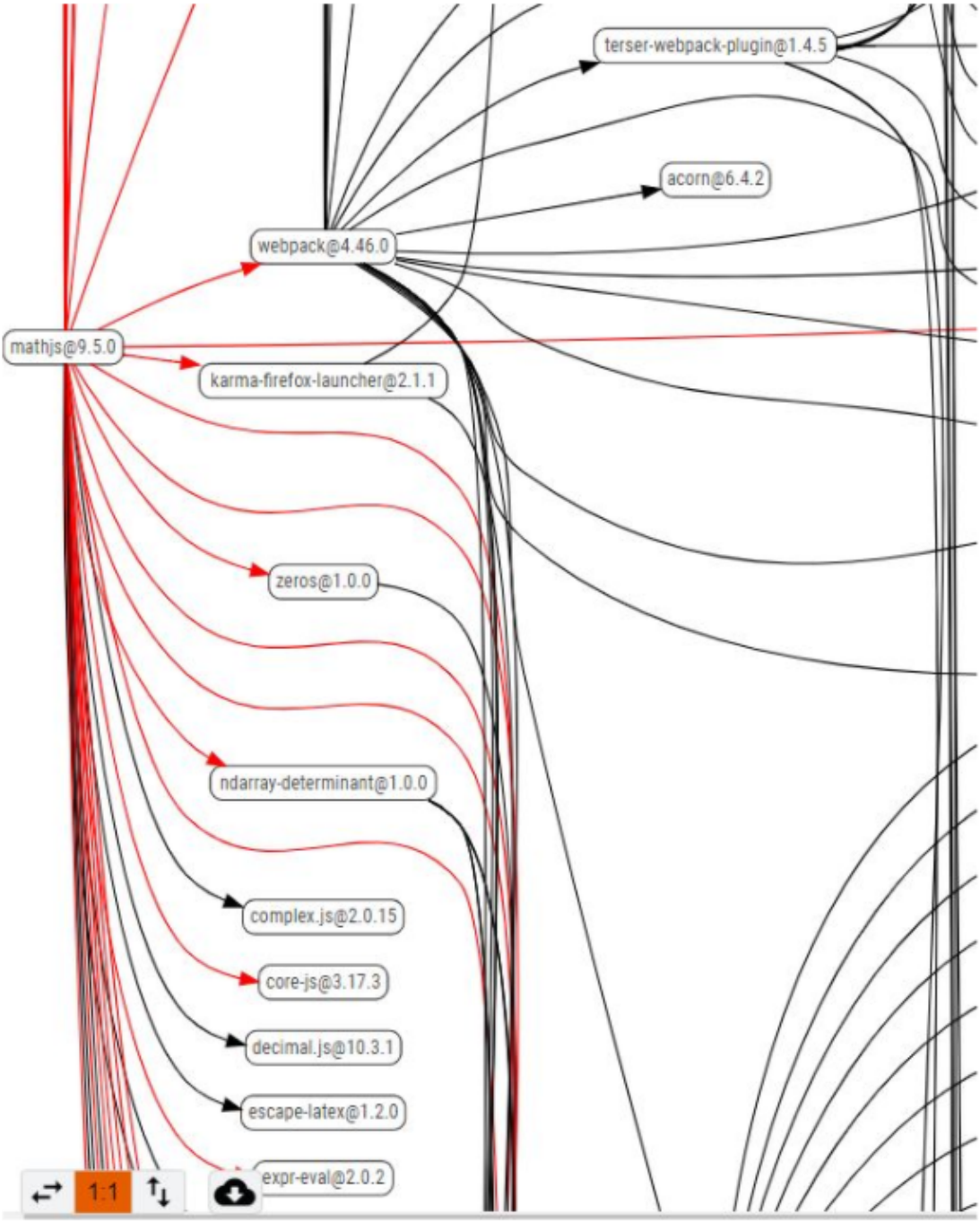

Figure 2.3: `mathjs` dependency graph(cropped)

what is provided in hardware virtualization. The implementation depends on the OS developer. The Linux kernel, for example, limits resource consumption (e.g., CPU, memory, and disk) via the control group (cgroup). Resource isolation is implemented through the namespace [16]. The following are some namespaces and the corresponding isolated resources:

- Process ID namespace: isolate the process list between containers

- User namespace: isolate user IDs and group IDs
- Network namespace: isolate network interfaces

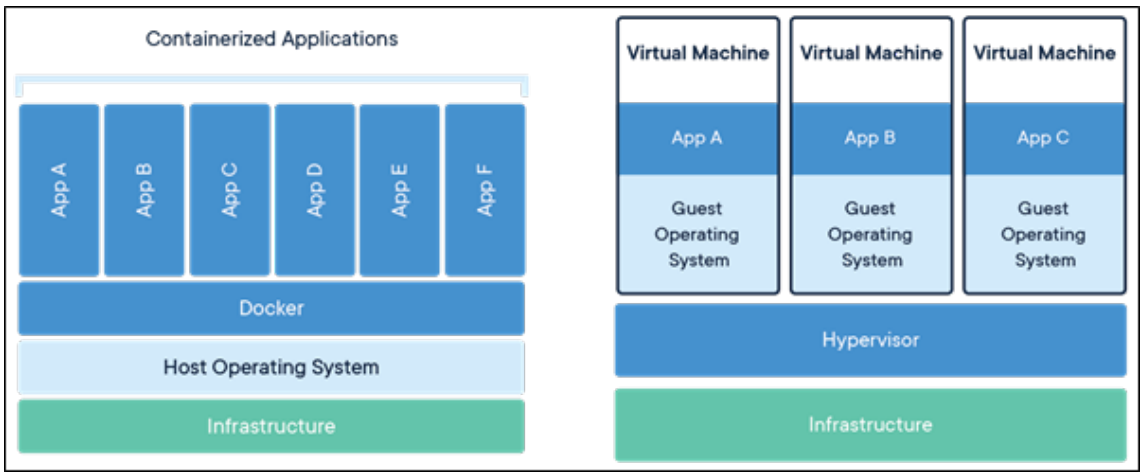

Figure 2.4: left: OS virtualization; right: hardware virtualization [23]

To package and deploy an application inside a container, developers start from a base image and add other extra dependencies required to run their application. Bloat in containers occurs when it has more dependencies and libraries than what is required to run the application. Most of the time, the base image itself contributes the most to the bloat.

It is relevant to study as the bloats can be large, e.g., Rastogi et al. [7] reported that the container occupied 20 times more disk space than what is required to run the application without any issue. They propose a technique where the application running in the container is instrumented to find out the exact files needed for it to run properly. All other files can then be safely removed since they were not used by the application. DockerSlim[3] also employs a similar technique to remove container bloats. We describe both tools in more detail in Section 3.

One disadvantage of a large container image size is the potential security issue. Any dependency can be a possible attack vector. Another disadvantage is on container startup time. To start a container, the image must be present in the machine before proceeding to unpack and run it. If the image is not present, it needs to be pulled (downloaded) from some image provider.

The problem becomes worse if the container is used in an edge or serverless computing environment. For example, a service might be on-demand or ephemeral, where it is only started upon request as opposed to a long-running service. As a consequence, the container startup time is no longer an amortized cost; rather, it is inflicted on every service instantiation.

[3]https://github.com/docker-slim/docker-slim/

3

Related Work

There exists a large body of existing researches on software debloating tools. From executable binaries, operating systems, to techniques specific to a certain programming language. In this thesis, we are interested in tools that can be used for web services. Generally, it depends on which programming language is used for that particular components. For example, a web servers may be written in C, JavaScript, PHP, or Java.

In the following sections, we describe how these tools can be used for each categories: web servers, front-end JavaScript, or containers.

3.1 Google Closure Compiler

Google Closure Compiler[1] is a source-to-source compiler for JavaScript. That is, it takes JavaScript source codes and optimize it by removing unused codes and minimize it. This results in much smaller code size, thus reducing the bandwidth required to transfer the code to the client side.

3.2 JavaScript Tree Shaking

Tree shaking[2], in the context of JavaScript, is a concept used to remove unused (dead) code. ECMAScript 2015 introduces a module with the `import/export` syntax. With this, developers can explicitly import a subset of functions exported by an external library and the bundler can then remove other unused functions, resulting in a smaller bundle size.

3.3 UFFRemover

A JavaScript package consists of one or more modules bundled together, where each module can implement a smaller portion of the application logic. Developer can

[1] https://developers.google.com/closure/compiler
[2] https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking

import these packages as dependencies for their application. UFFRemover [12] is a JavaScript debloating tool used to remove unused modules and functions from imported modules. This is relatively similar to the Tree Shaking technique, but with more aggressive bloating. That is, the Tree Shaking technique only supports removing functions and variables that are explicitly exported by a module through the `export` syntax, whereas UFFRemover is able to remove unused functions and variables that are not exported by dynamically analyzing the execution paths during runtime.

3.4 ModClean

ModClean[3] is used to remove unused files bundled in JavaScript packages. For example, it may delete documentations, test codes, and configuration files that is not used by the program.

3.5 Less is More

In this work, Babak et al. [10] presents the benefits of debloating PHP applications, in terms of both security and reduced line-of-codes. In particular, they run it on PhpMyAdmin, MediaWiki, Magento, and WordPress. These applications provide many features where the logics are implemented in some files and functions. To identify which files and functions are used, they profile the application while running monkey testing (automated clicking on the web UI) and web crawling. Afterwards, they can remove the files and functions that are not touched during the profiling.

As a result, they are able to: (1) reduce the line-of-codes, on average, by 33.1%; and (2) reduce CVEs by 38%.

3.6 JShrink

In JShrink, Bruce and Zhang et al. [13] developed a debloating framework targeting the Java programming language. A big challenge in Java is its support for advanced dynamic language features like reflection, dynamic class loading, dynamic proxy classes, etc that makes code reachability analysis difficult. In fact, whether a code is executed may only be known during runtime, based on some user inputs or configuration. Thus, automated debloating must be done with care so that the debloated applications would not crash during runtime.

To achieve this, the authors use both static and dynamic language analysis feature. Some notables debloating they can do are: unused field removal, method inlining, and class hierarchy collapsing. To ensure the original application's behavior, they also implement checkpointing. That is, if a debloating transformation causes some tests to fail, then the transformation operation is reverted. As a result, they are

[3]https://www.npmjs.com/package/modclean

able to: (1) reduce projects size by up to 46.8% (average of 14.2%); and (2) better behavior preservation (reduced test failures by 98% compared to Jax, and by 84% compared to ProGuard).

3.7 Object Culling and Concretization for Assurance Maximization (OCCAM)

OCCAM [6] is a tool for C/C++ that can debloat both the main application and its shared libraries by statically analyzing the LLVM bitcode. The authors describe the process as winnowing – a static analysis and code specialization technique using partial evaluation. The steps to *winnow* an application are as follows:

1. The application and its shared libraries are compiled to LLVM bitcode format.

2. Partial evaluation: The code is analyzed by getting compile-time constants and its control flow to help find unreachable codes. For example, a function call supplied with the same function arguments (e.g., control flag) can be trimmed by removing unreachable codes that rely on such arguments.

3. Remove unused variables and functions detected after the partial evaluation step.

4. Link back the winnowed application and its shared libraries.

3.8 Nibbler

Unlike OCCAM that operates and analyzes LLVM bitcode, Nibbler [9] works by disassembling binary programs to detect function calls and calculate the call graphs between the main application and its shared libraries. Function calls are classified into three categories:

1. **Function calls to locally defined function.** In disassembled code, local function calls are represented by the `CALL` and `JMP` instructions.

2. **Function calls to shared libraries.** Resolved by linker during runtime. In Linux, calls to external functions are performed through Procedure Linkage Table (PLT) and each external function has a PLT entry in the binary. Upon calling these external functions, the dynamic linker resolves the symbols and get the shared libraries that implement the functions in the system. For example, a call to `libc`'s `printf` will load the `printf` function located in `/lib/x86_64-linux-gnu/libc.so.6`

3. **Function calls via pointers.** Similar to call to locally defined functions, function calls via pointers correspond to the `CALL` and `JMP` instruction. To

determine whether a function is called with static analysis is hard and error-prone, therefore, authors opt to over-approximate and include all possible functions.

After parsing the binaries based on the function calls above, a function call graph is constructed. Then, it can detect and remove the unused functions.

3.9 Lightweight Multi-Stage Compiler-Assisted Application Specialization (LMCAS)

Similar to OCCAM, LMCAS [15] utilizes the LLVM compiler infrastructure to analyze the generated LLVM bitcode and further transform and optimize the programs during compilation stage.

An application typically consists of configuration logic and main logic to drive the program. LMCAS uses these two concepts as input to its debloating process. The authors described the boundary between these two logics as a *neck*. Once the necks (or boundary) between these two logics are detected, LMCAS can then use the knowledge learned from the configuration logic to optimize and specialize the main logic.

As a result, LMCAS is able to: (1) reduce a binary size by up to 25%; (2) reduce attack surface of code-reuse attacks by removing 51.7% of the total gadgets; and (3) eliminating 83% of known CVE vulnerabilities.

3.10 DockerSlim

DockerSlim[4] is a container debloating tool that removes unnecessary files (e.g., libraries and executables) from a container image. The bloats mostly come from files delivered as part of the base image. For example, a container built from Ubuntu base image[5] is at least 70 MB in size. This includes programs like GNU Coreutils which might not be needed by the main application running in the container, thus can be removed. To detect these unused files, DockerSlim uses both static and dynamic analysis to build a graph of the required files. Then, it creates a new smaller image based on only these required files preserving the application semantic.

DockerSlim is very effective in reducing the container image size. For example, a Python application image based on `ubuntu:14.04` image (438 MB) can be reduced to 16.8 MB (about 26 times reduction in size). Thus, it is a very effective debloating to reduce container image size. Some benefits of smaller image size include faster container deployment time in the cloud and faster testing in continuous integration (CI).

[4]https://github.com/docker-slim/docker-slim
[5]https://hub.docker.com/_/ubuntu

3.11 Cimplifier

Cimplifier [7] is another tool for debloating containers. Unlike DockerSlim, it can only do dynamic analysis on the executables in the container to detect what resources the executables used.

A novel feature compared to DockerSlim is that Cimplifier can do container partitioning. That is, splitting a container containing multiple executables into multiple container, placing each individual executable to the appropriate container. To facilitate the communication between these executables that live in different containers, the authors propose a mechanism called remote process execution (RPE) to glue these containers together. As a result, the tool produces a smaller container size with up to 95% size reduction.

4

Methods

In this section, we describe the methods and evaluation for debloating web services.

Figure 4.1 shows the high-level overview of our approach, with more details on the following sections.

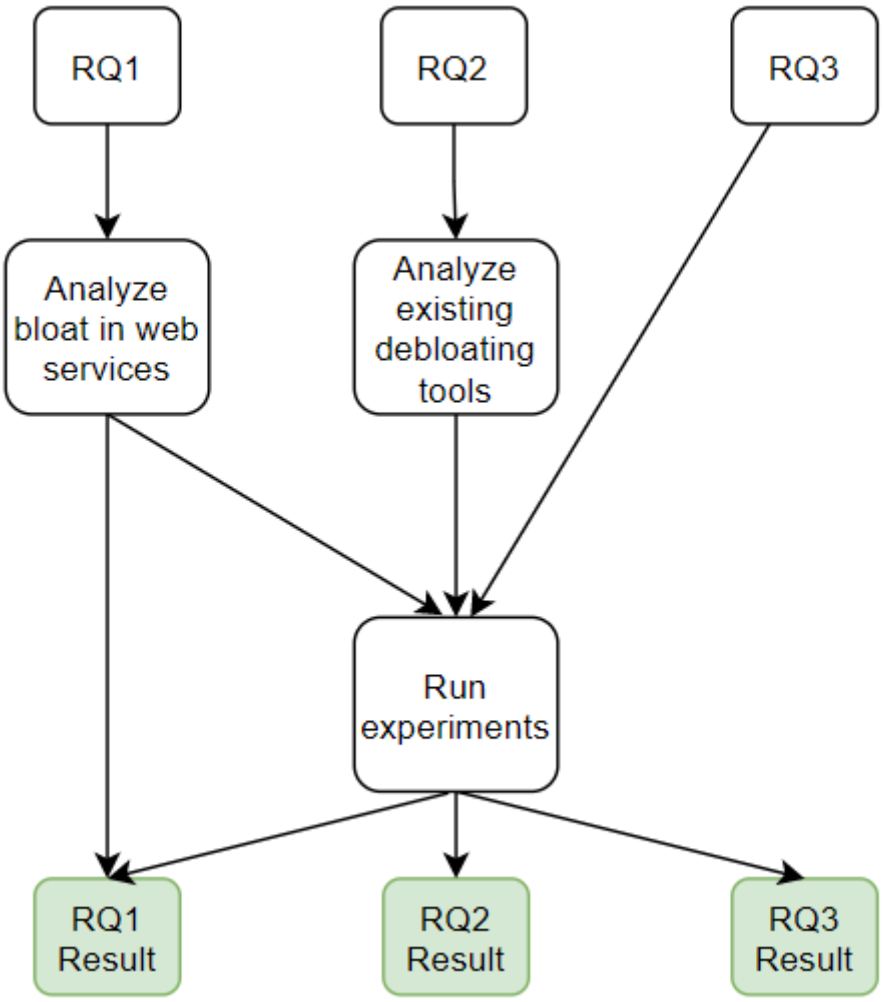

Figure 4.1: Overview of our approach in debloating web services.

We assume a web service that follows the multitier architecture [1] which consists of multiple parts: (1) the presentation layer (or commonly known as the front-end layer in web development) consisting of HTML, CSS, and JavaScript; (2) the domain layer that implements business rules and logics; and (3) the data layer for storing and managing persistent data for the application. Figure 4.2 illustrates this multitier system.

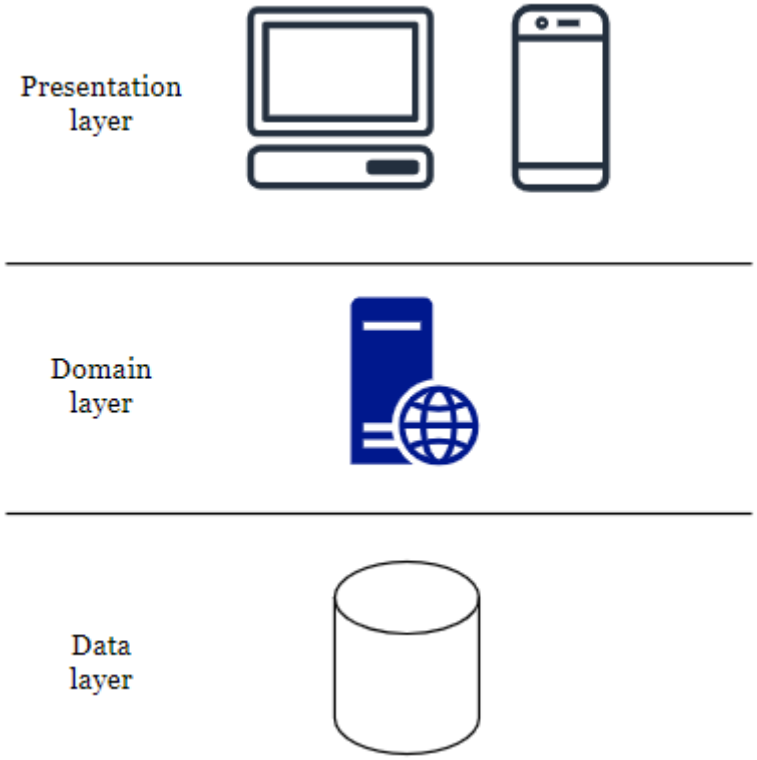

Figure 4.2: The Three-tier Architecture.

In this project, we limit our scope to the first two layers: the presentation layer and the domain layer. In particular, for the presentation layer, we assume a program written in JavaScript; and for the domain layer, we assume an HTTP web server environment.

Moreover, it is common for these services to be packaged and deployed in a container image. Therefore, we include containerization in our scope.

Before continuing further, for better clarity, we use other terms to refer to these different layers. For presentation layer, we refer to it as JavaScript since it is the common programming language used to implement user interfaces. For the domain layer, we refer to it as web servers.

4.1 Analyzing Bloat in Web Services

Figure 4.3 shows the overview of how we analyze bloat in web services.

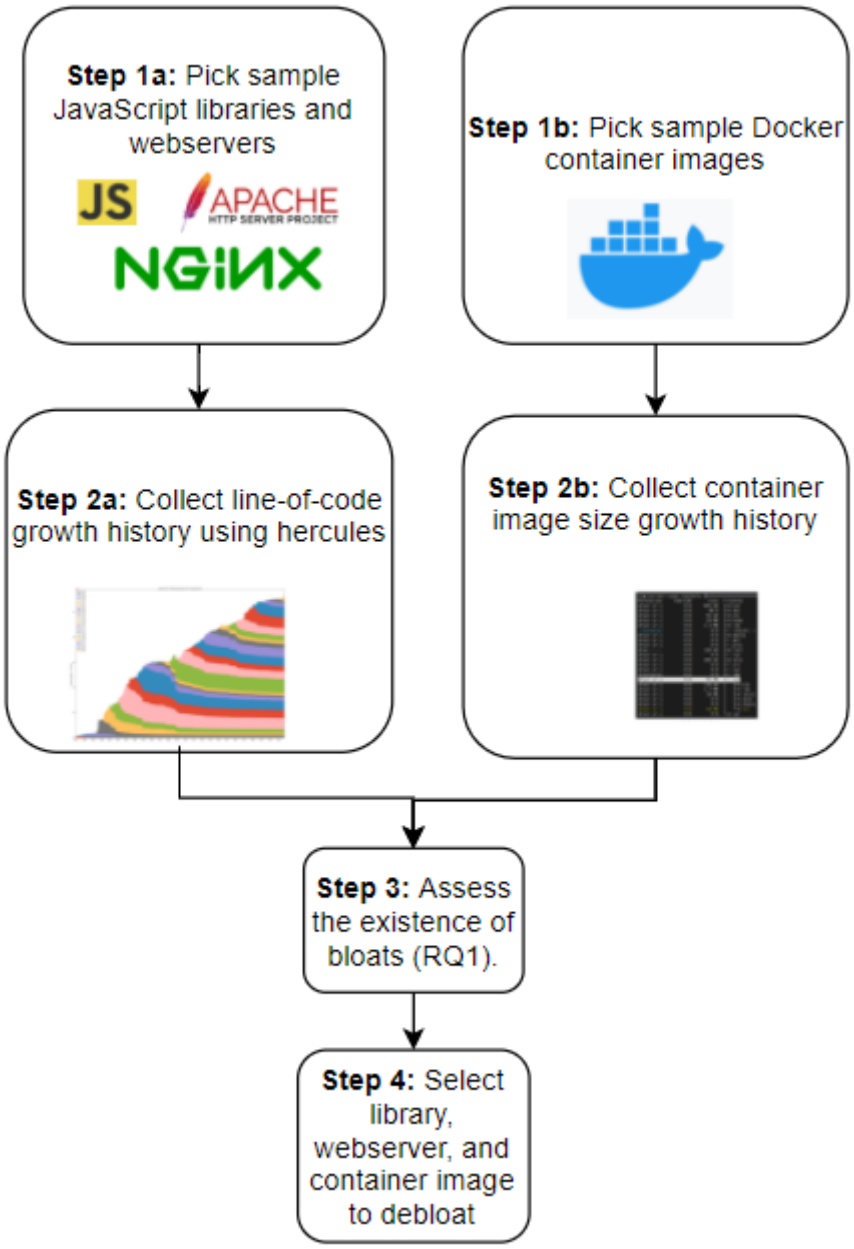

Figure 4.3: Analyzing bloats in different part of web services

Step 1 (both 1a and 1b): we make a selection from a sample of web servers, JavaScript libraries, and container images based on their popularity. For web servers, we choose Nginx[1] and Apache HTTPD[2], where both are used by more than 50% of websites, according to the Netcraft February 2021 Web Server Survey [18]. For JavaScript libraries, we choose the `date-fns`[3] (used for date manipulation) and `Math.js`[4] (used for advanced math functionalities). According to the Node Package Manager website (a package manager for the Node.js ecosystem), both are highly used libraries, with hundreds of thousands of weekly downloads. We also choose a reasonably sized Node.js application called `front-end` application [11]. For container images, we selected Python, Nginx, and Node.js, which are commonly used and have been downloaded billions of times according to the DockerHub website (a library for Docker images).

Step 2: in Step 2a, we use the `hercules`[5] program to measure the line-of-code growth rate. For container images (step 2b), we use the `docker image`[6] to display the size of container images.

Step 3: we then assess the bloats by looking at the line-of-code growth rate compared to the reduction rate. For container images, we look at the growth rate of the image size.

Step 4: finally, we select the web server, JavaScript library, and container images to debloat. We mainly base our selection on the complexity of the codes as well as ease of development; for example, even though both NginX and Apache HTTPD are widely used, we found that NginX is relatively easier to compile and run compared to Apache HTTPD.

4.2 Evaluation

Figure 4.4 shows the overview of the process of debloating and evaluating a component. In the following subsections, we describe the process in more detail for all three components: web servers, JavaScript libraries, and container images.

4.2.1 Webserver

We evaluate the web servers by measuring its throughput, i.e., how many HTTP requests it can handle per second. Since we are using web servers written in the C programming language, we also measure the amount of functions removed and the size of the executable binary.

[1] http://nginx.org/
[2] https://httpd.apache.org/
[3] https://github.com/date-fns/date-fns
[4] https://github.com/josdejong/mathjs
[5] https://github.com/src-d/hercules
[6] https://docs.docker.com/engine/reference/commandline/images/

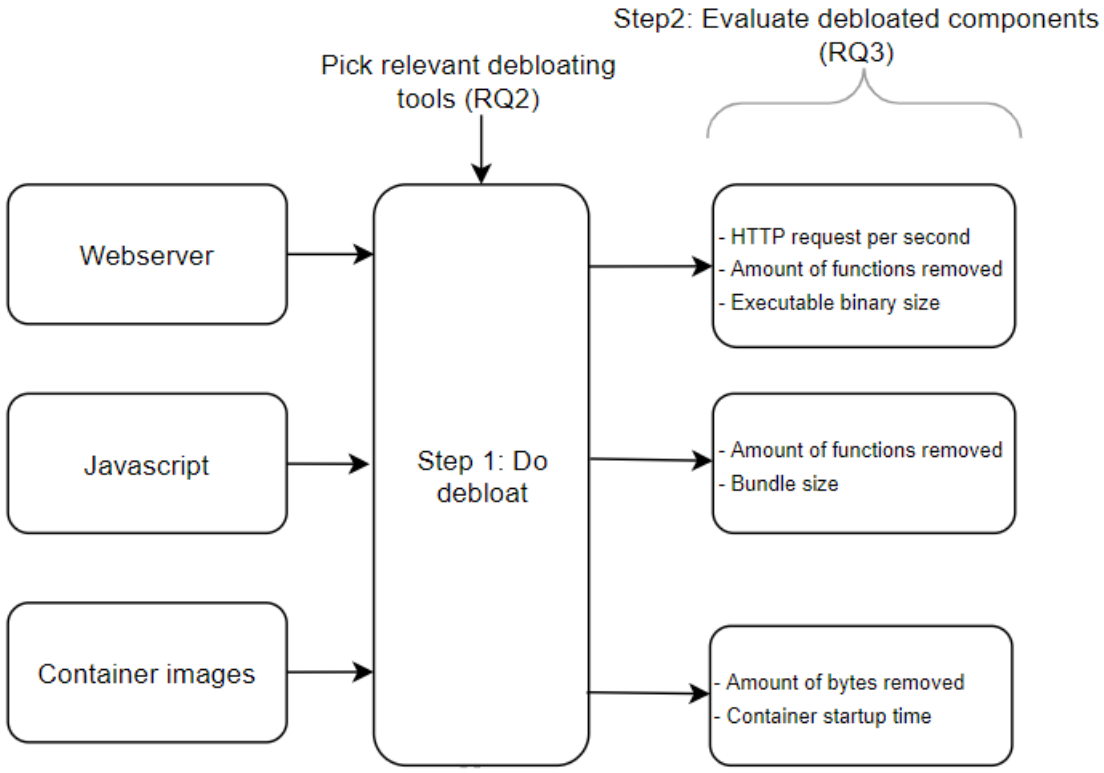

Figure 4.4: Debloat and evaluate components

4.2.1.1 HTTP request-per-second Benchmark

We evaluate the performance between the original NginX and the debloated versions by measuring how many requests per second it can handle. To do so, we use the Apache `ab` benchmarking tool[7] to perform 10000 requests to the NginX server with 100 concurrent clients.

4.2.1.2 Reduction rate for function counts and executable binary size

To analyze the compiled binary, we use various Linux command line tools described below to do binary analysis.

First, we use `size`, part of the GNU Binutils[8], command to calculate and list the section sizes of a binary. The size of a binary is determined by its $text + data + bss$ sections. To calculate a binary size's reduction rate, we use the following the formula:

$$reduction\% = \frac{original\_size - debloated\_size}{original\_size}$$

In addition to that, we also need to calculate the number of functions and bytes removed. We cannot find any existing tools that fit our needs. Some of the debloating tools do expose some useful statistics, but we need other kinds of data as well. For example, given a statically compiled executable binary and a shared library that it depends on, we want to find which functions in the shared library that are not in the binary. In the end, we rolled out our own solution by pipelining some binary analysis tools with Linux text processing utilities.

We illustrate this with a specific example. Suppose that we want to find which functions in the `libssl`(used for TLS) library that is not in a statically compile NginX binary that was compiled with `libssl`. To do this, we can use the following commands:

```
# Write a list of functions from nginx and
#   libssl to intermediate files.
$ objdump --syms libssl.so.1.1 | grep ' F .text' \
    | awk '{printf $6 "\n"}' > libssl_func_list.txt
$ objdump --syms nginx | grep ' F .text' \
    | awk '{printf $6 "\n"}' > nginx_func_list.txt

# Find a function in libssl that is not in nginx.
$ grep -v -Fxf nginx_func_list.txt libssl_func_list.txt \
    | wc -l
```

[7]https://httpd.apache.org/docs/2.4/programs/ab.html
[8]https://www.gnu.org/software/binutils/

Apart from that, we also want to know the total size (bytes) of these removed functions. Compared to before, it is a bit more complex. To achieve this, we use:

```
# Save the function symbol and its size to an intermediate file.
$ nm -S libssl.so | awk 'NF == 4 {printf $0 "\n"}' \
    > libssl_func_symbol_size.txt

# Find the list of functions NOT in the binary.
$ grep -v -Fxf nginx_func_list.txt libssl_func_list.txt \
    > libssl_not_func_list.txt

# Use the above two intermediate files and gawk
#   to sum up the byte size for all functions.
$ grep -f libssl_not_func_list.txt libssl_func_symbol_size.txt \
    | gawk '{print "0x" $2}' \ # Convert to hexadecimal format
    | gawk --non-decimal-data '{sum += $1;} END{ print sum;}'
```

4.2.2 JavaScript

A single bundle file is created by combining all JavaScript source code files. To get the file size of the bundle, we use `du`. To obtain the number of functions removed, we count the number of functions in the bundle minus the original function count.

4.2.3 Container

We use the `docker image` command to get the image size after debloat. To analyze the content of container's image layer (e.g., to check which files got removed), we use the `dive`[9] tool. This tool will unpack the image and display the filesystem of the image.

For container startup time, we use the Bash `time`[10] shell keyword when starting the container. To prevent errors due to disk caching, we purge the cache on every run with the command:

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Note that we do not include the image pull (download) time since this is very much dependent on the speed of the underlying network speed. This, however, is an essential aspect to consider for production deployment.

[9] https://github.com/wagoodman/dive
[10] https://www.gnu.org/software/bash/manual/html_node/Pipelines.html#index-time

4.3 System configuration

All of our work is performed on a machine with specification listed in the Table 4.1.

Operating Systems	Ubuntu 16.04.2 LTS, kernel 4.4.0-83-generic, x86_64
Processor	Intel(R) Xeon(R) CPU E5620 @2.40GHz
RAM	12 GB
Hard-disk	1 TB

Table 4.1: System configuration.

5

Results and Discussions

This chapter presents the results obtained from various debloating tools aimed at different level of a web service. We analyze the results from running debloating tools on programs written in C and JavaScript, and some container images.

5.1 The presence of code bloat in web services

In the following subsections, we show the existence of bloats in web server, JavaScript, and container images.

5.1.1 Bloat in web server code

We begin by analyzing the growth trend of NginX, a high performance HTTP web server implemented in C. Figure 5.1 shows the growth of NginX for almost two decades.

The growth trend is mostly due to the extra features and protocols it needs to support. Besides being used as a regular web server, NginX is also used as a reverse proxy server, a load balancer, and as a mail proxy server. Some of the supported protocols include TLS/SSL, email services(IMAP, POP3, SMTP, etc.), and the newer HTTP/2 protocol. Furthermore, looking into the git commit history to see some trends, it even provides native support for the GRPC protocol[1], a high performance RPC framework by Google.

Fortunately, many of these features and protocols are available as modules that can be included or excluded based on what a particular user wants to support. These modules can be compiled together as part of the final NginX executable binary (static module), or dynamically loaded (dynamic module) at runtime. For example, users that want the GRPC support can use the `ngx_http_grpc_module`[2] module.

Next, we analyzed the growth of another major web server, the open-source Apache HTTP Server Project. It predates NginX, and is considered as a more heavyweight

[1] https://grpc.io/

[2] http://nginx.org/en/docs/http/ngx_http_grpc_module.html

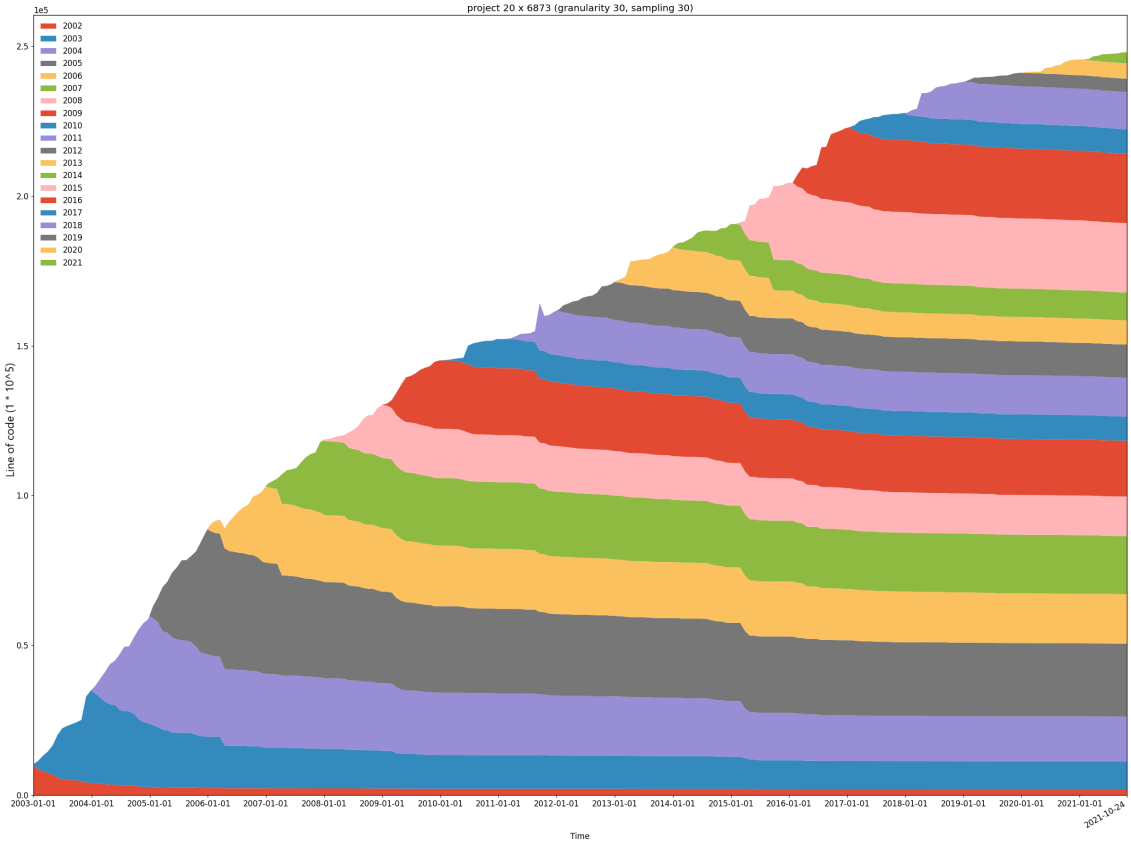

Figure 5.1: NginX line-of-code growth from 2003 to 2021.

web server compared to NginX. For example, it natively supports dynamic content and provides more advanced configuration settings. Figure 5.2 shows the growth of the project for more than two decades.

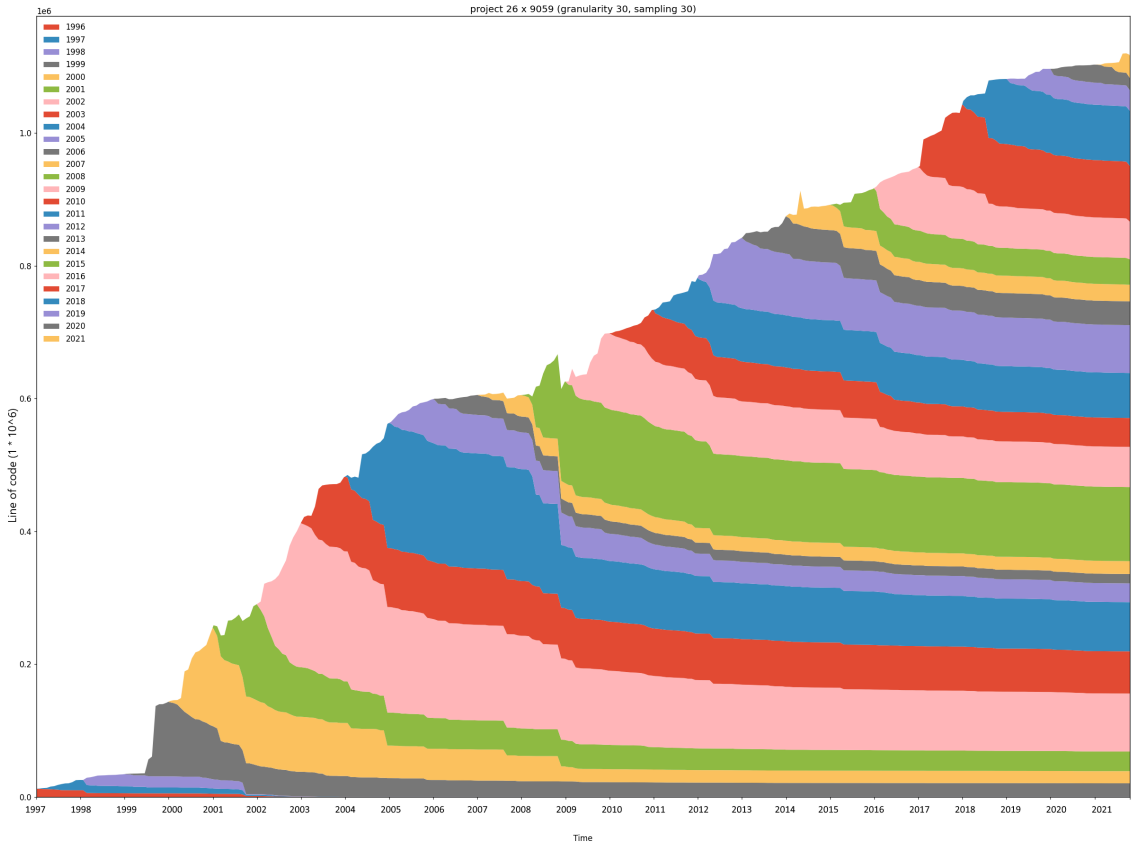

Figure 5.2: Apache HTTP Server line-of-code from 1997 to 2021.

The growth follows the same reasoning as NginX: new features, new protocols, and various other use-cases. Also, similar to NginX, Apache is also implemented in a modular architecture. For example, developers that wants to write dynamic content generation using PHP or Python can use the `mod_php` or `mod_wsgi`, respectively.

5.1.2 Bloat in JavaScript code

`Math.js`[3] is a JavaScript library used for advanced math functionalities, beyond what is supported by the built-in Math[4] library. Some of its feature includes parsing and evaluate mathematical expressions, and supports data types with various precision (e.g., 64-bits numbers and arbitrary-precision number). It also supports multiple browsers from Chrome, Firefox, Safari, Edge, and Internet Explorer 11. Figure 5.3 shows the growth of the library from 2014 to 2021.

Another widely used library is `date-fns`[5], with 11 million weekly downloads shown

[3]https://github.com/josdejong/mathjs
[4]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
[5]https://github.com/date-fns/date-fns

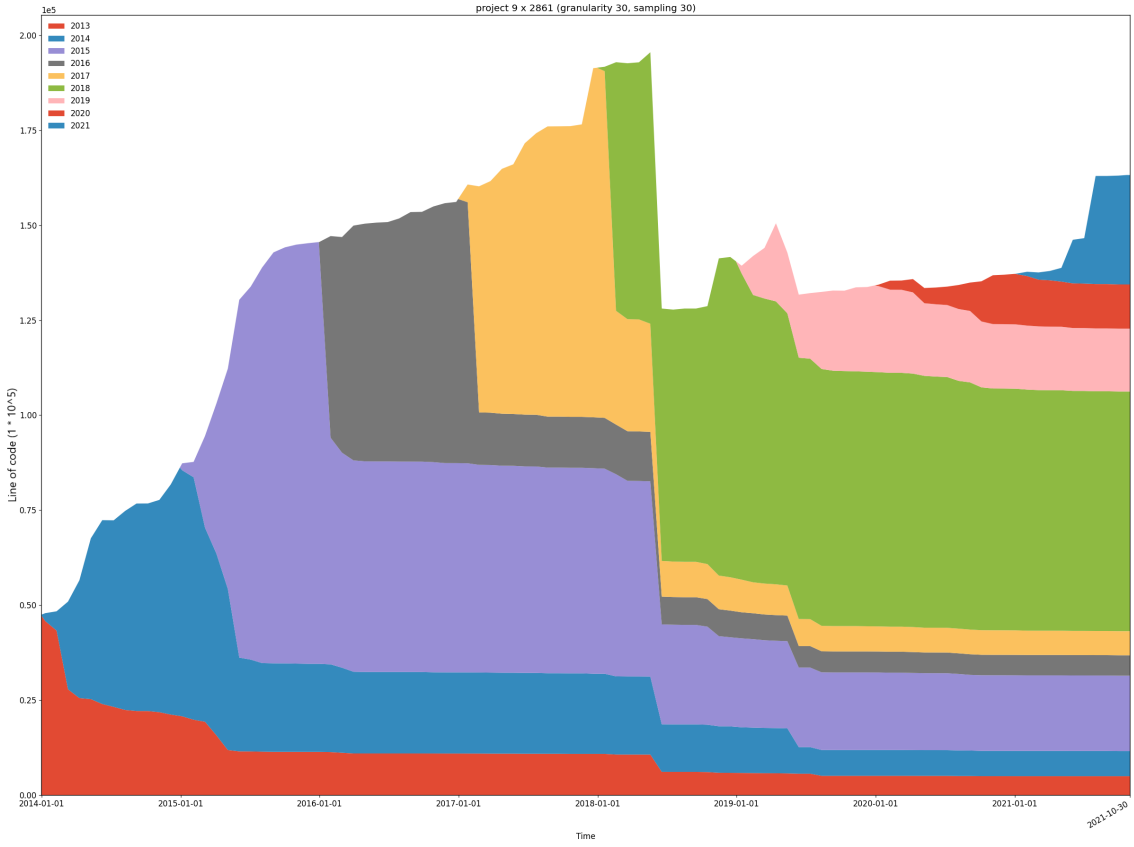

Figure 5.3: `Math.js` line-of-code growth from 2014 to 2021.

in the NPM package registry[6]. It is used to manipulate JavaScript dates both in browser or Node.js (server-side JavaScript). Figure 5.4 shows the code growth between 2015 and 2021.

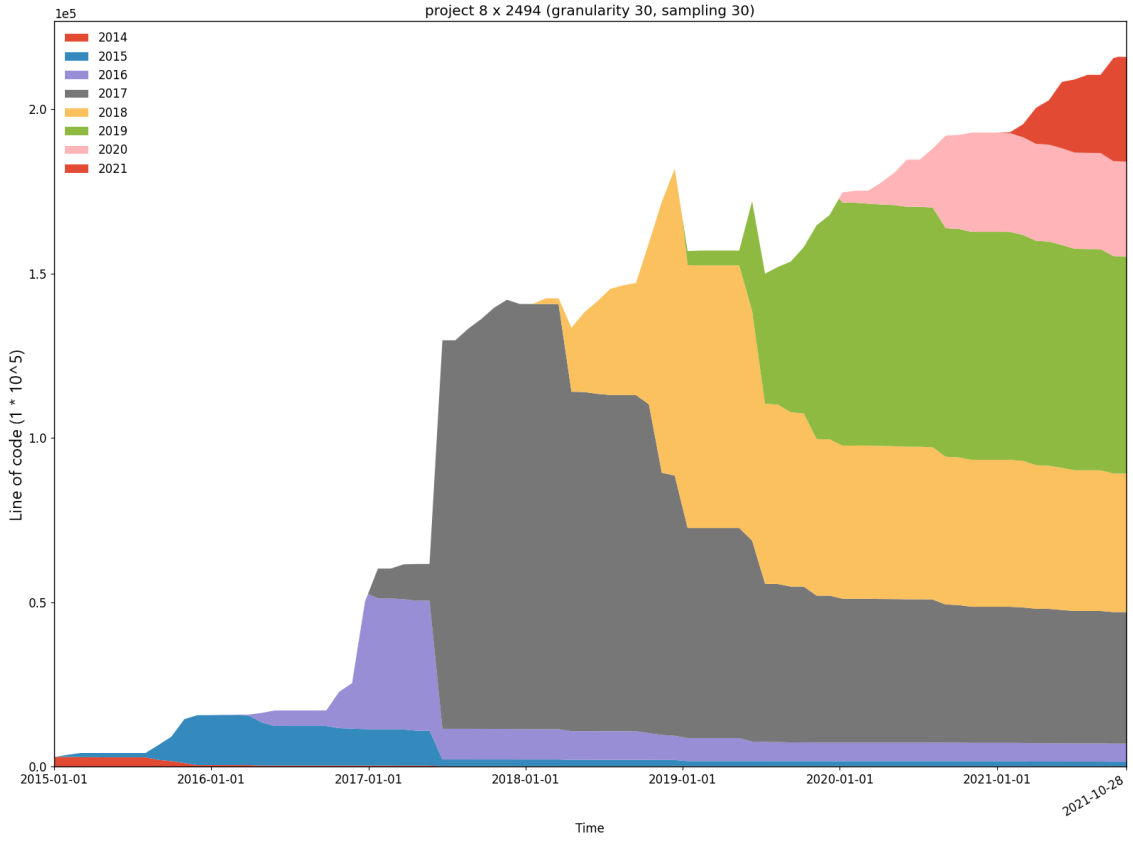

Figure 5.4: `date-fns` line-of-code growth from 2015 to 2021.

Fortunately, both of these libraries are built using the modern ES5 module. For example, in Math.js, it supports multiple type of numbers: 64-bits number, BigNumber (a number for arbitrary precision arithmetic), fractions, etc. Thus, a user who only needs the 64-bits number type does not need to import other unused number types.

5.1.3 Bloat in Docker container

Figure 5.5 shows the container image growth for Node.js, NginX, and Python images between 2014 and 2021. There are no big changes over the year, except for the Node.js image between the year 2018 (version 9.5.0) to 2019 (version 11.7.0), with a jump in size of around 135 MB. Upon further analysis, there are no big changes in the Node.js binary:

```
$ du -sh node-v9.5.0-linux-x64/ node-v11.7.0-linux-x64
70M     node-v9.5.0-linux-x64/
70M     node-v11.7.0-linux-x64
```

[6]https://www.npmjs.com/package/date-fns

Rather, the increase is due to one of the installed dependencies when creating the docker image. The layer size comparison for both versions can be compared from the hub.docker.com website.

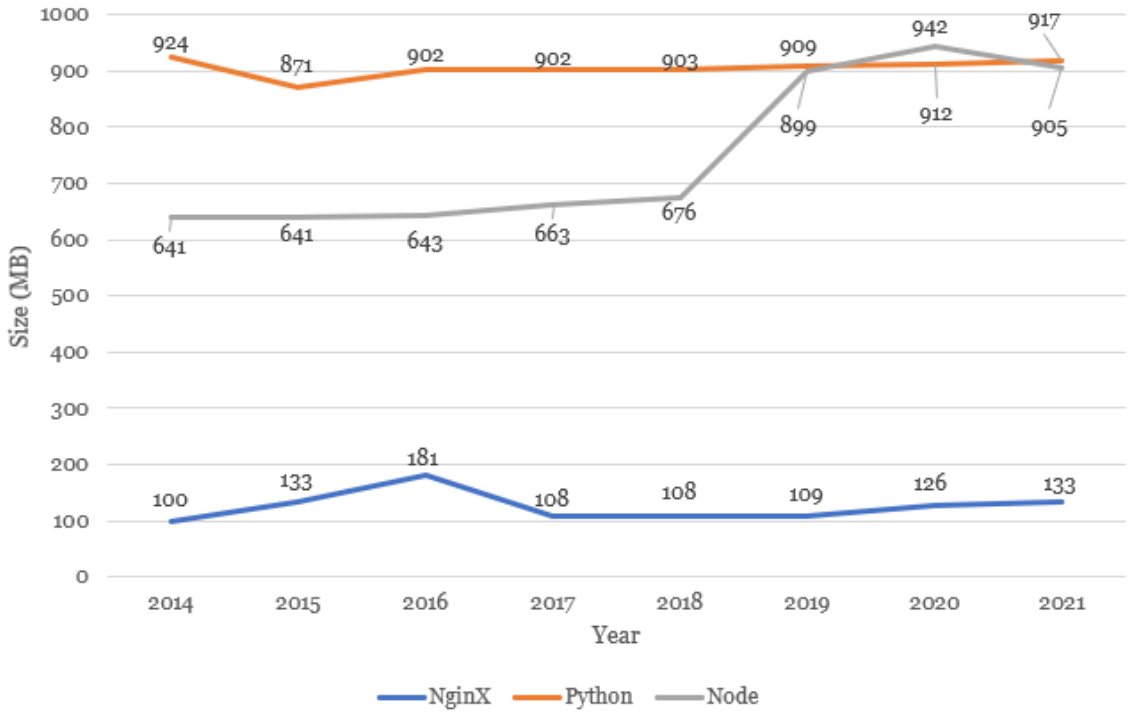

Figure 5.5: Container image growth for Node.js, NginX, and Python images

5.2 Evaluation on NginX Debloating

In Section 5.1.1, we have shown the code growth phenomenon on NginX and Apache HTTP. In this section, we pick NginX and evaluate the debloating using both OCCAM and Wholly tools. NginX debloating using Wholly has already been done before by Gelle et. al [8]; unfortunately, it still requires many fixes on the published code[7] due to changes in the dependent environment (Python, Ubuntu, etc).

On the other hand, NginX debloating on OCCAM is done from the ground-up. To the best of our knowledge, NginX has never been enabled on OCCAM before. We describe our experience on enabling it as well as evaluate the result.

5.2.1 OCCAM

OCCAM is able to debloat the main application as well as its library dependencies. This makes it suitable for a production-ready application like NginX which relies on many third-party libraries, as shown below:

[7]https://github.com/SRI-CSL/Wholly

```
# output simplified/truncated
$ ldd /usr/sbin/nginx
    linux-vdso.so.1 =>
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
    libcrypt.so.1 => /lib/x86_64-linux-gnu/libcrypt.so.1
    libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
    libssl.so.1.0.0 => /lib/x86_64-linux-gnu/libssl.so.1.0.0
    libcrypto.so.1.0.0 => /lib/x86_64-linux-gnu/libcrypto.so.1.0.0
    libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1
    libxml2.so.2 => /usr/lib/x86_64-linux-gnu/libxml2.so.2
    ...
```

The steps to use OCCAM are outlined as follows:

1. Generate LLVM bitcode for the target application and its dependent libraries.

2. Create an OCCAM manifest file, which contains, among other, the input flags (if any) for the executable binary and its dependent libraries.

3. Run the OCCAM tool with the supplied manifest file to get the debloated (or "slashed", in OCCAM terminology) application binary.

Step 1: Generating LLVM bitcode. The first step is to generate LLVM bitcode for the target application and its dependent libraries. In this case, it is the `nginx` binary together with some selected third-party libraries: `libc, libz, libpcre, libcrypto, libssl`. We compile all the source the code using LLVM clang compiler and used the `gllvm`[8] on the compiled binaries to extract its LLVM bitcodes.

Step 2: Creating OCCAM manifest file. The manifest file is used by OCCAM as a deployment context. It contains, among others, input flags (if any) to run the executable binary, environment variables, and library dependencies. The following manifest is used:

```
{ "main"        : "nginx.bc"
, "binary"      : "nginx_slashed"
, "modules"     : ["libpcre.shared.bc", "libz.shared.bc",
                   "libcrypto.so.bc", "libssl.so.bc", "libc.a.bc"]
, "native_libs" : ["-lpthread", "-lcrypt", "-ldl", "libc.a"]
, "static_args" : []
, "name"        : "nginx"
}
```

[8]https://github.com/SRI-CSL/gllvm

Step 3: Generating specialized binaries. `slash` is the front-end command line interface tool for running OCCAM. We use `slash` with the manifest file in the previous step to perform the core OCCAM's debloating algorithm:

```
$ slash --inter-spec-policy=onlyonce --intra-spec-policy=onlyonce \
        --stats --work-dir=slash nginx.manifest
```

Afterwards, it will output the resulting debloated binary.

5.2.2 Wholly

In Wholly, a package can be an application or a library. A *recipe.yml* contains the package's dependencies, URL for its source code, and the build steps needed to build the package. Figure 5.6 shows the partial NginX dependency graph.

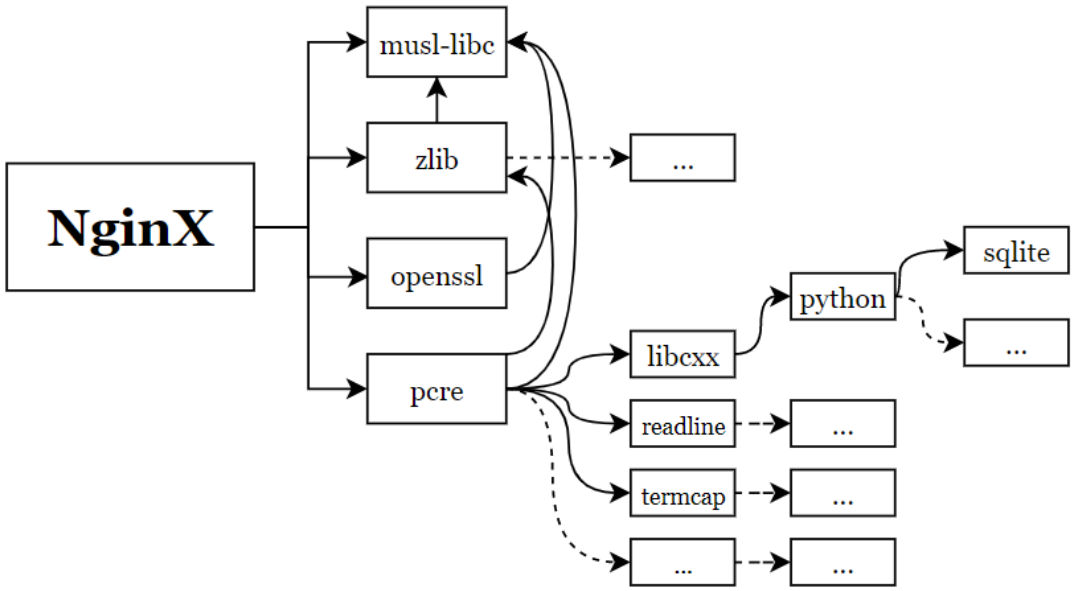

Figure 5.6: NginX dependency graph (partial)

Thus, to build the NginX package, we need to build all of its dependencies (total of 19 dependent packages). The packages' recipe works mostly as described with just some minor issues. For example, we need to update the links to download source code. Another issue is in compiling the Python runtime (CPython) package. One of the C file does not include the `sys/random.h` causing compilation error due to a missing preprocessor condition. We added the missing condition to fix the compilation error.

5.2.3 Debloated NginX Reduction Rate

After getting the generated NginX binaries from Wholly and OCCAM, we measure the reduction rate in terms of the number of functions removed, bytes removed, and the executable binary size. Table 5.1 and Table 5.2 shows the result. The bytes

Library	Baseline		NginX (OCCAM)		NginX (Wholly)	
	Number of functions	Size (bytes)	Function Removed	Bytes Removed	Functions Removed	Bytes Removed
libc	2139	566773	-	-	1398 (65%)	286565 (51%)
libz	117	81249	105 (90%)	48571 (60%)	81 (69%)	40934 (50%)
libpcre	50	92456	40 (80%)	81069 (88%)	25 (50%)	20813 (23%)
libcrypto	5216	1288522	4752 (91%)	946166 (73%)	1958 (38%)	340687 (26%)
libssl	866	253033	723 (83%)	184316 (73%)	358 (41%)	94329 (37%)

Table 5.1: NginX reduction rate (number of functions removed and bytes removed) between OCCAM vs Wholly. Percentage in parentheses shows the reduced amount.

Nginx Variants	**Binary size (Bytes)**
Nginx	5449450
Nginx (OCCAM)	3387929 (38%)
Nginx (Wholly)	3272476 (40%)

Table 5.2: NginX binary size. The percentage in parentheses shows the reduced amount compared to the original binary size (Nginx in the first row).

removed corresponds to the byte size for the removed functions. Note that the `libc` data for OCCAM is not available as the GNU `libc` is not supported yet for LLVM compiler. In general, OCCAM, which does LLVM bitcode analysis, is able to remove more codes compared to Wholly.

5.2.4 Debloated NginX benchmark

We benchmark the debloated NginX by calculating the average request per second (RPS) using the Apache benchmarking tool as described in the Methods section. Figure 5.7 shows the comparison between the RPS benchmark for the three variants of NginX: OCCAM, Wholly, and the original version.

Both OCCAM and Wholly show a slightly better performance. For Wholly, this is due to the static linking which reduces context switching [21]; Wholly author also reported similar findings [8]. For OCCAM, during the partial evaluation phase, it can produce a more optimized instructions; thus, resulting in better performance. OCCAM authors also reported a slight improvement on thttpd[9].

5.3 Evaluation on JavaScript Debloating

In this section, we present our results on debloating JavaScript application using the following debloating tools: Webpack, UFFRemover, Google Closure Compiler (GCC[10]), and ModClean. First, we present the result on a hypothetical application

[9]http://www.acme.com/software/thttpd/

[10]Not to be confused with the infamous GNU Compiler Collection.

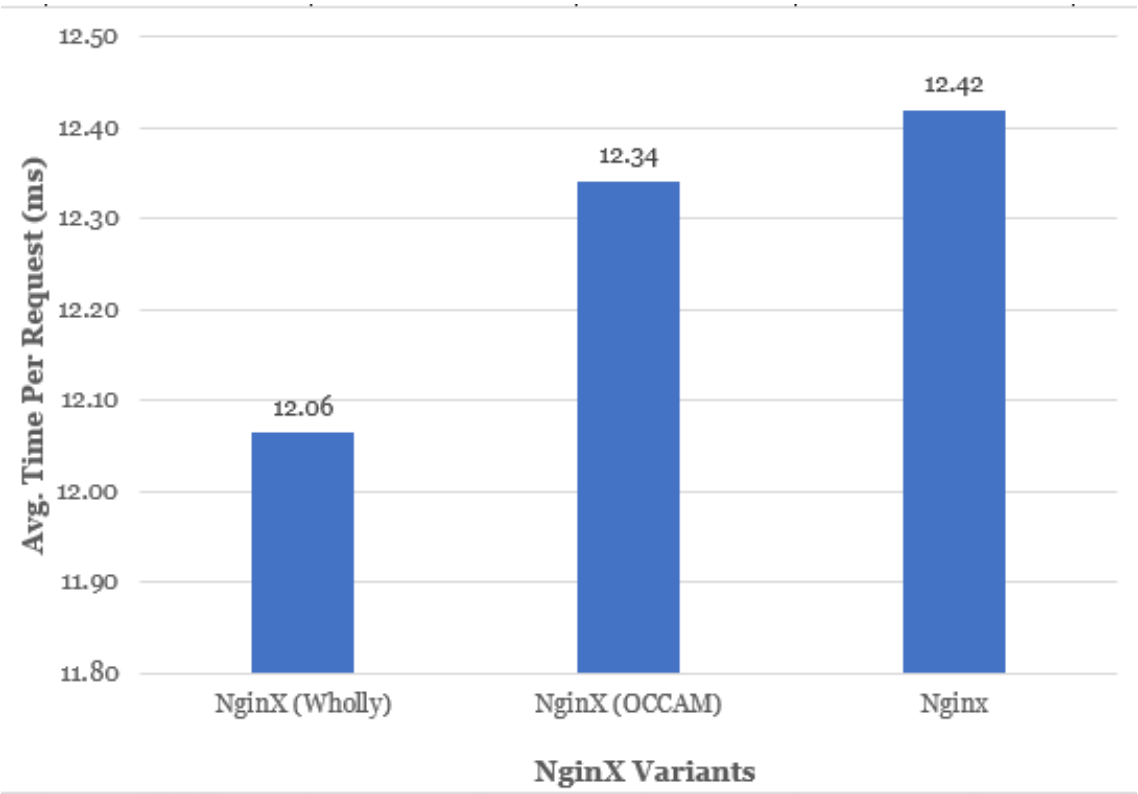

Figure 5.7: Benchmark for original NginX, NginX debloated with OCCAM, and NginX debloated with Wholly. Wholly is faster by about 3%, while OCCAM is faster by just 0.6%.

that uses the Math.js to do some calculations. Afterwards, we show the possibility of using some of these tools to reduce the size of `node_modules` as described in Section 2.1.2.

Math.js is a huge JavaScript library (1.6 MB) that supports many mathematical operations as well as evaluating mathematical expressions. An application may only need to use a portion of the library but ends up importing the whole library. Our goal, then, is to remove these unused codes as much as possible.

To demonstrate this, we created a sample JavaScript code that only uses the `fraction`, `add`, `divide`, `format` functions from Math.js. We then debloat it with Webpack, UFFRemover, and GCC. Table 5.3 shows the result.

Debloating Tools	Size (KB)	Reduction Rate (%)	Functions Removed	Reduction Rate (%)
UFFRemover	187	88.57	2969	95.56
Webpack	148	90.95	2957	95.17
UFFRemover + GCC	123	92.48	2969	95.56
Webpack + GCC	152	90.71	2957	95.17

Table 5.3: Debloated Math.js using UFFRemover, Webpack, and Google Closure Compiler.

An interesting observation to note is on how well the result of Webpack with tree-shaking enabled. It is almost similar to the UFFRemover which uses dynamic analysis. Tree-shaking is a recent development using the import syntax introduced in ECMAScript 2015. Other bundler like Parcel[11] still treats it as experimental.

In fact, we tried to use Parcel with Math.js but cannot make it to work since the bundled JavaScript file failed to load in the browser. Upon looking for the solution, we noticed other people are facing similar issue as well[12].

Since Parcel is still in active development, perhaps it will be more stable soon. Nonetheless, it is interesting to see the future of JavaScript's tree-shaking capabilities and how it will *shake* the issue of fast-growing JavaScript codes.

Next, we look into reducing the size of `node_modules` folder. GCC is a perfect option since it can be used to optimize and minimize regular JavaScript code. We demonstrate this using the `front-end` Node.js application.

Figure 5.8 shows the result. Using both ModClean and GCC, we are able to reduce the `node_modules` size by 42%. We also use the ModClean tool to remove extra files. Here, we can see another example of combining various tools to achieve a better result. Each tool solves different symptoms of bloat and can be composed into a pipeline of operations.

[11]https://github.com/parcel-bundler/parcel
[12]https://github.com/parcel-bundler/parcel/issues/1996

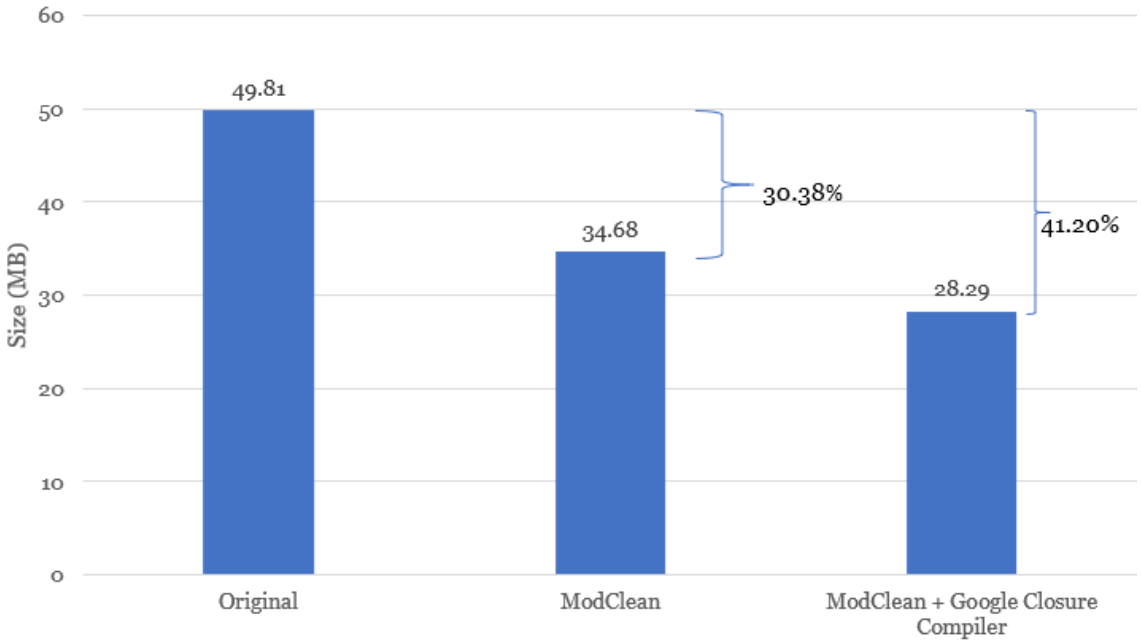

Figure 5.8: `node_module` size reduction using ModClean and GCC

5.4 Evaluation on Container Debloating

Upon reviewing the Cimplifier paper, we noticed that DockerSlim uses a similar technique as Cimplifier to dynamically trace the files used by an application. Of course, Cimplifier can do more apart from dynamic tracing, like partitioning executables into separate container as described in Section 3.11, but for the scope of this evaluation, we only need the dynamic tracing feature. Furthermore, DockerSlim is also open-source and actively maintained, thus it has a larger community of users and documentation.

5.4.1 DockerSlim

For this experiment, we run on four different images: Node.js, Python, NginX (Ubuntu), and NginX (Wholly). Figure 5.9 shows the result, with the average size reduction of 84%.

DockerSlim automatically analyze what files are used by the application and remove other unused files. However, it can be too aggressive and remove some required files. For example, in the Node.js container, it removes some images and CSS files used by the web pages. To fix it, user can give hints to guide DockerSlim's debloating process. In this case, we need to add the folder containing the images so that it would not get removed.

Another interesting observation is on the NginX image after using Wholly. When Wholly creates the container, it pulls the NginX source code and compile it. To compile it requires the whole compiler toolchain and the GNU userland resulting in a large image size. DockerSlim is able to remove all these unnecessary files resulting

in almost the same size as the slimmed NginX (Ubuntu) image. Here, we can see another example of composing debloating tools, similar to what we have described in Section 5.3.

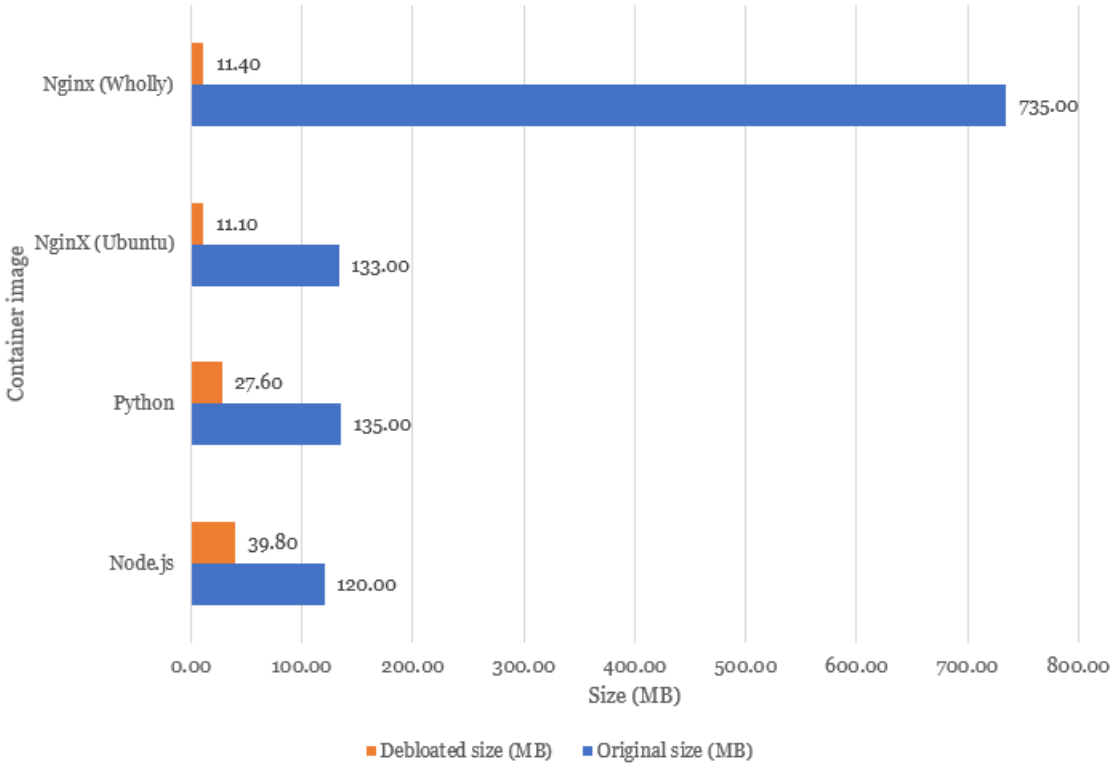

Figure 5.9: Container image size reduction after using DockerSlim. NginX image is reduced by 91.65%, Python by 79.56%, and Node.js by 66.83%.

Next, we take a look at some of the removed files. Figure 5.10 shows the image layer content between the slimmed image versus the original Node.js image. Most of the removed files are extra shared library object (`lib<Name>.so`) and other extra `node_modules` folder that are not used for running the application (but it is required for development).

Moreover, the Node.js image also contains `.git` folder which DockerSlim managed to detect and remove it. This is common during ongoing development where mistakes do happen if a developer forgot to package the code properly for deployment. Therefore, DockerSlim can be a very useful tool to detect these kinds of mistakes automatically.

Next, we look into how container image size affects container startup time. Table 5.4 shows the result between the original NginX Ubuntu image and the smaller image after running DockerSlim.

The smaller image is able to start around 7% faster compared to the original bigger image. Note that we obtained this measurement with both images already present in the machine. In a serverless or edge environment, the images are probably not on the same machine and need to be downloaded and unpacked before running.

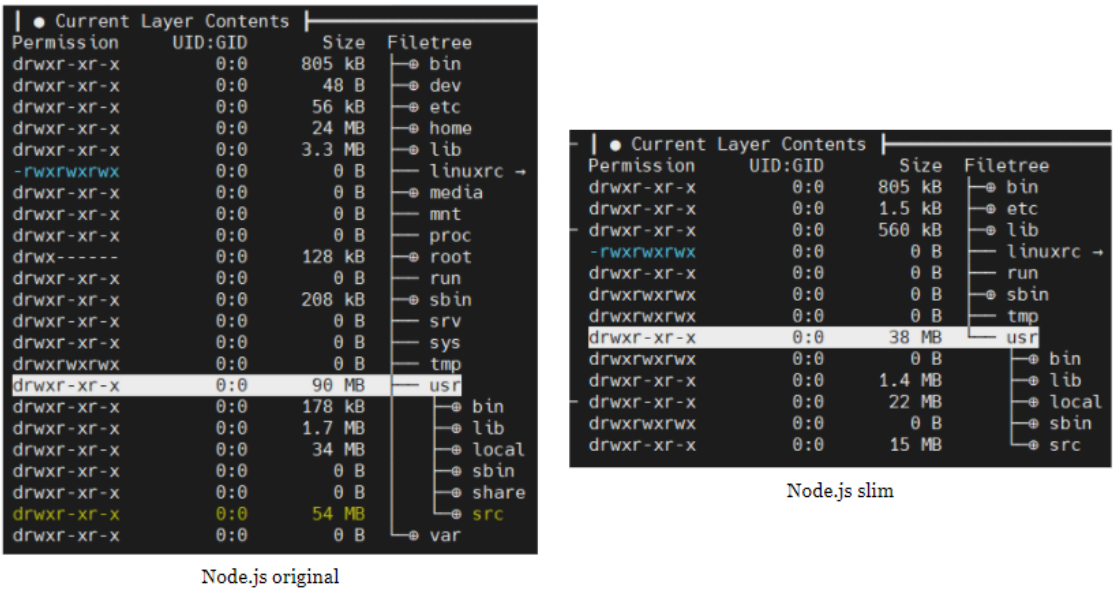

Figure 5.10: Node.js image layer content: original (left) versus slimmed (right).

Container	Startup Time (second)
NginX 1.12 (Ubuntu, 133MB)	2.6877
NginX 1.12 (Slim, 11.1 MB)	2.4959

Table 5.4: The smaller NginX-slim image is able to start 7% faster then the original NginX image.

Thus, including both network access time and image unpacking time, we would have observed a bigger impact on the time difference.

Unfortunately, the slimmed container makes it hard to debug. For example, without an interactive shell, a developer cannot just log into the container to perform further troubleshooting. Fortunately, this can be solved using the side-car pattern where a container with complete debugging tools is augmented with the slimmed container. For containers running in Kubernetes environment, another way is to use the ephemeral debug container[13].

[13]https://kubernetes.io/docs/tasks/debug-application-cluster/debug-running-pod/#ephemeral-container

6

Conclusion

The goal of the thesis was to show bloats in web services, its implication, and possible remedies with existing tools. We conclude our findings with respect to the research question in the next section. Afterward, we describe possible future works.

6.1 Answers to Research Questions

RQ1: Are there bloats in web services?

In Section 5.1, we have shown line-of-code growth rate for the JavaScript libraries and web servers. For container images, although the image size growth rate is not that much but there are still many opportunities to reduce the image size as our result in RQ3 shows.

RQ2: Are there existing debloating tools suitable for web services?

Yes, we have identified some tools relevant for debloating part of web services. The following list shows the tools that we used for each components.

- **Web server**: We used both OCCAM and Wholly to debloat NginX. Both required some efforts to make it work, especially for OCCAM as it requires manual works for every library dependencies.

- **JavaScript**: We used UFFRemover, Webpack tree-shaking, and Google Closure Compiler on a JavaScript library and Node.js application. To make it work, all of them require proper configurations as described in their documentations.

- **Container**: We used DockerSlim for the Python, NginX and Node.js. It can also be used on containers generated by Wholly. We found the project to be very well maintained with good documentation and examples. It automatically detects files that are not used. But it may remove additional files that are loaded in runtime; to prevent this, extra configurations are required.

RQ3: What are the benefits for debloated components in term of storage

reduction and/or performance gain?

In the following list, we describe the benefits using the tools mentioned in the RQ2 result above.

- **Web server**: On average, OCCAM managed to remove more than 80% of unused functions; while Wholly managed to remove around 50% of unused functions. This leads to reduction in the executable binary size with 38% reduction for OCCAM and 40% reduction for Wholly. Regarding request-per-second performance, Wholly is slightly better at around 3%, while OCCAM provides a tiny gain of 0.5%.

- **JavaScript**: We observed around 90% storage reduction on the final JavaScript bundled code. For ModClean and Google Closure Compiler, it managed to reduce the size of `node_modules` by 40%.

- **Container**: We showed that DockerSlim is able to reduce the NginX container image from 735 MB to 11.40 MB. On average, it managed to reduce the size of the container images by 80%. Last but not least, we demonstrate a performance benefit of smaller containers where the debloated NginX container image is able to start 7% faster.

6.2 Future Work

NginX uses a configuration file with its own syntax, or more commonly referred to as a Domain Specific Language (DSL). The idea is to create a debloating tool that understands the grammar of the DSL. The tool can then parse the NginX source code and remove irrelevant codes and logics.

From Figure 4.2, we have worked on both the presentation layer (front-end user interface) and the domain layer (web server) layer, but not the data layer. For example, one can try to use OCCAM on database management systems such as PostgreSQL[1] or MySQL[2].

[1]https://www.postgresql.org/
[2]https://www.mysql.com/

Bibliography

[1] R. Maier, T. Hädrich, and R. Peinl, *Enterprise Knowledge Infrastructures*. Apr. 2009, ISBN: 3540239154. DOI: 10.1007/3-540-27514-2.

[2] J. Kinder and H. Veith, "Precise static analysis of untrusted driver binaries," in *Formal Methods in Computer Aided Design*, 2010, pp. 43–50.

[3] 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*, ser. HotPower '11, Cascais, Portugal: Association for Computing Machinery, 2011, ISBN: 9781450309813. DOI: 10.1145/2039252.2039253. [Online]. Available: https://doi.org/10.1145/2039252.2039253.

[4] W. Commons. "Program memory layout." (2013), [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/5/50/Program_memory_layout.pdf.

[5] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 268–278, ISBN: 9781450322379. DOI: 10.1145/2491411.2491416. [Online]. Available: https://doi.org/10.1145/2491411.2491416.

[6] G. Malecha, A. Gehani, and N. Shankar, "Automated software winnowing," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15, Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1504–1511, ISBN: 9781450331968. DOI: 10.1145/2695664.2695751. [Online]. Available: https://doi.org/10.1145/2695664.2695751.

[7] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 476–486, ISBN: 9781450351058. DOI: 10.1145/3106237.3106271. [Online]. Available: https://doi.org/10.1145/3106237.3106271.

[8] L. Gelle, H. Saidi, and A. Gehani, "Wholly!: A build system for the modern software stack," in *Formal Methods for Industrial Critical Systems*, F. Howar and J. Barnat, Eds., Cham: Springer International Publishing, 2018, pp. 242–257, ISBN: 978-3-030-00244-2.

[9] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, pp. 70–83, ISBN: 9781450376280. DOI: 10.1145/3359789.3359823. [Online]. Available: https://doi.org/10.1145/3359789.3359823.

[10] 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 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1697–1714, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/azad.

[11] M. I. Rahman, S. Panichella, and D. Taibi, "A curated dataset of microservices-based systems," vol. Vol-2520, 2019.

[12] H. Vázquez, A. Bergel, S. Vidal, J. Díaz Pace, and C. Marcos, "Slimming javascript applications: An approach for removing unused functions from javascript libraries," *Information and Software Technology*, vol. 107, pp. 18–29, 2019, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.10.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918302210.

[13] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "Jshrink: In-depth investigation into debloating modern java applications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 135–146, ISBN: 9781450370431. [Online]. Available: https://doi.org/10.1145/3368089.3409738.

[14] U. Goel, S. Ludin, and M. Steiner, "Web performance with android's battery-saver mode," *CoRR*, vol. abs/2003.06477, 2020. arXiv: 2003.06477. [Online]. Available: https://arxiv.org/abs/2003.06477.

[15] M. Alhanahnah, R. Jain, V. Rastogi, S. Jha, and T. Reps, *Lightweight, multi-stage, compiler-assisted application specialization*, 2021. DOI: 10.48550/ARXIV.2109.02775. [Online]. Available: https://arxiv.org/abs/2109.02775.

[16] [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[17] *Dependency Hell | How npm Works*. [Online]. Available: https://npm.github.io/how-npm-works-docs/theory-and-design/dependency-hell.html (visited on 01/16/2022).

[18] *February 2021 Web Server Survey*, en-gb. [Online]. Available: https://news.netcraft.com/archives/2021/02/26/february-2021-web-server-survey.html (visited on 04/19/2021).

[19] *HTTP Archive: State of the Web*, en. [Online]. Available: https://httparchive.org/reports/state-of-the-web (visited on 05/26/2021).

[20] M. Saboff and M. Hoitink, *Built in modules (js standard library)*. [Online]. Available: https://tc39.es/proposal-built-in-modules/ (visited on 10/06/2021).

[21] *Scott Bronson - Performance testing shared vs. static libs.* [Online]. Available: https://gcc.gnu.org/legacy-ml/gcc/2004-06/msg01956.html (visited on 01/17/2022).

[22] *The modern web on a slow connection.* [Online]. Available: https://danluu.com/web-bloat/ (visited on 05/26/2021).

[23] *What is a Container? | App Containerization | Docker*, en. [Online]. Available: https://www.docker.com/resources/what-container (visited on 04/04/2021).

A

Appendix 1

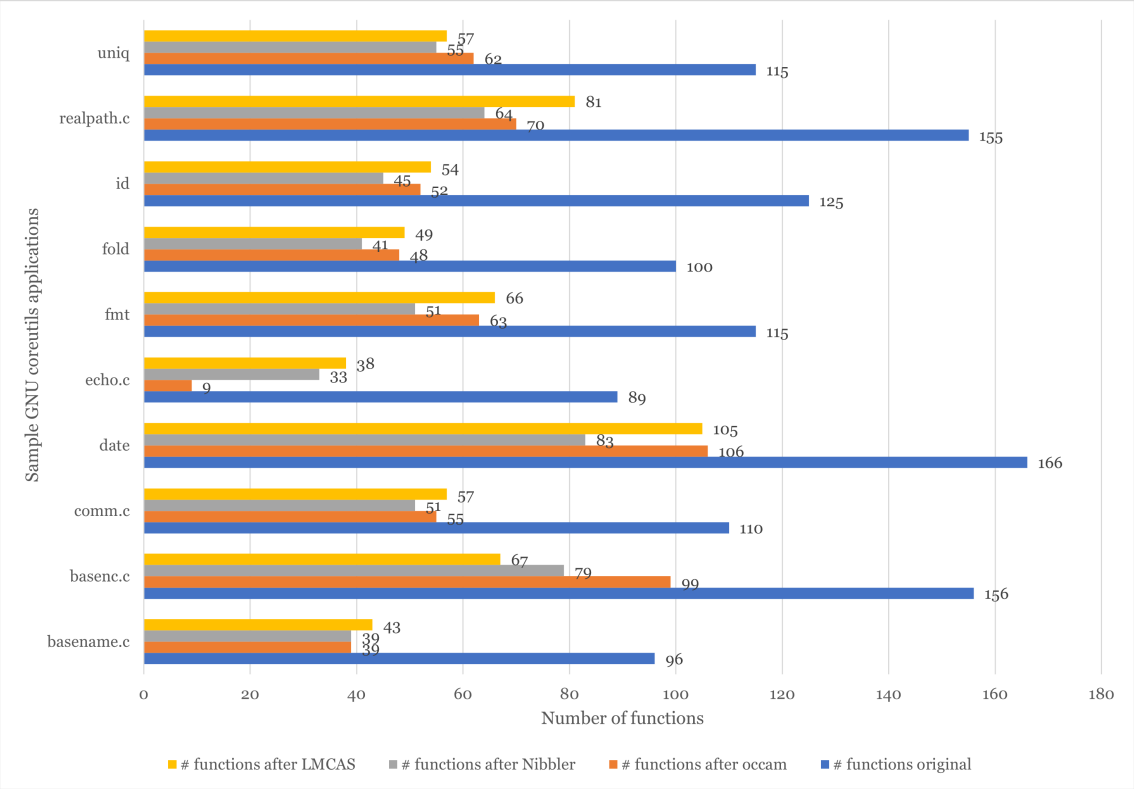

Figure A.1: Comparison of reduction rate between OCCAM, Nibbler, and LMCAS for selected applications in GNU coreutils.

Library	Functions Count	Functions Removed	Reduction Rate
httpd (main app)	1083	468	56.79%
libapr	689	282	59.07%
libaprutil	371	89	76.01%
libpcre	46	18	60.87%

Table A.1: Apache HTTPD after running OCCAM. The "Functions Count" column is the original number of functions before performing OCCAM.

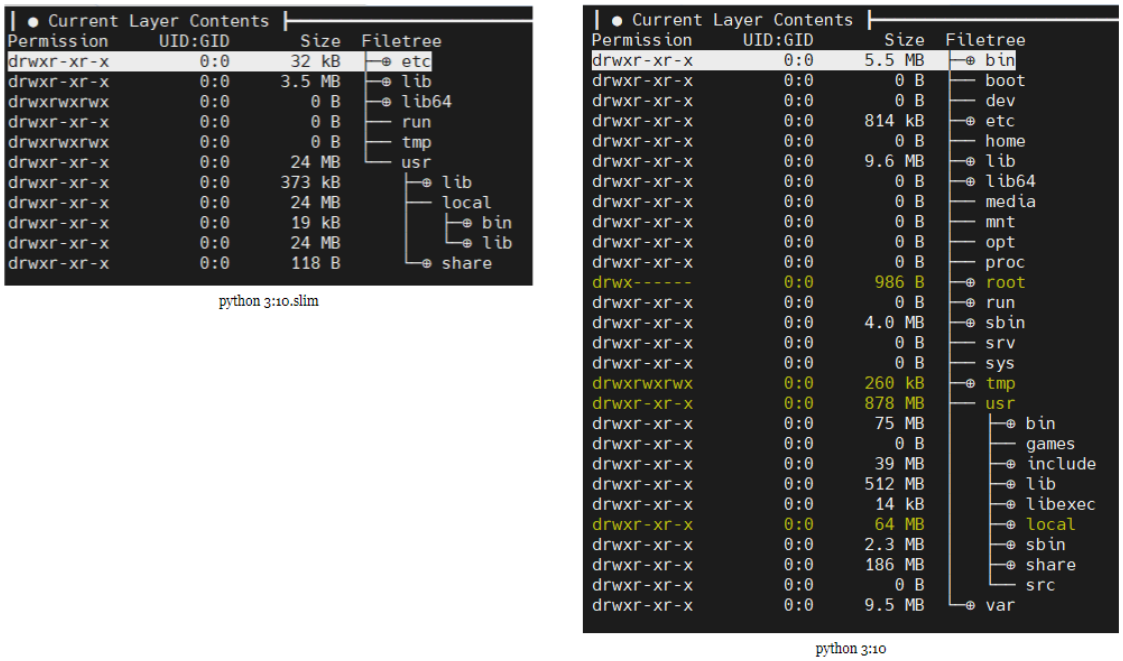

Figure A.2: Python 3.10 image layer content: original (left) versus slimmed (right).

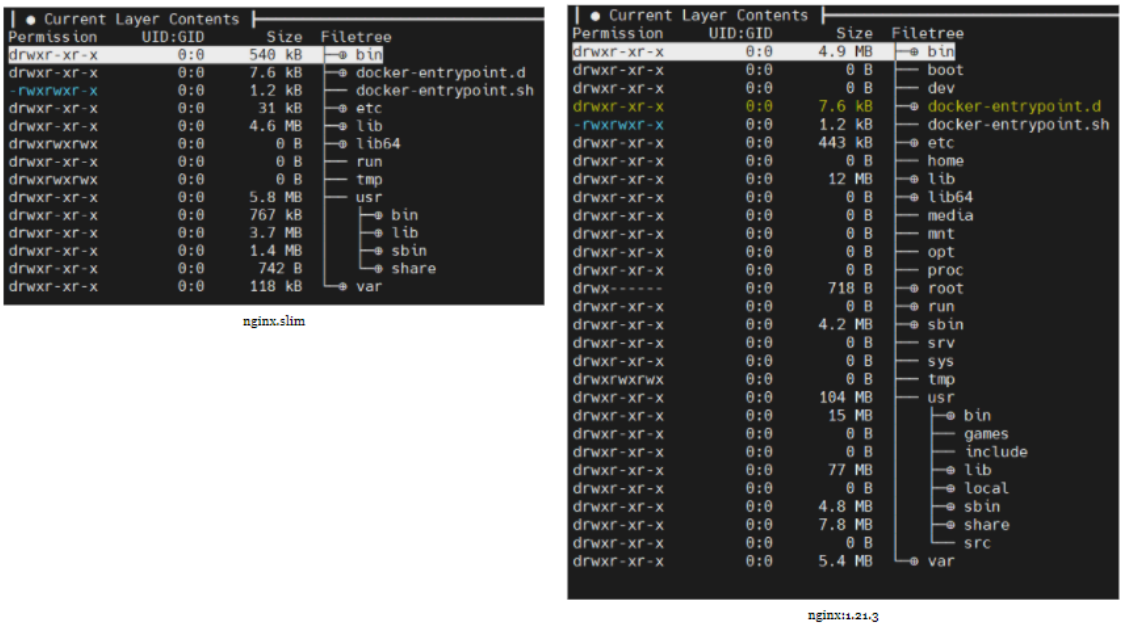

Figure A.3: NginX (Ubuntu) image layer content: original (left) versus slimmed (right).